

Kuormantasauksen soveltaminen Teamboard 2.0 -palvelussa

Jarmo Viinikanoja

Opinnäytetyö
Syyskuu 2014

Tietotekniikan koulutusohjelma
Tekniikan ja liikenteen ala



JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES



Tekijä(t) Viinikanoja, Jarmo	Julkaisun laji Opinnäytetyö	Päivämäärä 18.9.2014
	Sivumäärä 122 + 19	Julkaisun kieli Suomi
		Verkojulkaisulupa myönnetty (X)
Työn nimi Kuormantasauksen soveltaminen Teamboard 2.0 -palvelussa		
Koulutusohjelma Tietotekniikan koulutusohjelma		
Työn ohjaaja(t) Juha Piispanen Mika Rantonen		
Toimeksiantaja(t) N4S@JAMK Marko Rintamäki		
<p>Tiivistelmä</p> <p>Opinnäytetyön toimeksiantaja oli N4S@JAMK, joka on Digilen Need for Speed – tutkimusohjelmassa mukana oleva Jyväskylän ammattikorkeakoulun projekti. Ohjelmassa kokeillaan käytännössä asiakastuntemukseen perustuvia reaaliaikaisia liiketoimintamalleja.</p> <p>Työn tavoitteena oli tutustua kuormantasaajien toimintaan pilviympäristössä sekä selvittää kuinka kuormantasausta voidaan hyödyntää pilvipalvelussa, ja mitä kuormantasausta tehdessä tulee huomioida. Palvelusta esimerkkinä käytettiin N4S@JAMKIN Teamboardia, jolle simuloitiin käyttäjien toimintoja. Testien avulla pyrittiin havaitsemaan palvelun pullonkaulat, jotka muuttamalla palvelu saadaan kestävämmän enemmän kuormaa.</p> <p>Työssä suoritettiin mittauksia, joissa testattiin palvelun yhtäaikaisten käyttäjien kestämistä, kuormantasaajan toimintaa, sekä paikallistettiin pullonkauloja. Tulosten pohjalta tehtiin muutoksia ja niiden hyöty todennettiin mittauksilla.</p> <p>Työn tuloksena palvelusta tehtiin skaalautuva ja siihen lisättiin HAProxy kuormantasaajaksi. Kuormantasauksella ja ongelmakohtien korjaamisella palvelu saatiin kestävämmän 720 yhtäaikaista käyttäjää. Käyttäjien määrä voidaan nostaa helposti tasaamalla kuormaa useammalle koneelle.</p>		
Avainsanat (asiasanat) Kuormantasaus, Teamboard, Locust, HAProxy, N4S@JAMK		
Muut tiedot		



Author(s) Viinikanoja, Jarmo	Type of publication Bachelor's Thesis	Date 18.9.2014
	Pages 122 + 19	Language Finnish
		Permission for web publication (X)
Title Adapting Loadbalancing in Teamboard 2.0 service		
Degree Programme Data Network Technology		
Tutor(s) Piispanen, Juha Rantonen, Mika		
Assigned by N4S@JAMK Rintamäki, Marko		
<p>Abstract</p> <p>The thesis was assigned by N4S@JAMK, a project within JAMK University of Applied Sciences. N4S@JAMK is also a part of research project DIGILE. N4S adopts a real-time experimental business model based upon deep customer insight.</p> <p>The aim of this thesis was to explore the operation of the load balancer in a cloud environment, find out how load balancing can be used in the cloud, and what should be taken into account while deploying load balancer. Teamboard of N4S@JAMK was used as an example of a service, for which user functions were simulated. The tests aimed to identify bottlenecks of the service, so that the service could withstand more load by fixing identified bottlenecks.</p> <p>The thesis carried out measurements, which tested how many simultaneous users the service can handle, how the load balancer works. It also located bottlenecks in the service.-A fix was made for the service based on the results, and their benefits were verified by measurements.</p> <p>As a result, the service was made scalable, and HAProxy was added to load balance the traffic for the service. By using the load balancer and by fixing the problems the service was able to handle 720 concurrent users. The limit of the concurrent users can easily be increased by balancing load on several machines.</p>		
Keywords Load balancing, Teamboard, Locust, HAProxy, N4S@JAMK		
Miscellaneous		

Sisältö

1	Työn lähtökohdat.....	9
1.1	Toimeksiantaja	9
1.2	Tavoitteet	10
2	Pilvipalvelut	10
2.1	Yleistä	10
2.2	Yleistyminen.....	11
2.3	Pilvipalvelujen luokittelu	11
2.4	DigitalOcean.....	12
3	Tiedonsiirtoprotokollat	13
3.1	OSI-malli	13
3.2	Protokollien tyypit	15
3.2.1	Yhteydellinen	15
3.2.2	Yhteydetön.....	15
3.3	TCP	15
3.4	HTTP	17
3.5	WebSocket.....	18
4	Työssä käytettyjä käsitteitä	19
4.1	API	19
4.2	Soketti	19
4.3	Välityspalvelin	20
4.4	DNS.....	21
5	Kuormantasaus	21
5.1	Yleistä	21
5.2	Staatinen kuormantasaus.....	22
5.3	Dynaaminen kuormantasaus.....	22
5.4	Kuormantasaaja (Load Balancer).....	22
5.4.1	Yleistä	22
5.4.2	HAProxy	23
5.4.3	Apache	24
5.4.4	Kuormantasaus algoritmit	24

5.4.5	Tila-tarkastukset	28
6	Ohjelmistotestaus.....	28
6.1	Yleistä	28
6.2	Staatinen analyysi	29
6.3	Dynaaminen analyysi	29
6.4	Testaustavat.....	29
6.5	Ohjelmistotestaus työkalut	30
6.5.1	Apache Jmeter.....	30
6.5.2	LoadUI ja SoapUI	30
6.5.3	Locust	30
7	Teamboard.....	31
7.1	Yleistä	31
7.2	Komponentit	32
7.2.1	Socket .IO	32
7.2.2	Redis	33
7.2.3	MongoDB.....	33
7.2.4	Bcrypt	33
8	Toteutus	34
8.1	Vaatimukset ja lähtökohdat	34
8.2	Testausohjelmistojen asennus	34
8.2.1	HAProxy	34
8.2.2	Apache	35
8.2.3	SoapUI & LoadUI	36
8.2.4	JMeter.....	36
8.3	Testien luonti	37
8.3.1	SoapUI.....	37
8.3.2	LoadUI	40
8.3.3	JMeter.....	41
8.4	Kuormantasaajien vertailu	43
8.4.1	Yleistä	43
8.4.2	HAProxy vs Apache-testi	44
8.5	Mittaus 1: Alkutilanne	46

8.5.1	Suunnittelu.....	46
8.5.2	Toteutus.....	46
8.5.3	Tulokset	47
8.5.4	Tulosten analysointi	51
8.6	Mittaus 2: Tietokannan erottaminen	52
8.6.1	Suunnittelu.....	52
8.6.2	Toteutus.....	53
8.6.3	Tulokset	56
8.6.4	Analysointi	59
8.7	Mittaus 3: Kuormantasaajan testaaminen	59
8.7.1	Suunnittelu.....	59
8.7.2	Toteutus.....	59
8.7.3	Tulokset	60
8.7.4	Analysointi	63
8.8	Mittaus 4: Sisäänkirjaus vs taulun luonti.....	63
8.8.1	Suunnittelu.....	63
8.8.2	Toteutus.....	64
8.8.3	Tulokset	65
8.8.4	Analysointi	68
8.8.5	Jatkotoimenpiteet.....	68
8.9	Mittaus 5: Tehokäyttäjien simulointitestit	69
8.9.1	Suunnittelu.....	69
8.9.2	Testiympäristön valmistelu.....	70
8.9.3	Locust-asennus	71
8.9.4	Locust-testin luonti.....	72
8.9.5	Toteutus.....	77
8.9.6	Tulokset	78
8.9.7	Tulosten analysointi.....	88
8.10	Mittaus 6: Rauhallisten käyttäjien simulointitestit	90
8.10.1	Suunnittelu	90
8.10.2	Toteutus	90
8.10.3	Tulokset.....	90

8.10.4	Analysointi	93
8.11	Mittaus 7: Tehokäyttäjien simulointi yksityisillä tauluilla	94
8.11.1	Suunnittelu	94
8.11.2	Toteutus	94
8.11.3	Tulokset.....	95
8.11.4	Analysointi	96
8.12	Mittaus 8: Brypt-palvelin erotettuna.....	98
8.12.1	Suunnittelu	98
8.12.2	Toteutus	98
8.12.3	Tulokset.....	98
8.12.4	Analysointi	101
8.13	Mittaus 9: Shardien lisääminen tietokantaan.....	102
8.13.1	Suunnittelu	102
8.13.2	Toteutus	103
8.13.3	Tulokset.....	103
8.13.4	Analysointi	108
8.14	Mittaus 10: Kuorman jakaminen tasaisesti.....	109
8.14.1	Suunnittelu	109
8.14.2	Toteutus	109
8.14.3	Tulokset.....	110
8.14.4	Analysointi	111
9	Pohdinta	111
9.1	Johtopäätökset	111
9.2	Tulokset.....	114
9.3	Jatkokehitys	115
9.4	Saatujen tulosten luotettavuus	116
9.5	Hyöty	117
	Lähteet	119
	Liitteet.....	123
	Liite 1. HAProxyn asetukset.....	123
	Liite 2. Apachen asetukset.....	125
	Liite 3. Locust testin asetukset.....	127

Liite 4. Mittausten 5-9 tulokset koottuna.....	131
Liite 5. Mittauksen 7 tulokset.....	132
Liite 6. Mittauksen 8 tulokset.....	137

Kuviot

Kuvio 1.OSI-malli.....	13
Kuvio 2. TCP-kehys.....	16
Kuvio 3. Round Robin –menetelmä.....	25
Kuvio 4. Painotettu Round Robin –menetelmä.....	26
Kuvio 5. Painotettu vähiten yhteyksiä –menetelmä	27
Kuvio 6. Teamboardin workspace -näkymä.....	31
Kuvio 7. Teamboardin taulu -näkymä.....	32
Kuvio 8. SoapUI HTTP-yhteyden testaaminen	37
Kuvio 9. Login-vaiheen luonti	38
Kuvio 10. Login-vaiheen testaaminen	39
Kuvio 11. getToken-komentosarja.....	39
Kuvio 12. Taulunluonti-vaiheen asetukset.....	40
Kuvio 13. LoadUI-testi.....	41
Kuvio 14. Jmeter, käyttäjien luonti ja määrän asettaminen.....	42
Kuvio 15. Jmeter, Login-testin luominen	42
Kuvio 16. Jmeter, Login-testin graafinen seuraaminen	43
Kuvio 17. HAProxyn tilastot.....	44
Kuvio 18. HAProxy vs Apache 750 käyttäjällä.....	45
Kuvio 19. Ympäristön rakenne alkutilanteessa.....	46
Kuvio 20. Teamboard-tilastot testin aikana.....	48
Kuvio 21. HAProxyn tilastot testin aikana.....	49
Kuvio 22. Kuormitustesti suoraan SoapUI:sta	50
Kuvio 23. LoadUI- ja SoapUI-testien aiheuttamat rasitukset Teamboardille...	51
Kuvio 24. Muutosten vaikutus ympäristöön	56
Kuvio 25. Mongo Shardin prosessorin tilastot testin aikana	56
Kuvio 26. Jmeter Login-testin graafinen seuraaminen uudelle Teamboardille	57
Kuvio 27. Testien aiheuttamat rasitukset uudistetulle Teamboardille	58

Kuvio 28. Tulokset SoapUI:lla 900 käyttäjää yhtä API-konetta vastaan	60
Kuvio 29. Tulokset SoapUI:lla 900 käyttäjää kahta API-konetta vastaan	61
Kuvio 30. Tulokset SoapUI:lla 900 käyttäjää kaksiytimistä API-konetta vastaan	62
Kuvio 31. SoapUI 900 sisäänkirjautumista	65
Kuvio 32. SoapUI 900 taulunluontia	66
Kuvio 33. 900 sisäänkirjautumista yhdelle koneelle	67
Kuvio 34. 4000 taulun luonnin rasitus yhdelle koneelle	68
Kuvio 35. Hajautettu Teamboard.....	69
Kuvio 36. Sisäisen nimipalvelimen käyttö Locustille.....	71
Kuvio 37. 250 käyttäjän testi yhdelle API-koneelle	78
Kuvio 38. 250 käyttäjän testi kahdelle API-koneelle	79
Kuvio 39. 250 käyttäjän testi kolmelle API-koneelle	79
Kuvio 40. 250 käyttäjän testi kolmelle API-koneelle	80
Kuvio 41. Testin tulokset eri API-konemäärillä 250 käyttäjällä	81
Kuvio 42. 500 käyttäjän testi kahdelle API-koneelle	82
Kuvio 43. 500 käyttäjän testi kolmelle API-koneelle	83
Kuvio 44. 500 käyttäjän testi neljälle API-koneelle	83
Kuvio 45. Prosessorin käyttö eri API-konemäärillä 500 käyttäjällä	84
Kuvio 46. 750 käyttäjän testi kahdelle API-koneelle	85
Kuvio 47. "Failures"-välilehti; 750 käyttäjää, 2 API-konetta	85
Kuvio 48. 750 käyttäjän testi kahdelle API-koneelle, virheiden jälkeen	86
Kuvio 49. 750 käyttäjän testi kolmelle API-koneelle	86
Kuvio 50. 750 käyttäjän testi neljälle API-koneelle	87
Kuvio 51. Prosessorin käyttö eri API-konemäärillä 750 käyttäjällä	87
Kuvio 52. 500 rauhallista käyttäjää, 4 API-konetta	91
Kuvio 53. 750 rauhallista käyttäjää, 4 API-konetta	91
Kuvio 54. 1000 rauhallista käyttäjää, 4 API-konetta	92
Kuvio 55. Kaikkien rauhallisten käyttäjätestien aiheuttamat kuormitukset	92
Kuvio 56. Testien aiheuttamat rasitukset API-koneen prosessorille	99
Kuvio 57. Testien aiheuttamat rasitukset Shard-koneen prosessorille	100
Kuvio 58. Testien aiheuttamat rasitukset HAProxy-koneen prosessorille	100

Kuvio 59. Testien aiheuttamat rasitukset Bcrypt-koneen prosessorille.....	101
Kuvio 60. Neljä API-konetta, 1500 käyttäjää.	103
Kuvio 61. Shard1-koneen prosessorin rasitus	104
Kuvio 62. Shard2-koneen prosessorin rasitus	104
Kuvio 63. Viisi API-konetta, 1500 käyttäjää	105
Kuvio 64. 1500 käyttäjän aiheuttama rasitus prosessorille viidellä ja neljällä API-koneella	105
Kuvio 65. Viisi API-konetta, 1750 käyttäjää	106
Kuvio 66. Shard1-koneen prosessorin rasitus 1750 käyttäjällä	106
Kuvio 67. Shard1-koneen prosessorin rasitus 1750 käyttäjällä	107
Kuvio 68. HAProxy-koneen prosessorin rasitus 1750 käyttäjällä.....	107
Kuvio 69. API1-koneen prosessorin rasitus 1750 käyttäjällä	108
Kuvio 70. HAProxy:n statistiikkasivu, kuorman jakaantumisen ongelma	108
Kuvio 71. Viisi API-konetta, 1750 käyttäjää, vähiten yhteyksiä -menetelmä.	110
Kuvio 72. HAProxy:n statistiikkasivu, maksimi istuntojen jakautuminen	110
Kuvio 73. Hintavertailu yksi ja kaksiytimisten koneiden välillä	113

Taulukot

Taulukko 1. 250 tehokäyttäjän kuormitustestin tulokset	82
Taulukko 2. 500 tehokäyttäjän kuormitustestin tulokset	84
Taulukko 3. 750 tehokäyttäjän kuormitustestin tulokset	88
Taulukko 4. 250 käyttäjää privaateilla tauluilla	95
Taulukko 5. 500 käyttäjää privaateilla tauluilla	95
Taulukko 6. 750 käyttäjää privaateilla tauluilla	95
Taulukko 7. Tulokset Bcryptin ollessa erillisenä palveluna.	99
Taulukko 8. Hintavertailu ydinten määrän mukaan.....	112

Lyhenteet

ACK	Acknowledge
AJP	Apache JServ Protocol
API	Application Programming Interface
BSD	Berkeley Software Distribution
DNS	Domain Name System
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IETF	The Internet Engineering Task Force
IP	Internet Protocol
OSI-malli	Open System Interconnection model
PaaS	Platform as a Service
RAM	Random Access Memory
RPS	Requests Per Second
SaaS	Software as a Service
SSL	Secure Sockets Layer
SYN	Synchronize
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

1 Työn lähtökohdat

1.1 Toimeksiantaja

Työn toimeksiantaja on N4S@JAMK. N4S@JAMK on Jyväskylän ammattikorkeakoulun projekti, joka on mukana Need for Speed ohjelmassa.

Digilen käynnistämän Need for Speed (N4S) -tutkimusohjelman tavoitteena on antaa yrityksille keinoja ohjelmistoliikennetoiminnan nopeuttamiseksi. Ohjelmassa kokeillaan käytännössä reaaliaikaisia liiketoimintamalleja, jotka perustuvat asiakastuntemukseen. Ohjelman toteuttavat suomalaiset ohjelmistoyritykset. Digile-ohjelma kestää neljä vuotta ja sen rahoittaa Tekes. (N4S-ohjelma 2014.)

Ohjelmalla on kolme painopistettä (N4S-ohjelma 2014.):

Arvon tuominen reaaliajassa: Suomalaisessa ohjelmistointensiivisessä teollisuudessa on menty kohti arvopohjaista ja mukautuvaa reaaliaikaista liiketoiminnan mallia. Muutoksen tukemiseksi on luotu tekninen infrastruktuuri ja sille vaadittavat ominaisuudet.

Syvällinen asiakastuntemus – parempi tuotto: Hyödynnetään moninaisia data- ja tietolähteitä jotta saadaan hankittua syvällistä tietämystä asiakkaiden tarpeista ja käyttäytymisestä. Tiedon avulla voidaan parantaa myyntiä ja saada tuottoa panostamisesta tuotteiden ja palveluiden kehittämistyöhön.

”Elohopeabisnes” – Uutta rahaa etsimässä: Keskittyy siihen, kuinka yritykset ja yhteisöt voivat käyttäytyä nestemäisen elohopean tavoin löytäen ja valaten uusia uria. Se merkitsee kykyä mukautua uusiin liiketoiminnan olosuhteisiin ja etsiä aggressiivisesti uusia mahdollisuuksia uusilla markkinoilla vähäisellä vaivalla. Tämän mahdollistaa jatkuva ja aktiivinen strateginen painostus.

1.2 Tavoitteet

Työn tavoitteena oli tutustua kuormantasaajien toimintaan pilviympäristössä, perehtymällä kuinka pilviympäristössä voidaan hyödyntää kuormantasausta ja mitä kuormantasausta tehdessä tulee ottaa huomioon. Lisäksi tutkittiin tapoja jolla pystytään tuottamaan kuormitusta pilvessä sijaitsevalle palvelulle. Esi-merkkinä palvelusta käytettiin N4S@JAMKin Teamboardia, jolle kuormantasaushjelmistoilla pyrittiin simuloimaan käyttäjien toimintoja. Kuormitustestien avulla testattiin, kuinka palvelu toimii erilaisten kuormitusten alla, sekä etsittiin pullonkauloja, jotka muuttamalla palvelu saadaan kestäämään enemmän kuormaa. Toimeksiantajan lopullisena tavoitteena oli, että Teamboard saataisiin todistetusti kestäämään 10000 yhtäaikaista käyttäjää.

2 Pilvipalvelut

2.1 Yleistä

Pilvipalvelu eli cloud computing -termi tulee tietoliikenne- ja puhelinverkkojen dokumentointitavasta. Verkot sisältävät useita laitteita ja ovat monimutkaisia, siksi verkko yleensä esitetään yksinkertaisuuden vuoksi pilvenä. (Heino 2010, 32-33.)

Pilvipalvelu tarkoittaa ohjelmiston, infrastruktuurin tai alustan tarjoamista ”pilvestä”. Sen sijaan, että laitteet asennettaisiin omaan konesaliin, vuokrataan tilaa pilvipalvelusta, jonne laitteet asennetaan. Pilvipalvelun etuja ovat skaalautuvuus, nopea käyttöönotto ja luotettavuus. (Pilvipalvelut 2014a.)

Pilvipalvelut ovat yleensä paikasta riippumattomia palveluja, joita käytetään verkon kautta. Organisaation tai henkilön tarvitsemat sovellukset, tiedot tai

palvelut sijaitsevat siis pilvipalvelun tarjoajan palvelimella. (Aalto-yliopiston pilvipalveluohje 2011, 1.)

Pilvipalveluissa sovellusten ja palveluiden ylläpito on pilvitarjoajan vastuulla, joten yritys voi keskittyä ydinliiketoimintaansa. Pilvipalvelut tuovat myös kustannustehokkuutta, koska palvelun ostaja säästää laitteisto- ja ohjelmistolienssikustannuksissa sekä ylläpitokustannuksissa. Asiakas myös maksaa palvelusta käyttömäärän perusteella ja pilvipalvelut saadaan skaalautumaan asiakkaan tarpeiden mukaisesti. (Pilvipalvelut 2014b.)

2.2 Yleistyminen

Suomalaisista organisaatioista 80 % hyödyntää pilvipalveluja. Pilvipohjaisten ohjelmistojen (SaaS) hyödyntäminen on yleistymisen kärjessä. Myös infrastruktuuripalvelujen (laas) hyödyntäminen on yleistymässä. (Pilvipalvelut työntyy IT-ostamiseen 2014.)

Tärkeimpiä syitä pilvipalveluiden hyödyntämiseen ovat toiminnallisuus, kustannustehokkuus ja tietoturva. Pilvipalveluiden odotetaan tuovan parempaa toiminnallisuutta, joustavuutta, kustannustehokkuutta ja nopeuttavan käyttöönottoa. (Pilvistrategia suomalaisissa yrityksissä 2013.)

2.3 Pilvipalvelujen luokittelu

Platform as a Service (PaaS)

Koneiston tarjoajalla on täysin virtuaalinen palvelinympäristö. Asiakkaalle vuokrataan osa palveluista, jolloin asiakas voi hyödyntää koneiston kapasiteettiä ja työkaluja API-ohjelmointirajapinnan kautta. PaaS on hyödyllisin asiakkaille, jotka pystyvät itse rakentamaan haluamansa sovellukset. (Heino 2010, 50–52.)

Infrastructure as a Service(IaaS)

Palveluntarjoaja ylläpitää internetissä virtuaalista konesalia tai konesaleja. Asiakas voi sitten vuokrata käyttöönsä etukäteen määriteltyjä ja hinnoiteltuja osioita. Asiakaskäyttöliittymä käsittää hallintakonsolin ja komentorivityökaluja, joilla asiakas muodostaa yhteyden osioihinsa. Asiakas perustaa lohkoihin käyttöjärjestelmät, joihin sitten voi asentaa halutut sovellukset. Loppukäyttäjä yleensä muodostaa yhteyden palveluun selaimella, mutta on olemassa muitakin tapoja. Tarjoaja voi myös rakentaa asiakkaalle dedikoituja fyysisiä palvelimia. Tunnetuin IaaS-palveluntarjoaja on Amazon Web Services. (Heino 2010, 52–53.)

Software as a Service(SaaS)

Tässä palvelumuodossa asiakas haluaa itselleen vain sovelluksen, joka jaetaan loppukäyttäjälle tietoliikenneyhteyden kautta selaimen. Palveluntarjoaja huolehtii kaikesta muusta. Palveluntarjoaja hyödyntää elastista provisointia, virtualisointia ja jaetun ympäristön tapoja pitääkseen tuotantokustannuksia alhaisempana, kuin mihin asiakas itse pystyisi. Etuna on se, että palveluntarjoaja on jo rutinoitunut vaikeisiin tilanteisiin, ja asiakas pystyy keskittymään ohjelmiston seurantaan ja asiakkaiden hallintaan. (Heino 2010, 53-54.)

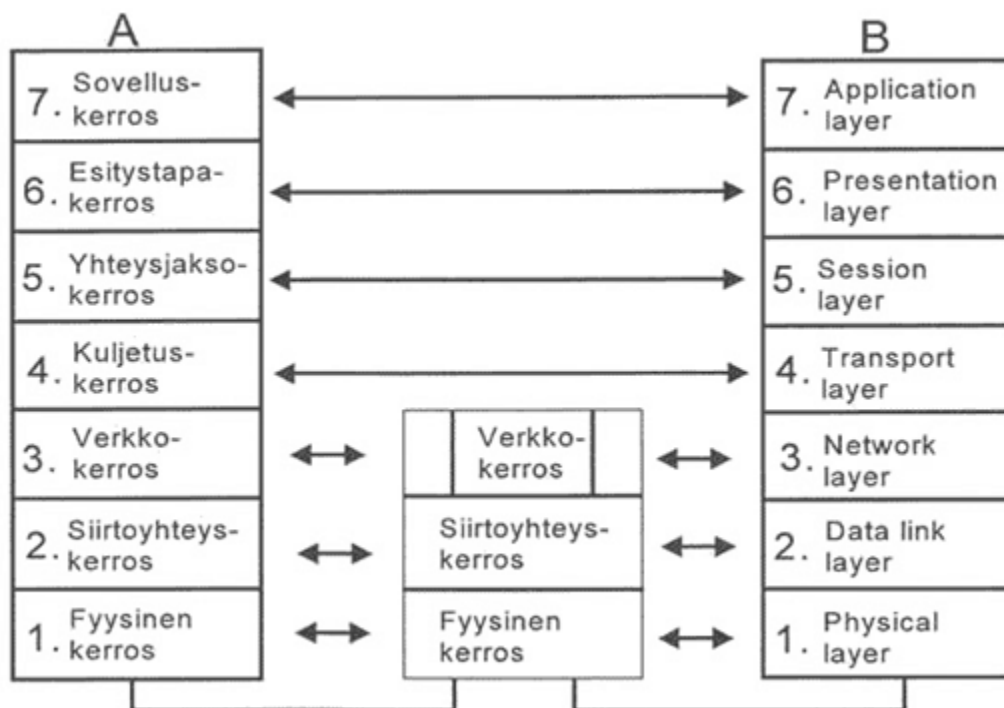
2.4 DigitalOcean

DigitalOcean on sovellusalusta, joka on luotu tuotekehittäjille, joiden tarvitsee käynnistää ja skaalata sovelluksiaan nopeasti. Se tarjoaa tuotekehittäjille myös hyvän ympäristön komentorivin testailuun sekä omien palvelinten kustomoinnin opettelemiseen. DigitalOceanin etuja ovat hinnat ja helppokäyttöisyys, edullisimmat koneet maksavat 5 \$ kuukaudessa. Hallintapaneeli on yksinkertainen. Käytössä on myös DigitalOceanin oma API-ohjelmointirajapinta. (Introduction 2014.)

3 Tiedonsiirtoprotokollat

3.1 OSI-malli

OSI-malli (Open System Interconnection model) on kansainvälisen standardointijärjestön 1970-luvun lopulla aloittaneen avointen tietojärjestelmien liitäntämällin kehittämisen tulos. Se määrittelee tietoliikennejärjestelmän kerrosteisen verkkorakenteen. Se tarjoaa peruskäsitteistön ja toimii tietoliikenteen standardien perustana. Siinä on määritelty kehykset tietoliikenteen standardintyölle ja protokollien suunnittelulle. OSI-malli jakaa viestinnän seitsemään kerrokseen kuvion 1 mukaisesti. (OSI-malli n.d.)



Kuvio 1. OSI-malli (OSI-malli n.d.)

Kerrokset ovat riippumattomia viereisistä, mutta toisaalta ne tukeutuvat alla olevan kerroksen palveluihin. Jokaisella kerroksella on oma rajapintansa, jon-

ka kautta kerros voi antaa ylä- ja alapuolella oleville kerroksille määritettyjä pyyntöjä tai ilmoituksia. (OSI-malli n.d.)

OSI-mallin kerrosten tehtävät ovat seuraavat (The OSI Models's Seven Layers Defined and Functions Explained 2014):

Sovelluskerros toimii rajapintana sovellusten ja tiedonsiirtämisen välillä. Se määrittelee sovelluksien väliselle kommunikaatiolle rajapinnan verkkoon. Esimerkiksi sähköpostin katsotaan kuuluvan tälle kerrokselle.

Esitystapakerros määrittelee, millaista tietoa voidaan siirtää sovellusten välissä kommunikaatioissa. Se voi tarjota myös tiedon pakkauksen tai muita palveluja tiedon esitysmuodon muuttamiselle.

Istuntokerros tarjoaa palvelut yhteyden muodostamiseen ja purkamiseen sekä vuorojen jakamiseen keskusteluun ja lähettämiseen. Istuntokerroksen ansiosta keskeytyneitä siirtoja tai yhteyksiä voidaan jatkaa. Se siis täydentää kuljetuskerroksella tapahtuvia asioita.

Kuljetuskerros tarjoaa tietoturvan ja huolehtii datapakettien saapumisesta oikeassa järjestyksessä. Tarvittaessa se hoitaa myös pakettien uudelleen lähettämisen.

Verkkokerros hoitaa reitittämisen sekä mahdollistaa verkko-osoitteistot. Se tarjoaa palvelua esimerkiksi kääntämään loogiset osoitteet tai nimet fyysisiksi osoitteiksi

Siirtoyhteyshierros mahdollistaa virheettömän tiedonsiirron fyysisen kerroksen yli.

Fyysinen kerros huolehtii bittivirran siirtymisestä fyysiseltä laitteelta toiselle. Se määrittelee sähköiset, optiset, mekaaniset ja toiminnalliset rajapinnat fyysiselle laitteistolle.

3.2 Protokollien tyypit

3.2.1 Yhteydellinen

Yhteydellisessä protokollassa luodaan osapuolten välille suora yhteys. Molemmat osapuolet ovat koko ajan tietoisia yhteyden tilasta. Yhteydellisen protokollan tietopakettia kutsutaan segmentiksi. Se tarjoaa enemmän palveluja yhteydettömään verrattuna. Se käyttää enemmän verkkoa kuin yhteydetön, koska paketteja täytyy siirtää myös yhteyksien hallitsemiseksi. (Protokollatyypit n.d.)

3.2.2 Yhteydetön

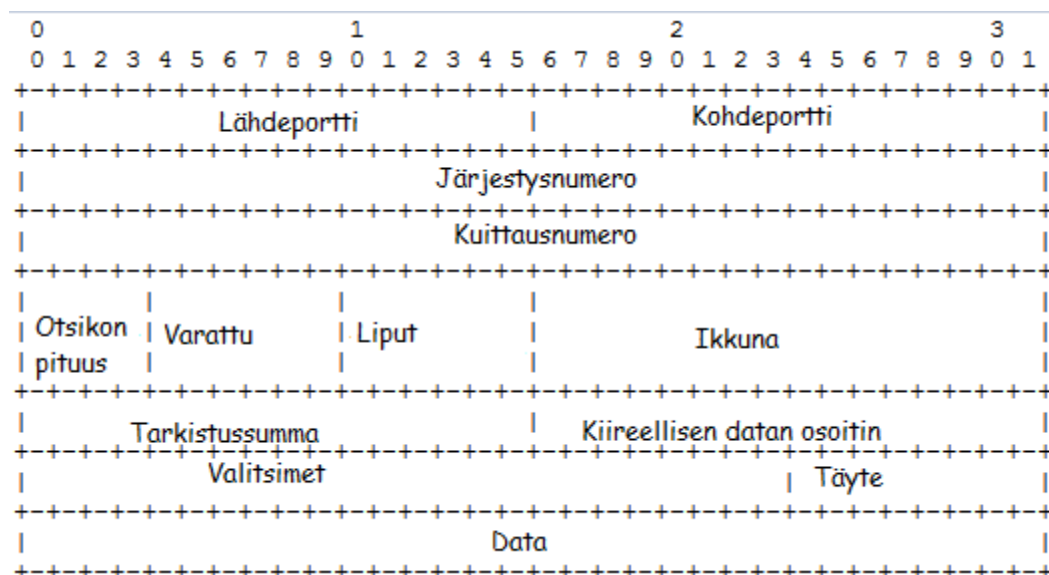
Yhteydettömässä protokollassa ei valmistella tiedonsiirtoa vastapuolen kanssa. Data välitetään saman tien eteenpäin, ja se voikin kulkea eri reittejä eri kerroilla. Yhteydettömän protokollan tietopakettia kutsutaan tietosähkeeksi. Etuja yhteydettömässä protokollassa ovat sen nopeus, ja se tuottaa vähemmän liikennettä kuin yhteydellinen protokolla, koska ei tarvita paketteja yhteyksien hallitsemiseksi. (Protokollatyypit n.d.)

3.3 TCP

TCP-lyhenne tulee sanoista Transmission Control Protocol, määritetty IETF-organisaation standardissa RFC793. TCP on yhteydellinen protokolla, eli ennen itse datan lähettämistä koneiden välillä luodaan koneiden välille yhteys. TCP käyttää yhteyden muodostamiseen kolmitiekättelyä. Kolmitiekättelyssä yhteyden aloittaja lähettää ensin SYN-paketin kohdelaitteelle. SYN-paketin

saatuaan kohdelaite vastaa lähettämällä SYN/ACK-paketin. Kun aloittaja saa SYN/ACK-paketin, se lähettää ACK-paketin vastauksena kohdelaitteelle. TCP-kehys on 32 bitin moninkerta. Luotettavuus TCP:ssä tulee sen sanomissa oleviin numerointiin ja niiden kuittaamiseen. Vastaanottaja lähettää kuittaussessa tiedon, mitä tavua se odottaa lähettäjältä seuraavaksi. Mikäli kuittausta ei kuulu lähettäjälle, mikä voi johtua paketin sisällön muuttumisesta virheelliseksi, tai ettei paketti ole päässyt perille, lähettäjä tietyn odotusajan jälkeen lähettää uudelleen viimeisessä kuittaussanomassa olleesta tavusta alkaen kaiken uudelleen. (Koivisto 2014.)

TCP-kehysen rakenne käy ilmi kuviosta 2.



Kuvio 2. TCP-kehys (RFC793 1981.)

Jokainen datapaketti sisältää kuittauskentän, joten molempien osapuolten lähettäessä dataa kuittaussanomia ei tarvitse lähettää erikseen, koska ne menevät datasanomien mukana (Koivisto 2014).

3.4 HTTP

HTTP-lyhenne tulee sanoista Hypertext Transfer Protocol eli hypertextin siirtoprotokolla. HTTP on yhteydetön protokolla, jossa asiakas lähettää pyyntöjä palvelimelle ja palvelin lähettää vastauksen takaisin. HTTP-protokollan oletusportti on 80, mutta muitakin portteja voidaan käyttää. (RFC2616 1999.)

HTTP:n käyttämiä metodeja ovat seuraavat (RFC2616 1999):

GET – Haetaan tietoa. Jos pyynnössä viitataan dataa tuottavaan prosessiin, palautetaan tuotettu data kokonaisuutena.

POST – Lähetetään jotain palveluun, esimerkiksi pyyntöjä tai lomakkeen tietoja.

OPTIONS – Käytetään kysymään palvelimen tai resurssien ominaisuuksia

HEAD – Kuin GET mutta palvelin palauttaa pelkät otsikkotiedot.

PUT – Tallennetaan kokonaisuuksia. Joko muokataan olemassa olevaa lähdettä, tai mikäli ei ole aiempaa lähdettä, niin luodaan uusi jos mahdollista.

DELETE – Poistetaan kokonaisuuksia.

TRACE – Käytetään pyyntöjen palauttamiseen sellaisenaan, kuin minä palvelin ne vastaanottaa. Tätä voidaan hyödyntää testaamisessa ja diagnosoinnissa.

CONNECT – Käytetään päälle jäävän yhteyden pyytämisessä, esimerkiksi SSL-tunneloinnissa.

Vastauskoodien ensimmäinen numero määrittelee, mihin kategoriaan vastaukset kuuluvat:

1xx: Informatiivinen, pyyntö saatu jatketaan prosessointia

2xx: Onnistuminen, toiminto vastaanotettu, ymmärretty ja hyväksytty.

3xx: Uudelleenohjaus, lisätoimintoja vaaditaan pyynnön toteuttamiseksi.

4xx: Asiakasohjelman virhe, pyynnössä väärä syntaksi tai pyyntöä ei voida toteuttaa.

5xx: Palvelimen virhe, palvelin epäonnistui toteuttamaan oikeanlaisen pyynnön. (RFC2616 1999.)

3.5 WebSocket

WebSocket on TCP -pohjainen protokolla, joka käyttää HTTP-protokollaa kättelyvaiheessa. WebSocket –protokollalla on kaksi vaihetta: kättely ja tiedonsiirto. Yhteyden muodostamiseksi asiakasohjelma lähettää viestin (RFC 2011):

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Palvelin vastaa kyseiseen pyyntöön lähettämällä viestin:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Kun asiakas sekä palvelin ovat lähettäneet kättelyviestit, aloitetaan tiedonsiirto vaihe. Tiedonsiirto vaiheessa käytössä on kaksisuuntainen kommunikaatio kanava jossa molemmat osapuolet voivat lähettää tietoa riippumatta toisesta osapuolesta. (RFC 2011).

4 Työssä käytettyjä käsitteitä

4.1 API

API-lyhenne tulee englannin kielen sanoista Application Programming Interface. Se on ohjelmointirajapinta, esimerkiksi sovellus tarvitsee sitä kutsuakseen pilvestä resurssia tai toiminnetta. (Heino 2010, 76.)

API toimii rajapintana eri ohjelmien välillä, jolloin ohjelmat voivat hyödyntää toisiensa palveluja ja resursseja. Esimerkiksi kirjan jakelijat voivat tarjota kaukoille ohjelman, jota käyttämällä kauppias näkee kirjojen tilanteen varastossa. Jakelijat voisivat myös tarjota API:n, joka suoraan tarkastaa kirjojen saatavuuden varastosta. Etuna olisi, että kysely voitaisiin tehdä myös kaupan omasta ohjelmasta, eikä ohjelmaa tarvitsisi vaihtaa jokaisen eri jakelijan kohdalla. Toinen etu olisi, että jakelijat voisivat muuttaa omaa sisäistä järjestelmäänsä miten haluaa, kunhan sen API-rajapinnan toiminta pysyy samana. Eli API on kirjasto komentoja ja käytänteitä, joita voidaan suorittaa järjestelmälle. Se mahdollistaa järjestelmien resurssien ja komentojen käyttämisen järjestelmän ulkopuolelta. (What is an API? 2012.)

4.2 Soketti

Soketti (socket) on päätepiste kaksisuuntaisessa kommunikaatiolinkissä kahden verkossa olevan ohjelmiston välillä. Esimerkiksi pysyvän kommunikaatio-

tiokanavan muodostamiseen verkossa asiakas-palvelin ohjelmiston kanssa täytyy sekä asiakas- että palvelinohjelmiston muodostaa yhteys toisiinsa. Molemmat ohjelmat liittävät soketin yhteydelle. Kommunikoitaessa sokettiin kirjoitetaan tai sitä luetaan. (Lesson: All About Sockets 2014.)

Hassinen (2014) määrittelee soketin seuraavasti:

Soketin avulla voidaan lukea ja kirjoittaa tiettyyn määriteltyyn verkkoyhteyteen. Tämän verkkoyhteyden kuvaa ja yksilöi käytettävä IP-osoite ja portti. Soketti kommunikaatio vaatii aina vastinparin, joka käyttää samaa porttia ja sijaitsee valitun ip-osoitteen spesifoimassa kohteessa. (Hassinen 2014.)

4.3 Välityspalvelin

Välityspalvelin (proxy) on tietokone, joka toimii yhdyskäytävänä kahden verkon välissä. Välityspalvelin pystyy myös pitämään välimuistissaan sivustoja, joilla on käyty, jolloin samoille sivustoille mennessä ei koko sivua tarvitse ladata uudestaan sen alkuperäisestä lähteestä. Välityspalvelimilla pystytään lisäämään turvallisuutta. Verkosta toiseen kommunikoitaessa välityspalvelin ottaa dataa sisään yhdestä portista ja uudelleenohjaa sen ulos toisesta portista. Välityspalvelin estää suoran yhteyden ulkoverkosta sisäverkkoon vaikeuttaen näin hakkerien pääsyä sisäverkon laitteisiin. (What is a proxy server? n.d.)

Käänteinen välityspalvelin (reverse proxy) näkyy käyttäjälle kuin normaali web-palvelin. Asiakas lähettää pyynnöt käänteiselle välityspalvelimelle, joka sitten päättää mihin pyynnöt lähetetään. Se palauttaa vastauksen sisällön aivan kuin se olisi itse niiden lähde. (Apache Module mod_proxy 2014.)

4.4 DNS

Domain Name System eli nimipalvelu on palvelu, joka muuttaa verkkotunnukset IP-osoitteiksi ja IP-osoitteet verkkotunnuksiksi. Nimipalvelujärjestelmä koostuu nimipalvelimista, jotka ovat verkkotunnustietoja sisältäviä palvelinohjelmia. Palvelinohjelma toimii usein vain siihen tarkoitukseen varatussa tietokoneessa eli nimipalvelimessa. Verkkotunnuksen vikatilanteessa toiminnan varmistamiseksi jokainen verkkotunnus tulisi määritellä vähintään kahteen toisistaan riippumattomaan nimipalvelimeen. Nimipalvelimet jaetaan resolvoivaan ja autoratiiviseen nimipalvelimeen. Resolvoivalta nimipalvelimelta voi esimerkiksi organisaation laitteet kysyä nimipalvelukysymyksiä, sillä se tuntee DNS-puun rakenteen ja osaa selvittää vastaukset kysymyksiin. Autoratiivinen nimipalvelin sisältää vähintäänkin yhden verkkotunnuksen tiedot ja siitä resolvoivat palvelimet voivat kysyä siihen vyöhykkeeseen liittyviä tietoja. (Nimipalvelimet 2014.)

5 Kuormantasaus

5.1 Yleistä

Palvelimella on rajoitetut resurssit, ja koska esimerkiksi nettisivustolle voi tulla käyttäjiä enemmän kuin mitä yksi palvelin pystyy käsittelemään, täytyy kyseisen sivuston pyöriä useammalla palvelimella. Tätä sanotaan skaalautumiseksi. Skaalautuminen ei ole ongelma intranetissä, koska käyttäjien määrä harvoin nousee ja se on tiedossa. Nopeiden yhteyksien levinneisyyden ja suurien käyttäjämäärien myötä tämä kuitenkin on ongelma internetissä. Sivuston ylläpitäjän täytyy löytää tapoja levittää kuormaa useammalle palvelimelle, joko sovelluspalvelimessa olevan sisäisen mekanismin, ulkoisten komponenttien tai arkkitehtuurin uudelleensuunnittelun avulla. (Tarreau 2014a.)

Kuormantasauksella työtaakka hajautetaan tasaisesti kahdelle tai useammalle koneelle, verkkolinkille, prosessorille, kiintolevyille tai muulle laitteelle parhaan resurssien käytön, maksimaalisen suorituskyvyn, minimaalisen vasteajan saavuttamiseksi sekä ylikuormituksen välttämiseksi. Redundanttisuus lisääntyy käytettäessä samaan tehtävään useampaa komponenttia, koska silloin yhden komponentin hajoaminen ei välttämättä haittaa. (Mcheick, Mohammed & Lakiss 2011, 104.)

5.2 Staattinen kuormantasa

Staattisessa kuormantasauksessa käytetään kuormantasausalgoritmeja, jotka hajauttavat liikenteen sisään tulevalle liikenteelle ennalta asetettujen sääntöjen mukaan (What is the difference between Static and Dynamic Load Balancing? n.d).

5.3 Dynaaminen kuormantasa

Dynaaminen kuormantasa monitoroi aktiivisesti kuormitustasoja ulosmenokanavilla ja näiden reaaliaikaisten tietojen avulla pyrkii hajauttamaan kuorman järkevästi. (What is the difference between Static and Dynamic Load Balancing? n.d.)

5.4 Kuormantasaaja (Load Balancer)

5.4.1 Yleistä

Kuormantasaaja on laite, joka toimii käänteisenä välityspalvelimena ja hajauttaa verkon tai ohjelmiston liikennettä useammalle palvelimelle. Kuormantasaajia käytetään nostamaan kapasiteettia (samanaikaisia käyttäjiä) ja ohjelmistojen luotettavuutta. Ne parantavat ohjelmistojen yleistä toimintaa vähentämällä

ohjelmiston ja verkon sessioiden hallintaan ja ylläpitoon liitettyjen palvelimien taakkaa. (Load Balancer n.d)

Kuormantasaajat jaetaan kahteen kategoriaan: OSI-kerroksen 4 ja OSI-kerroksen 7 kuormantasaajiin. Neljännen OSI-kerroksen kuormantasaajat toimivat verkko- ja siirtokerroksilla olevien protokollien (IP, TCP, FTP, UDP) datan perusteella. Seitsemännen OSI-kerroksen kuormantasaajat hajauttavat pyynnöt ohjelmistokerroksen protokollien kuten HTTP:n datan perusteella. (Load Balancer n.d.)

Kuormantasaaja auttaa lisäämään palvelun saatavuutta vähentämällä yksittäisen virheen mahdollisuutta ajaa palvelu alas. Se jakaa kuormaa useammalle samaa palvelua tarjoavalle koneelle, jolloin yhden koneen hajotessa liikenne voidaan ohjata toimiville koneille ja palvelu pysyy näin ollen saatavilla. (Understanding Load Balancing n.d.)

5.4.2 HAProxy

HAProxy on ilmainen, todella nopea ja luotettava ratkaisu, joka tarjoaa korkeaa saatavuutta, kuormantasausta ja välityspalvelua TCP- ja HTTP-pohjaisille sovelluksille. Se soveltuu erityisesti web-sivustoille, joihin kohdistuu raskasta kuormaa ja tarvitaan pysyvyyttä tai Layer7-prosessointia. Kymmenien tuhansien yhteyksien tukeminen on selvästi realistista nykyaikaisilla laitteistoilla. HAProxy pystyy seuramaan laitteiden tilaa tila-tarkistusten avulla.

Tuetut alustat:

- Linux 2.4
- Linux 2.6
- Solaris 8/9
- Solaris 10
- FreeBSD 4.10 – 8
- OpenBSD 3.1 -> nykyiseen

(HAProxy 2014.)

Keskittason laitteistolla, joissa on RAM-muistia (Random Access Memory) 1GB:n verran, HAProxy pystyy hoitamaan oikein säädetyillä asetuksilla 40000–50000 yhtäaikaista yhteyttä (Davis 2012).

HAProxy on erittäin tietoturvallinen, siitä on löydetty vain yksi haavoittuvuus vuonna 2002 ja sekin korjattiin alle viikossa. Tietoturvallisuus itse välityspalvelimelle on tärkeää, koska vaikka se suojaakin muita laitteita olemalla käyttäjien ja oikeiden laitteiden välissä, välityspalvelin itsessään on paljastettuna hyökkäyksille. HAProxyn versiot 1.5-dev12 eteenpäin tukevat SSL-protokollaa. (HAProxy 2014.)

HAProxy käyttää perusasetuksella tila-tarkistukseen TCP-yhteyden muodostamisyritystä. Jos se asetetaan tekemään http-tarkastuksia, niin silloin se TCP-yhteyden muodostuttua lähettää palvelimelle HTTP-pyyntöjä, 2xx ja 3xx alkuiset vastaukset hyväksytään, muutoin palvelimen todetaan olevan alhaalla. (Tarreau 2011.)

5.4.3 Apache

Apachen saa toimimaan kuormantasaajana, mutta se vaatii moduuleja `mod_proxy` ja `mod_proxy_balancer`. Apache tukee HTTP, FTP ja AJP13 protokollien kuormantasausta. Tuettuja kuormanjakamisen algoritmeja on kolme, pyyntöjen määrä, painotettu liikenteen määrä ja odottavien pyyntöjen määrä. Apachen versiot 2.1 ja siitä eteenpäin tukevat kuormanjakamista. (Apache Module `mod_proxy_balancer` 2014.)

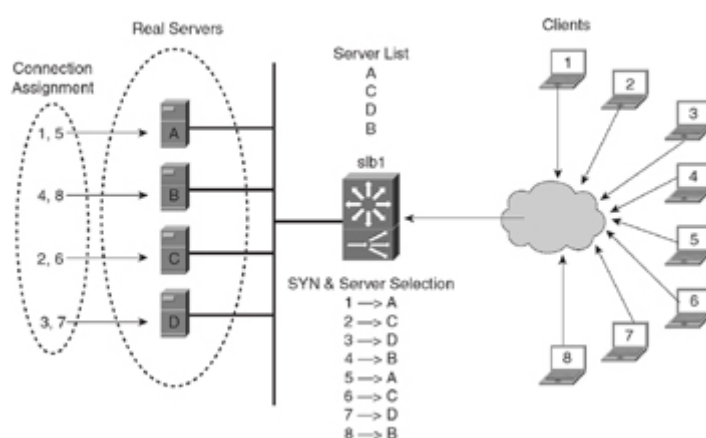
5.4.4 Kuormantasaus algoritmit

Kuormantasaajat käyttävät erilaisia menetelmiä liikenteen tasoittamiseksi.

Round Robinissa liikenne jaetaan vuorotellen koneille joille liikenne on mää-

ritetty jaettavaksi. Heikkoutena tässä menetelmässä on, ettei siinä huomioida kuormitusten määrää eri koneilla, vaan liikenne jaetaan tilanteesta riippumatta vuorotellen. (Understanding Load Balancing 2014.)

Kuviossa 3 on esitetty Round Robin -menetelmän toimiminen. Tässä menetelmässä ainoastaan koneiden saatavuus, eli onko kone päällä vai ei, vaikuttaa päätöksiin.



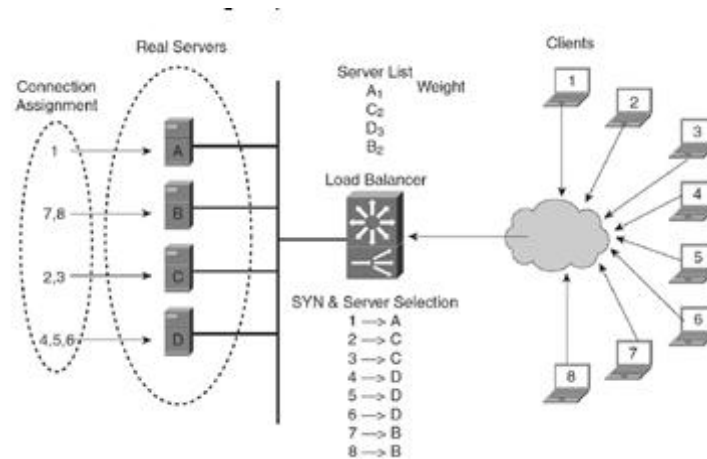
Kuvio 3. Round Robin –menetelmä (Arregoces & Portolani 2003, kappale 16.)

Weighted Round Robin, eli painotettu Round Robin, toimii kuin Round Robin, mutta koneille voidaan asettaa painoarvoja, jonka mukaan liikenne jaetaan (Algorithms 2014).

Painotettu Round Robin on hyödyllinen tilanteissa, joissa esimerkiksi jokin kone on kaksi kertaa tehokkaampi kuin toinen kone. (Intro to Load Balancing for Developers – The Algorithms 2009.)

Kuviossa 4 on esitetty painotetun Round Robinin toimintaa. Palvelin A saa yhden pyynnön, jonka jälkeen palvelin C saa kaksi seuraavaa pyyntöä. Seuraavat kolme pyyntöä ohjataan puolestaan palvelimelle D, jonka jälkeen kaksi

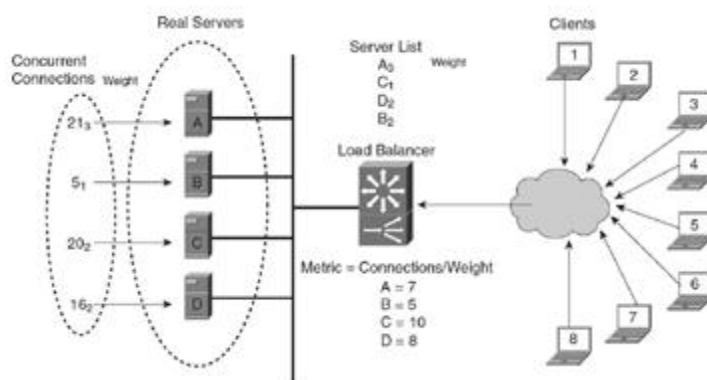
seuraavaa pyyntöä ohjataan palvelimelle B. Tämän jälkeiset pyynnot jaetaan samalla tavalla.



Kuvio 4. Painotettu Round Robin –menetelmä (Arregoces & Portolani 2003, kappale 16.)

Least Connection eli suomeksi ”vähiten yhteyksiä”-menetelmässä pyynnot ohjataan koneelle jolla on vähiten yhteyksiä (Algorithms 2014).

Weighted Least Connection eli ”painotettu vähiten yhteyksiä”-menetelmä, toimii muuten samalla tavalla kuin Least Connection, mutta koneille annetaan painoarvot. Yhteys ohjataan koneelle jolla on pienin metric-arvo, joka on yhteyksien määrä jaettuna painoarvolla. Kuviossa 5 seuraava pyyntö päättyisi palvelimelle B, koska sillä on pienin metric-arvo. (Arregoces & Portolani 2003, kappale 16.)



Kuvio 5. Painotettu vähiten yhteyksiä –menetelmä (Arregoces & Portolani 2003, kappale 16.)

Random: Liikenne ohjataan sattumanvaraisesti koneille (Algorithms 2014).

Dynamic Round Robin: Toimii kuin painotettu Round Robin, mutta painoarvo koneille määritetään jatkuvalla koneiden monitoroinnilla, joten arvot muuttuvat usein. Tämän avulla voidaan esimerkiksi jakaa kuormaa koko ajan koneelle jolla on paras vasteaika. (MacVittie, D 2009.)

Source: Samasta IP-osoitteesta tuleva asiakas ohjataan samalle koneelle (Anicas 2014).

First: HAProxy:n käyttämä menetelmä, jossa yhteydet ohjataan ensimmäiselle farmissa saatavilla olevalle koneelle, jolla on pienin ID. ID-arvot ovat käsin-määriteltäviä. Samalle koneelle ohjataan yhteyksiä, kunnes saavutetaan asetettu maksimi yhteyksien määrä, joka voidaan asettaa konekohtaisesti. Tämän jälkeen liikenne ohjataan seuraavaksi pienimmälle ID:lle. Tämän menetelmän tarkoitus on pystyä käyttämään mahdollisimman vähän koneita, vain niiden tarpeen mukaan, jolloin ylimääräiset koneet voidaan sammuttaa kun niille ei ole tarvetta. Parhaan hyödyn saamiseksi tästä menetelmästä tulisi pilvessä olevaa ohjainta käyttää tarkastamaan mahdollisimman usein koneiden käyttöä ja sulkemaan niitä, mikäli ne eivät ole käytössä. Sama ohjain voisi seurata

backend-palvelinten jonoa nähdäkseen milloin on tarvetta käynnistää lisää koneita. HAProxyn "http-check send-state"-asetusten avulla voidaan myös lähettää tietoa palvelinten kuormituksesta. (Tarreau 2014b.)

5.4.5 Tila-tarkastukset

Kuormantasaajat tekevät tietyin väliajoin tila-tarkastuksia (health checks) palvelimille varmistaa, että ne ovat päällä ja kunnossa. Palvelimen ollessa alhaalla kuormantasaajan tulisi huomata tämä ja lopettaa liikenteen välittäminen alhaalla olevalle palvelimelle. Siihen, kuinka nopeasti kuormantasaaja reagoi tilanteeseen, vaikuttaa kuinka paljon viivettä on tarkastusten välissä, kuinka monta kertaa tarkastus tehdään ennen päätöksen tekoa sekä kuinka kauan odotetaan vastausta ennen luovuttamista. Aikojen asettaminen todella alhaisiksi voi auttaa havaitsemaan virheet nopeasti, mutta pahimmassa tapauksessa se rasittaa palvelua paljon. (Health Checking On Load Balancers: More Art Than Science 2012.)

6 Ohjelmistotestaus

6.1 Yleistä

Ohjelmistotestauksen avulla pyritään parantamaan ohjelman laatua, havaitsemalla siitä virheitä, parantamalla suorituskkyä ja varmistamaan ohjelman toimiminen oikein erilaisissa tilanteissa. Testauksen avulla pyritään myös määrittämään raja-arvot joissa ohjelma toimii oikein. Ohjelmistotestaus jaetaan staattisiin ja dynaamisiin analyyseihin. (Kaitto 1996.)

6.2 Staattinen analyysi

Staattisessa analyysissä etsitään virheitä ja puutteellisuuksia ohjelmasta ajamatta itse ohjelmaa. Etsintä voidaan suorittaa manuaalisesti tutkimalla koodia tai automaattisesti esimerkiksi kääntäjän avulla. Staattisen analyysin avulla voidaan havaita esimerkiksi muuttujien väärinkäyttöä, parametrien väärinkäyttöä sekä funktioiden väärinkäyttöä. (Kaitto 1996.)

6.3 Dynaaminen analyysi

Dynaamisessa analyysissä seurataan ohjelman ja koodin käyttäytymistä ohjelman suorituksen aikana. Testauksen edellytyksenä on toimiva ohjelma tai ympäristö, jossa funktioita voidaan suorittaa. Dynaamisen testauksen avulla voidaan testata asioita, jotka olisivat joko mahdottomia tai liian vaikeita testata staattisella analyysillä. (Kaitto 1996.)

6.4 Testaustavat

Testaustapoja on olemassa monia, joita käytetään tapauksesta riippuen ja sen perusteella mitä testeillä halutaan selvittää. Testaustavat eivät ole toisiaan poissulkevia. Tässä työssä käytetään suorituskyvyn testausta, jonka avulla voidaan testata kuinka palvelu toimii tietyn kuorman alla sekä onko se skaalautuva ja luotettava. Suorituskyvyn testaamisen avulla voidaan tunnistaa pulonkaulat ja mahdolliset riskit. (Ohjelmiston testaus. n.d.)

Tässä työssä käytetään suorituskyvyn testaamisen lisäksi regressiotestausta, jossa uudelleen käytetään aikaisempia testejä muutoksien jälkeen, jotta varmistutaan siitä, etteivät muutokset rikkoneet tuotteita (Kaitto 1996).

6.5 Ohjelmistotestaus työkalut

6.5.1 Apache Jmeter

JMeter on vapaan lähdekoodin työpöytäsovellus. Se on 100 % Java-sovellus, joka on suunniteltu kuormantasauksen toiminnalliseen käyttäytymisen ja suorituksen mittaamiseen. Alkuunsa se suunniteltiin testaamaan web-sovelluksia, mutta sitä on jälkeinpäin laajennettu muihin testitoimintoihin. (Apache Jmeter 2014.)

6.5.2 LoadUI ja SoapUI

SoapUI on ilmainen avoimen lähdekoodin järjestelmäriippumaton toiminnan testaus-ratkaisu. SoapUI:ssa on helposti käytettävä graafinen käyttöliittymä, joten sillä on helppoa ja nopeaa luoda testejä. Se tukee kaikkia standardeja protokollia ja teknologioita. SoapUI:sta on saatavilla myös maksullinen Pro versio, joka sisältää vielä enemmän toimintoja ja se onkin suunnattu enemmän yrityksen tarpeisiin. (What is SoapUI? 2014.)

LoadUI on ilmainen kuormituksen testausohjelmisto. Sen ”vedä-ja-pudota”-käyttöliittymän ansiosta pystytään luomaan, säätämään ja jakamaan uudelleen reaaliajassa omia kuormantestauksia. Kuten SoapUI sekin tukee kaikkia standardeja, protokollia ja teknologioita. (API Load Testing Made Easy 2014.)

6.5.3 Locust

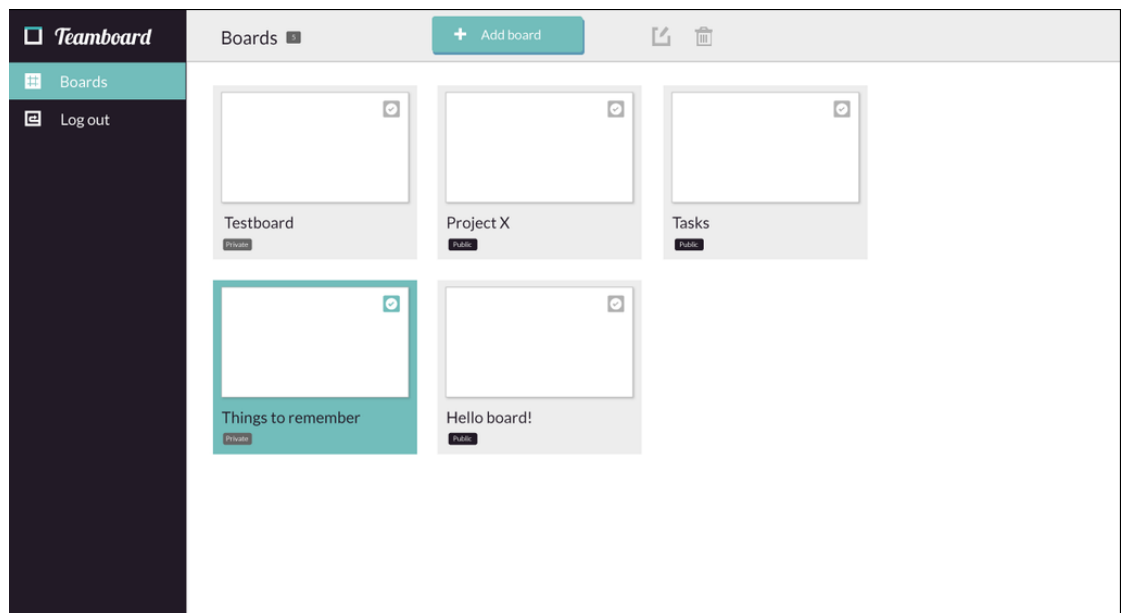
Locust on käyttäjän aiheuttaman kuorman testaustyökalu. Se on tehty web-sivustojen kuormantestaukseen sekä todentamaan kuinka monta yhtäaikaista käyttäjää järjestelmä pystyy käsittelemään. Käyttäjien luomistahti ja toimintamallit voidaan määrittää etukäteen, ja testiä voidaan reaaliajassa tarkkailla web-käyttöliittymästä. Jokainen käyttäjä on itsenäinen prosessi, jonka ansios-

ta voidaan luoda monimutkaisia tilanteita käyttäen Python-ohjelmointikieltä. Locust pystytään jakamaan toimimaan useammalla laitteella, joka mahdollistaa tuhansien käyttäjien luonnin. (Heyman, Byström, Hamrén & Heyman 2014.)

7 Teamboard

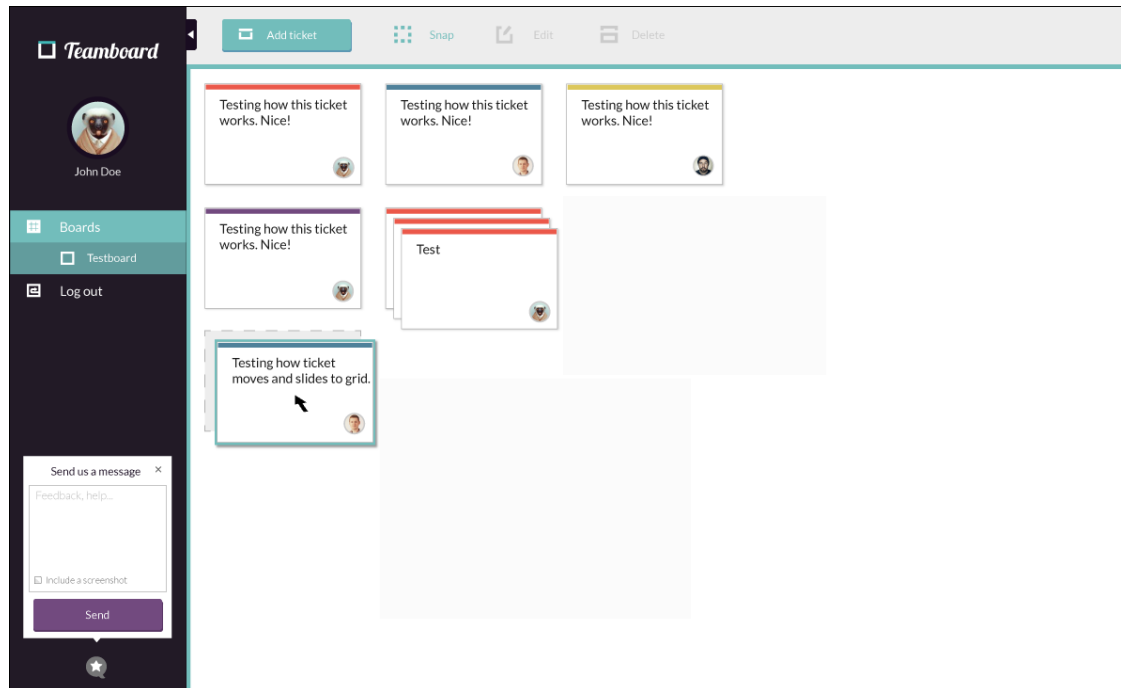
7.1 Yleistä

Teamboard on palvelu, jossa käyttäjät voivat reaaliaikaisesti luoda tauluja. Tauluissa käyttäjä voi sitten lisätä, poistaa ja järjestellä lappuja. Reaaliaikaisuudesta huolehtii Socket.IO. Palvelun on tarkoitus olla tietoturvallinen, siksi tunnistautumisessa käytetään Bcrypt ohjelmistoa. Tietokannan skaalautuvuuden takaamiseksi käytetään MongoDB tietokantaohjelmaa. Kuviossa 6 näkyy Teambordin perusnäkö. Kuviossa on luotuna 5 taulua, joihin käyttäjä voi mennä sisään.



Kuvio 6. Teambordin workspace -näkö

Kuviossa 7 näkyy Teamboardin taulu-näkymä, johon on luotu useampia lappuja. Käyttäjät voivat luoda, poistaa ja liikuttaa lappuja.



Kuvio 7. Teamboardin taulu -näkö

7.2 Komponentit

7.2.1 Socket .IO

Socket.IO mahdollistaa kaksisuuntaisen reaaliaikaisen tapahtumapohjaisen kommunikaation. Socket.IO toimii kaikilla alustoilla, selaimilla tai laitteilla. Se keskittyy nopeuteen sekä luotettavuuteen tasa-arvoisesti. (Socket.IO 2014.)

7.2.2 Redis

Redis on open source ja BSD lisensoitu kehittynyt avain-arvojen varasto. Sitä käytetään yleensä tietorakenne-palvelimena, koska avaimet voivat sisältää listoja, hasheja, merkkijonoja tai muuta sellaista. Redis toimii useimpien tunnettujen ohjelmointikielien kanssa. Sen suorituskky perustuu siihen, että se toimii muistissa olevilla tiedostoilla. Käyttötarkoituksesta riippuen tiedostot voidaan tallentaa levyille välillä tai lisätä kaikki komennot lokiin. (Introduction to Redis n.d.)

7.2.3 MongoDB

MongoDB on tietokantaohjelma, joka tarjoaa korkeaa suoritusta ja saatavuutta sekä helppoa skaalautuvuutta. Tietokannan kapasiteettiä voidaan kasvattaa tietokannan ollessa toiminnassa ilman alhaalla olo aikaa. MongoDB käyttää dynaamista mallia tietokannassa, joka mahdollistaa sen, että tietueilla ei tarvitse olla samaa rakennetta ja ne voivat sisältää keskenään erityyppistä tietoa. (Introduction to MongoDB 2014.)

7.2.4 Bcrypt

Bcrypt on ohjelma, joka käyttää Blowfish salausalgoritmia. Bcryptin vahvuus perustuu sen hitauteen, koska sille voidaan määrittää kuinka työläs tiiviste-funktio on. Koneiden tehojen kasvaessa voidaan lisätä työmäärää, jolloin tiivistämisestä tulee hitaampaa. Esimerkiksi MD5 tiivistää salasanan "yaaa" alle mikrosekunnissa, työ kertoimella 12 asetettu Bcrypt tiivistää samaa salasanaa 0.3 sekuntia. (Hale 2010.)

8 Toteutus

8.1 Vaatimukset ja lähtökohdat

Teamboard käyttää TCP ja HTTP protokollia, joten käytettävän kuormantasaajan tulisi osata käsitellä näitä protokollia eli siis toimia OSI-mallin neljännellä kerroksella. Jokainen luotu kone maksaa DigitalOceanissa, joten siellä ei ole järkevää toteuttaa kuormantasausta, jossa koneet ovat poissa päältä kun niitä ei tarvita. DigitalOceanin pilvessä virtuaalikoneen resurssien lisäämiseksi kone on sammutettava hetkeksi. Tämä on ongelma, koska silloin kuormantasaaja olisi hetken poissa päältä. Tästä syystä kuormantasaajan olisi hyvä pystyä skaalautumaan horisontaalisesti, eli tarvittaessa hyödyntämään usean koneen resursseja ilman uudelleen käynnistämistä. Kuormantasaajan avulla Teamboardin tulisi kestää mahdollisimman monta yhtäaikaista käyttäjää. Tässä vaiheessa Teamboard on asennettuna yhdelle virtuaalikoneelle DigitalOceanin tarjoamaan pilvipalveluun. Palveluun on ohjattu web-osoite teamboard.n4sjamk.org, joka osoittaa suoraan Teamboard-virtuaalikoneen IP-osoitteeseen. Ohjelmaa kehitetään jatkuvasti. Ympäristössä käytetään Ubuntu-virtuaalikoneita versiolta 12.04 ja 14.04.

8.2 Testausohjelmistojen asennus

8.2.1 HAProxy

HAProxy asennetaan käskyllä:

```
sudo apt-get install haproxy
```

Seuraavaksi asetetaan HAProxy käynnistymään init-scriptillä, muokkaamalla tiedostoa `/etc/default/haproxy`, kyseiseen tiedostoon asetetaan `ENABLED=1`. Tämän jälkeen muokataan tiedostoa `etc/haproxy/haproxy.cfg`.

Kyseisen haproxy.cfg tiedoston sisältö on Liite1:stä. Kun tiedostoa on muokattu, HAProxy käynnistetään käskyllä:

```
sudo service haproxy start
```

Nimipalvelimella muutetaan Teamboardin web-osoite, joka aiemmin osoitti Teamboard-virtuaalikoneelle, osoittamaan nyt HAProxy koneeseen. Alussa käytetään kuormantasausta menetelmänä Round Robinia, koska se on HAProxyn käyttämistä menetelmistä sopivin, sillä kuorman halutaan jakautuvan mahdollisimman tasaisesti. Round Robin mahdollistaa eri koneiden painoarvojen muuttamisen dynaamisesti. WebSocket yhteys on pysyvä, joten Socket-koneiden kuormantasausta täytyy toteuttaa niin, että samalta asiakkaalta tuleva yhteys päättyy aina samalla koneelle. Tähän käytetään HAProxyn kuormantasausta metodia nimeltä source.

8.2.2 Apache

Apache asennetaan koneelle käskyllä:

```
sudo apt-get install apache2
```

Apacheen täytyy asentaa välityspalvelimen tarvitsemat mod-lisäosat, tämä tehtiin käskyillä:

```
a2enmod proxy proxy_http  
a2enmod proxy proxy_balancer
```

Apachen asetuksia täytyy säätää sopiviksi, syöttää oikeat IP-osoitteet, sekä säätää kuormantasauksen asetukset, tämä tapahtuu muokkaamalla default-tiedostoa, joka tehtiin käskyllä:

```
nano /etc/apache2/sites-available/default
```

Kyseisen default-tiedoston sisältö on Liite 2:sta. Tiedoston muokkaamisen jälkeen Apache täytyy vielä uudelleen käynnistää käskyllä:

```
sudo service apache2 restart
```

8.2.3 SoapUI & LoadUI

Web-sivulta <http://www.soapui.org/> valitaan "Download SoapUI" ja versioista valitaan oikea käyttöjärjestelmän perusteella jonka jälkeen tiedosto ladataan. Seuraavaksi kyseisen tiedoston oikeuksia täytyy muuttaa, jotta se voidaan suorittaa. Tämä tapahtuu menemällä polkuun johon tiedosto ladattiin ja antamalla käsky:

```
chmod +x SoapUI-x64-5.0.0.sh
```

Ohjelma suoritetaan tämän jälkeen käskyllä:

```
./SoapUI-x64-5.0.0.sh
```

Käskyn jälkeen ohjelman asennusvelho käynnistyy ja tarjoaa mahdollisuutta ladata ja asentaa LoadUI.

8.2.4 JMeter

JMeter asennetaan käskyllä:

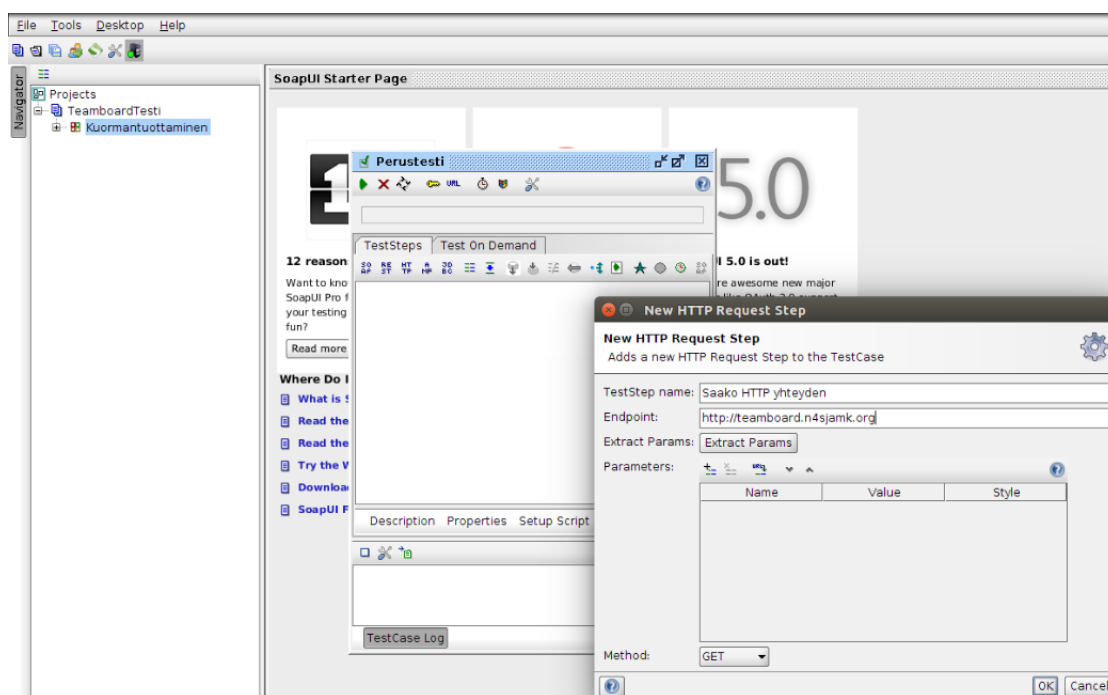
```
sudo apt-get install jmeter
```

Hetken kuluttua JMeter on valmiina käytettäväksi.

8.3 Testien luonti

8.3.1 SoapUI

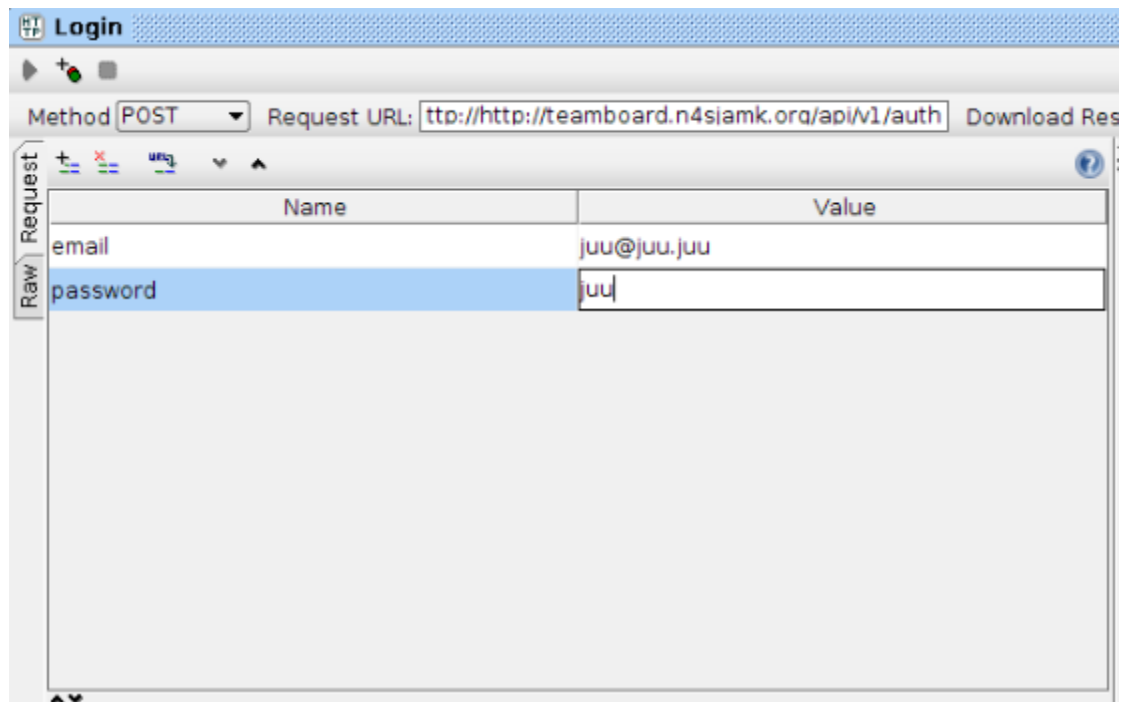
SoapUI:ssa painetaan hiiren oikealla Projects kohtaa ja valitaan ”New Project”, jonka jälkeen uudelle projektille määritellään nimi. Tämän jälkeen luodaan testiin vaiheita painamalla juuri luotua projektia ja valitaan ”Add step”. Vaiheelle määritellään nimi ja osoite, johon vaihe suoritetaan. Kuviossa 8 näkyy kuinka luodaan ”Perustesti”, johon luodaan vaihe jolla testataan HTTP-yhteyden toimivuus.



Kuvio 8. SoapUI HTTP-yhteyden testaaminen

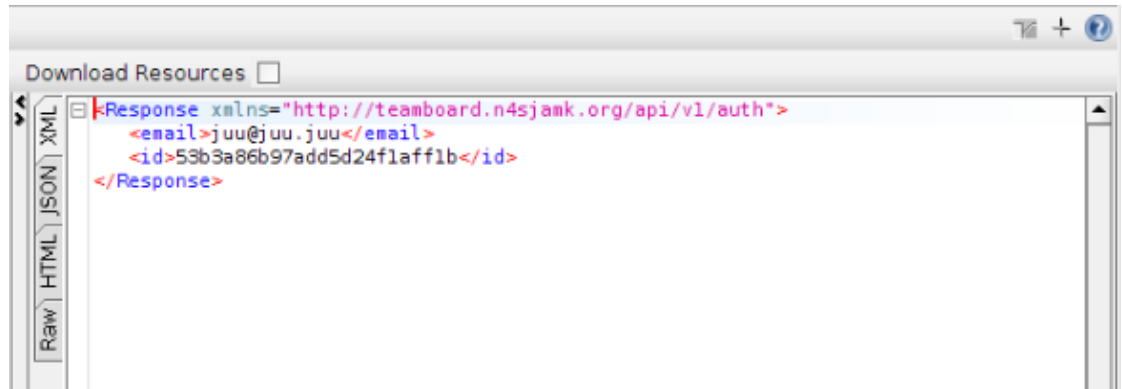
Login-vaihe luodaan kuviossa 9 näkyvällä tavalla. Kun metodina on POST, voidaan määrittää mitä tietoja pyynnössä halutaan lähettää. Polkuun täytyy tässä tapauksessa lisätä /api/v1/auth, koska sisäänkirjautuminen tapahtuu kyseiseen osoitteeseen. Tässä vaiheessa luodaan kentät email ja password, koska API vaatii sisäänkirjautumisen yhteydessä kyseiset tiedot. Arvoiksi ase-

tetaan millä tunnuksella ja salasanalla halutaan testata kirjautua sisään palveluun. Jotta kyseinen testi toimisi, täytyy nämä tunnukset olla rekisteröitynä palveluun.



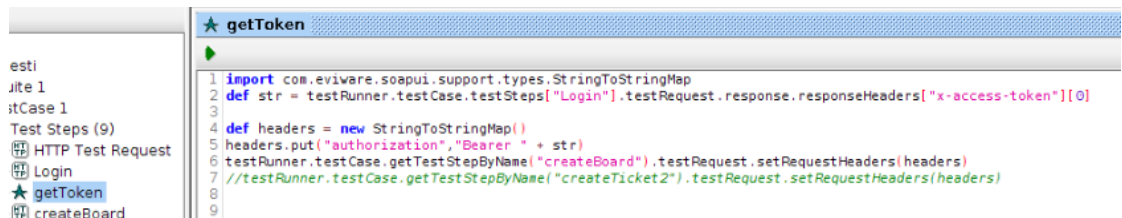
Kuvio 9. Login-vaiheen luonti

Vaihetta voidaan testata painamalla vihreää nuolta vasemmassa yläkulmassa. Testin vastaus tulee oikealla näkyvään laatikkoon. Kuviosta 10 voidaan nähdä mitä palvelin vastaa.



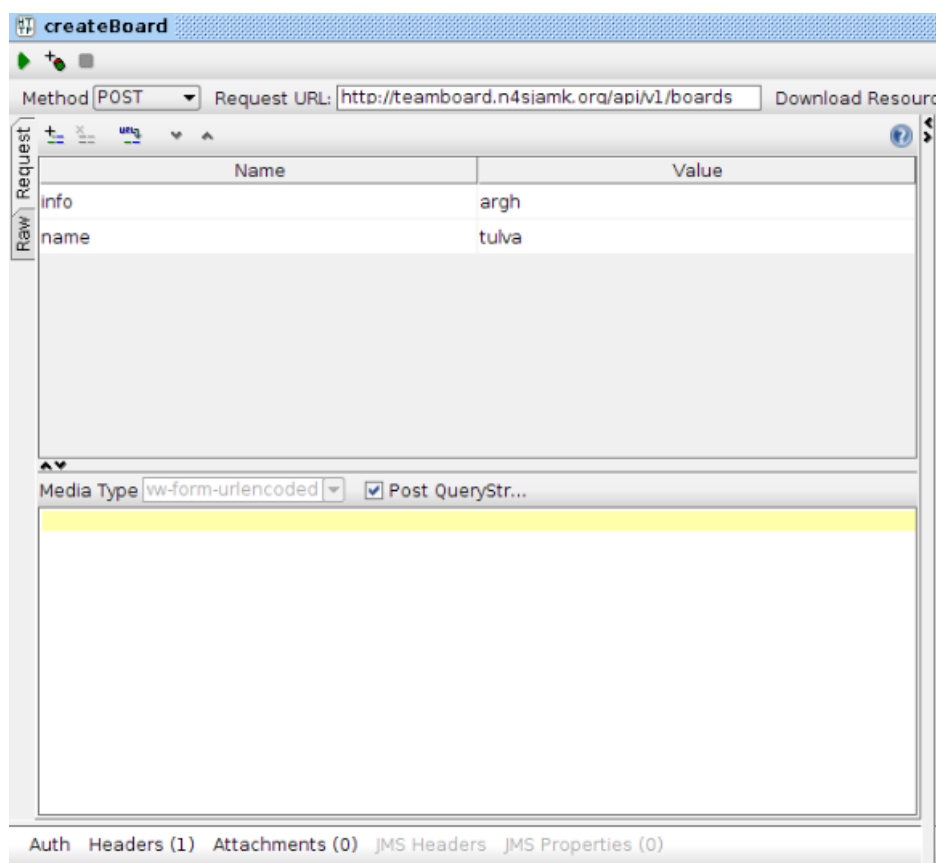
Kuvio 10. Login-vaiheen testaaminen

Palvelimen vastauksen otsikkokenttä sisältää poletin kohdassa "x-access-token". Tämä muuttuja on otettava talteen ja sisällytettävä myöhemmissä palvelimelle lähetettävissä pyynnöissä, koska sillä tunnistaudutaan palvelimelle. Muuttujan talteen saamiseksi otsikkokentästä täytyy luoda groove-komentosarja. Komentosarjan "getToken" sisältö näkyy kuviossa 11.



Kuvio 11. getToken-komentosarja

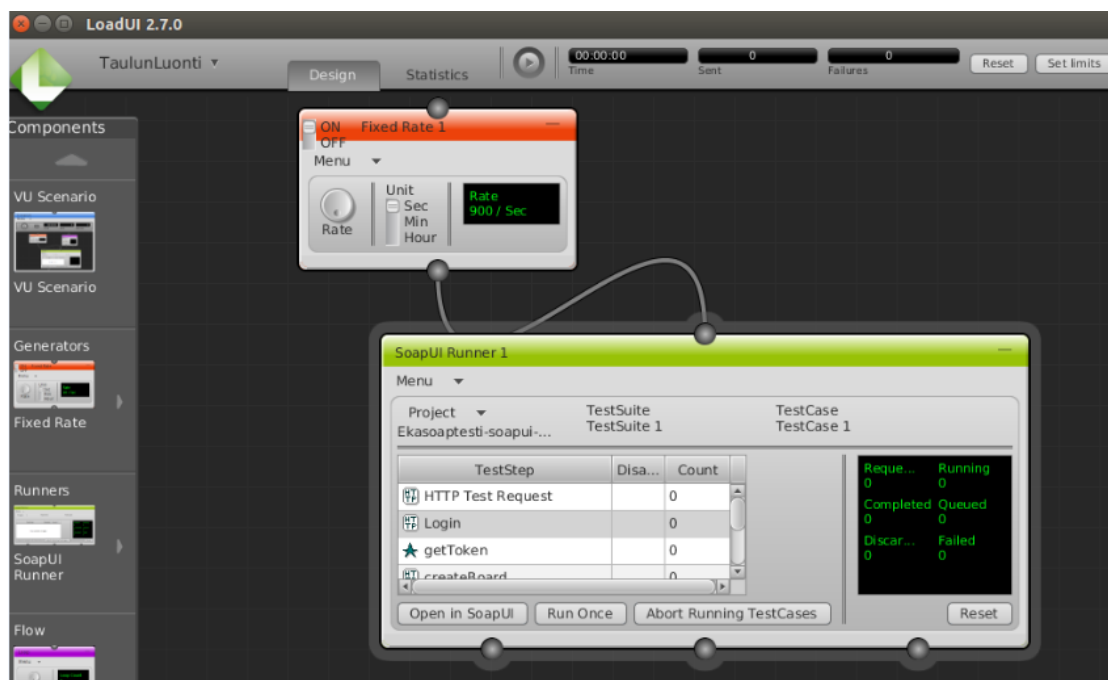
Komentosarjassa poletti otetaan talteen ja lisätään sitten vaiheen "createBoard" otsikkokenttään niin, että otsikkokentän sisältö on "authorization: Bearer +poletinsisältö". Tämä on muoto, jolla API-rajapinta vaatii tunnistautumisen tapahtuvan. Seuraava vaihe testaa taulunluomista. Erona aiempaan POST-pyyntöön, on osoitteen muuttaminen /api/v1/boards, sekä määritellään "info"- ja "name"-muuttujien avulla minkä niminen taulu halutaan luoda ja mitä se sisältää. Kuviossa 12 näkyy kuinka nämä asetukset ovat asetettu.



Kuvio 12. Taulunluonti-vaiheen asetukset

8.3.2 LoadUI

LoadUI ohjelmaan ladataan SoapUI:ssa tehty testi. Kyseinen testi yhdistetään Fixed Rate -komponenttiin, jonka avulla voidaan määrittää kuinka usein kyseinen testi ajetaan. Kuviosta 13 nähdään testissä käytetyt asetukset, mutta siinä ei ole vielä asetettu "Set Limits"-napin takaa kuinka monesti testi suoritetaan. Suoritettavien testien määrä on siis asetettu 900 kappaleeseen testiä varten, vaikkei se käy tästä kuviosta ilmi. Ensimmäisen testin yhteydessä selvisi myös, ettei LoadUI pysty tekemään 900 testiä sekunnissa, vaikka Fixed Rateen niin pystyi laittamaankin. Ohjelma ei valita asiasta ollenkaan, mutta testin kuluessa seuraamalla "sent"-palkissa näkyvää lukua voidaan huomata, ettei ohjelma tee kuin n.15 testiä sekunnissa. Luotettavien ja tasaisten tulosten saamiseksi syötetään Fixed raten arvoksi 10/sec.

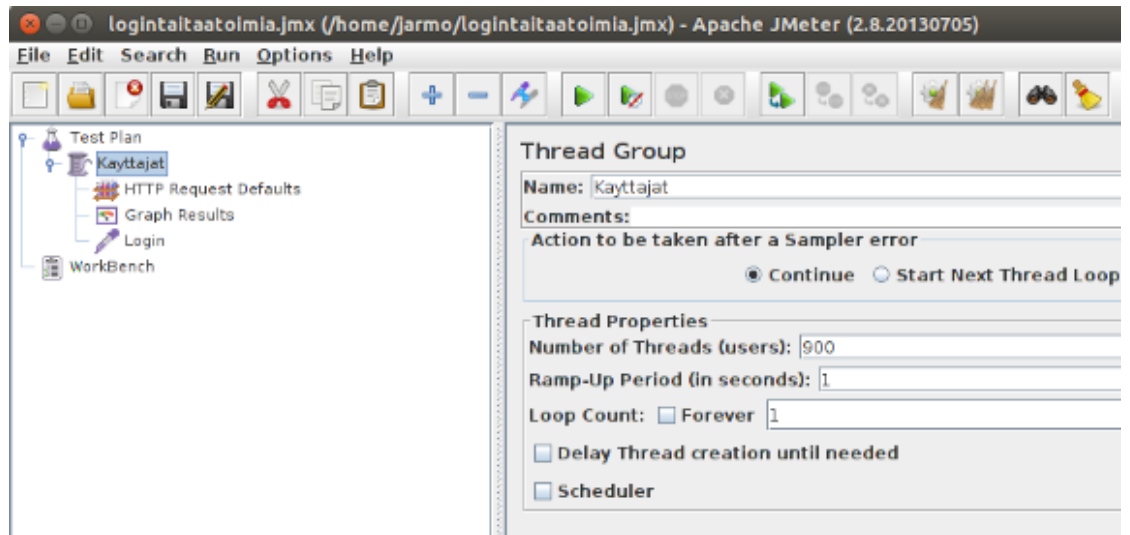


Kuvio 13. LoadUI-testi

Sama testi ajetaan suoraan SoapUI:sta. Tämä tapahtuu valitsemalla SoapUI:stä aiemmin luotu testi hiiren oikealla painikkeella ja valitsemalla Create Load Test. Tällöin kyseisestä testistä tehdään kuormitustesti, johon voidaan määrittää suoraan esimerkiksi käyttäjien määrä sekä viive-asetuksia.

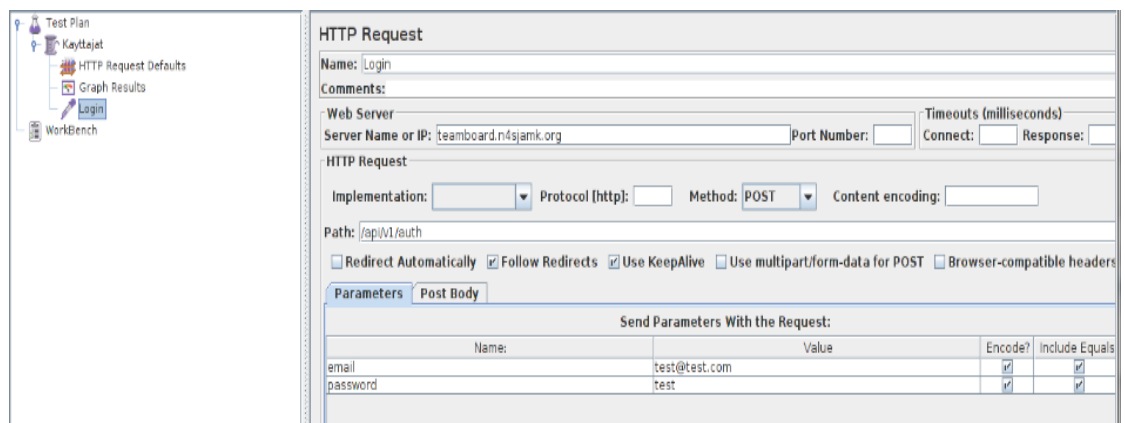
8.3.3 JMeter

Apache JMeter luodaan uusi suunnitelma nimeltä Test Plan. Sitten suunnitelmaan lisätään Thread Group, jossa voidaan määritellä nimen lisäksi käyttäjien määrä ja tahti, jolla käyttäjiä luodaan, sekä monesti silmukka suoritetaan. Kuvion 14 mukaisesti saadaan luotua ryhmä nimeltä käyttäjät, joiden määrä asetetaan olemaan 900.




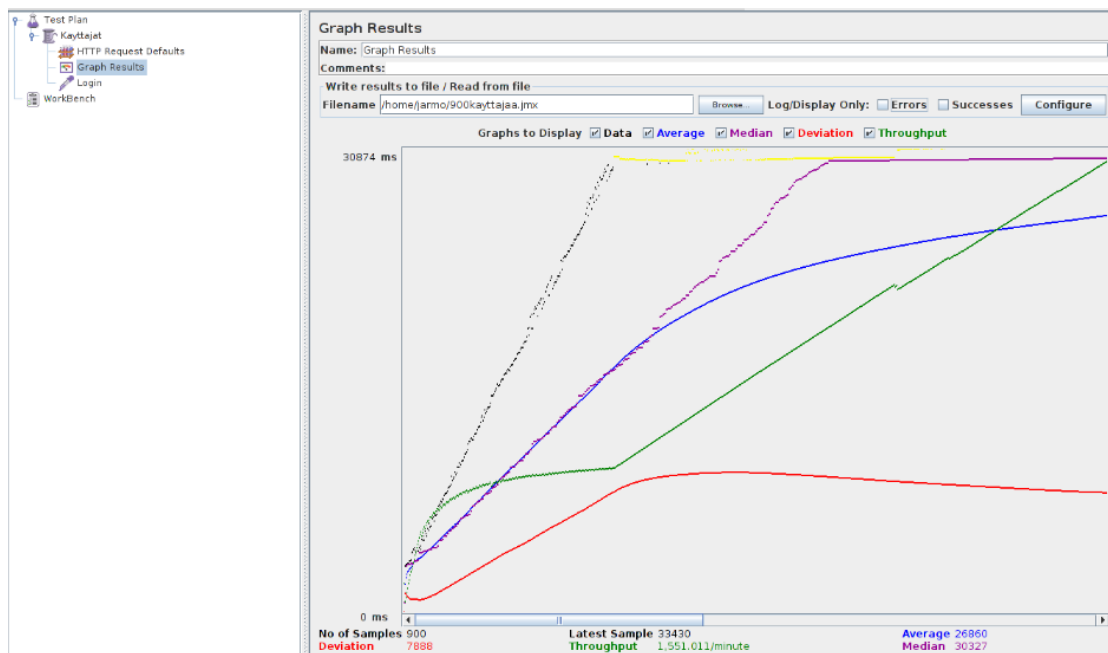
Kuvio 14. Jmeter, käyttäjien luonti ja määrän asettaminen

Tämän jälkeen luodaan HTTP pyyntö, jonka nimeksi asetetaan Login, osoitteeksi teamboard.n4sjamk.org sekä parametri kenttään aiemmin Teamboardille luodut testi tunnukset kuvion 15 mukaisesti.



Kuvio 15. Jmeter, Login-testin luominen

Testi aloitetaan painamalla -kuvaketta, jolloin painamalla Graph Resultsia pystytään seuraamaan testin kulkua kuvion 16 mukaisesta näkymästä reaaliajassa.



Kuvio 16. Jmeter, Login-testin graafinen seuraaminen

8.4 Kuormantasaajien vertailu

8.4.1 Yleistä

Apachesta on yleinen välityspalvelin ohjelmisto, sekä sitä kehitetään jatkuvasti. HAProxy taas tarjoaa enemmän menetelmiä kuormanjakamiseksi, joista ehkä tärkein tulevaisuuden kannalta on First -menetelmä. Tämän menetelmän avulla pystytään pilvipalvelussa, jossa ei makseta koneista silloin kun ne ovat poissa päältä, säästämään rahaa. HAProxyn etuja ovat myös sen turvallisuus sekä erittäin kustomoitavat asetukset.

HAProxy osaa myös näyttää tilastoja sen takana olevista laitteista, mikä miellytti toimeksiantajaa. Kuviossa 17 näkymä HAProxy:n tilastoista.

HAProxy version 1.5-dev25-a339395, released 2014/05/10
Statistics Report for pid 25355

> General process information

pid = 25355 (process 1), nbproc = 1
uptime = 7s 134.446s
current load: min/max = 0.00/0.00
maxconn = 1024, maxqueue = 4096, maxidle = 1
current conn = 22, current proc = 10, idle proc = 10
Running tasks: 1/2, db = 100%

Legend:

- active UP
- active UP, going down
- active DOWN, going up
- active DOWN
- not checked
- active or backup (DOWN for maintenance (MANT))
- active or backup (STOPPED for maintenance)
- Note: "NO,2/1/0/0/0" - UP with last heartbeat checked.

Display options:

- Server
- Stats, SCORE, ADDRESS
- Backend, IP
- CPU, usage

External resources:

- HAProxy, 1.5
- HAProxy, 1.5.1
- HAProxy, 1.5.1

Frontend		Backend		Server		Stats		Errors		Warnings		Status		LastChk		Mgmt		Act		Bck		Chk		Down		Downtime		Totals	
Conn	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Kuvio 17. HAProxy:n tilastot

8.4.2 HAProxy vs Apache-testi

SoapUI:lla kuormaa luodaan 750 käyttäjällä, ja sama testi ajetaan Teamboardille Apachen ja HAProxy:n kautta, jotta nähdään onko näiden kahden välillä eroja. Kuviossa 18 on ylempänä HAProxy:lle tehty testi ja alempana Apachelle tehty testi. Kuviota tutkimalla voidaan havaita HAProxy:n voittavan jokaisessa sarakkeessa, eli se käsittelee tavaraa nopeammin kuin Apache.

kuormitus

Threads: 750 Strategy: Simple Test Delay: 0 Random: 0.5 Limit: 1 Runs per Thread: 100%

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
HTTP Test Request	126	39204	6,304.4	39203	750	16.29	477750	10382	0	0
Login	264	7938	2,794.14	1732	750	16.29	41250	896	0	0
getToken	1	38	0.18	0	750	16.29	0	0	0	0
createBoard	37	6891	84.85	44	750	16.29	44942	976	0	0
setBoardID (disabled)	0	0	0	0	0	0	0	0	0	0
avaaBoard (disabled)	0	0	0	0	0	0	0	0	0	0
createTicket (disabled)	0	0	0	0	0	0	0	0	0	0
createTicket2 (disabled)	0	0	0	0	0	0	0	0	0	0
Nytoimii (disabled)	0	0	0	0	0	0	0	0	0	0
TestCase:	428	54071	9,183.58	40979	750	16.29	563942	12255	0	0

kuormitus

Threads: 750 Strategy: Simple Test Delay: 0 Random: 0.5 Limit: 1 Runs per Thread: 100%

Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
HTTP Test Request	210	29826	11,046.69	19651	750	8.4	477750	5351	0	0
Login	625	20802	7,610.86	6256	750	8.4	41250	462	0	0
getToken	1	11	0.05	0	750	8.4	0	0	0	0
createBoard	37	943	65.53	38	750	8.4	64202	719	0	0
setBoardID (disabled)	0	0	0	0	0	0	0	0	0	0
avaaBoard (disabled)	0	0	0	0	0	0	0	0	0	0
createTicket (disabled)	0	0	0	0	0	0	0	0	0	0
createTicket2 (disabled)	0	0	0	0	0	0	0	0	0	0
Nytoimii (disabled)	0	0	0	0	0	0	0	0	0	0
TestCase:	873	51582	18,723.141	25945	750	8.4	583202	6532	0	0

Kuvio 18. HAProxy vs Apache 750 käyttäjällä

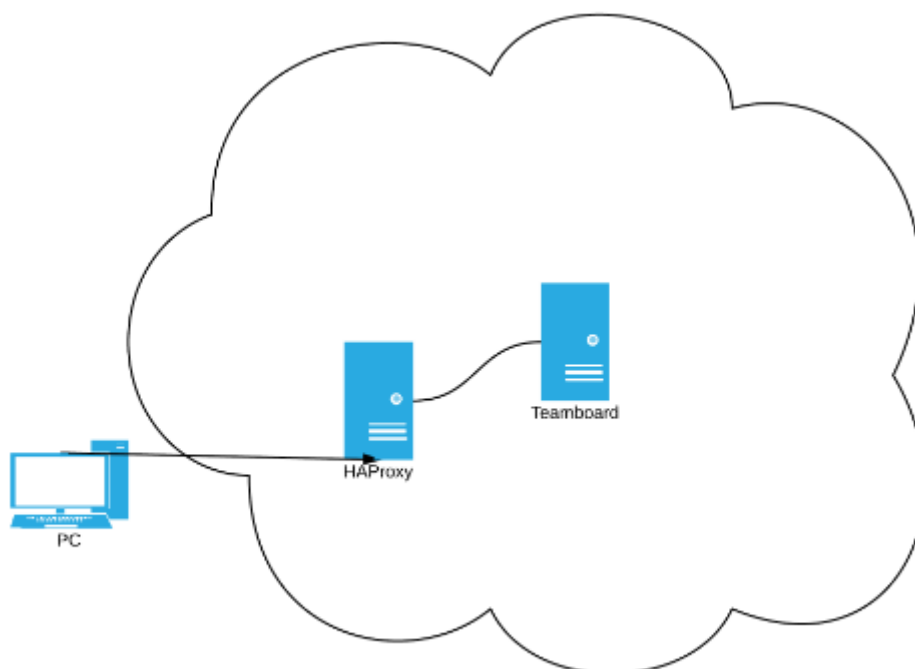
Ohjelmaa testatessa selvisi, että HAProxy käynnistyy ilman minkäänlaista viivettä ja näin ollen se on selvästi nopeampi käynnistymään kuin Apache, jolla saattoi kestää useita sekunteja käynnistystä. Tämä on erittäin tärkeä ominaisuus, sillä kun välityspalvelin on alhaalla, asiakas ei pääse sen takana oleviin palveluihin.

Testien yhteydessä testattiin myös Teamboardin toimintaa selaimella. Selaimella testatessa huomattiin, ettei Apache-välityspalvelin onnistu välittämään Teamboardin tarvitsemia socket-yhteyksiä. Sillä soketit välittävät muutostiedot kaikille käyttäjille Teamboardin taulussa, mutta kun Apachen kautta ohjattiin liikenne, niin muutokset eivät näkyneet muille, toisin kuin HAProxyssa. Apacheen asennettiin mod_proxy_wstunnel, muttei sitä saatu toimimaan Teamboardin kanssa lukuisista yrityksistä huolimatta. Koska socket-yhteydet toimivat HAProxyssä ilman mitään ongelmia ja yksinkertaisilla asetuksilla, päätettiin HAProxy valita tulevaksi kuormantasaajaksi.

8.5 Mittaus 1: Alkutilanne

8.5.1 Suunnittelu

Testataan kuinka vielä kehityksessä oleva Teamboard kestää kuormaa, jotta pystytään havaitsemaan mitä tulee muuttaa ja ottaa huomioon jatkoa varten. Tässä vaiheessa Teamboard pyörii yhdellä koneella ja mittauksissa käytetään HAProxya välityspalvelimena. Ympäristön rakenne käy ilmi kuviosta 19.



Kuvio 19. Ympäristön rakenne alkutilanteessa

8.5.2 Toteutus

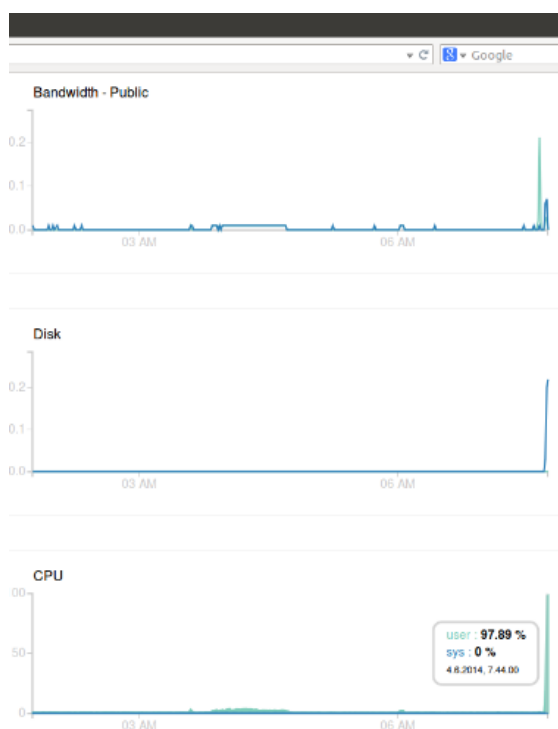
Testit suoritetaan useammalla ohjelmalla, jotta löydetään parhaimmat ohjelmat myöhempiä testejä varten. Testeissä käytetään 900 käyttäjää, koska testikone jumittuu sitä useammalla käyttäjällä SoapUI-testissä. Apache Jmeter-

ohjelmalla kirjaudutaan sisään 900:llä käyttäjällä. LoadUI ja SoapUI yhdistelmällä kirjaudutaan sisään 900:llä käyttäjällä. Suoritetaan myös SoapUI:n testi ilman LoadUI:ta, jotta nähdään tarvitaanko LoadUI:ta ollenkaan. Tutkitaan mitä testeistä selviää ja seurataan DigitalOceanin sivustolta välityspalvelin- ja Teamboard-virtuaalikoneiden reaaliaikaisia tilastoja testien aikana.

8.5.3 Tulokset

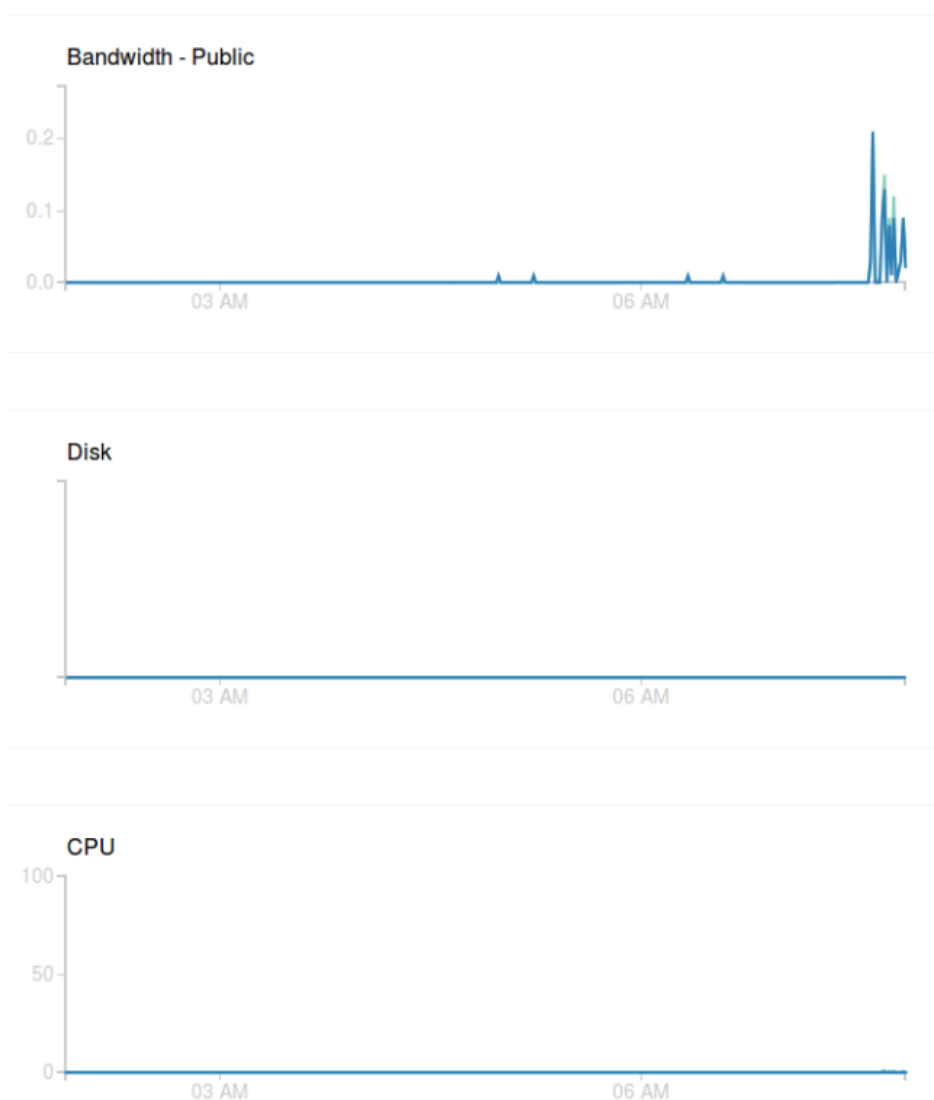
Jmeter Testi

Testin pyörittyä hetken aikaa ohjelma jäättyi useaksi sekunniksi ja jatkoi sitten toimintaa. Testin loputtua mennään selaimen kautta Teamboardiin, mutta sivusto ei vastaa. Seuraavaksi tarkastetaan DigitalOceanista Teamboard- ja HAProxy-laitteiden tilastot testien ajalta. Kirjautumalla sisään DigitalOceaniin ja valitsemalla Droplets valikosta Teamboard-virtuaalikone ja sen alta Graphs välilehti, saadaan näkyviin kuvion 20 mukaiset tilastot.



Kuvio 20. Teamboard-tilastot testin aikana

Kuviosta nähdään prosessorin käytön olleen vähäistä, kunnes lopussa sen käyttö nousee lähestulkoon sataan prosenttiin. Lopun nousu johtuu testistä ja selittää myös miksei Teamboard-sivustoon saatu yhteyttä. Prosessori joutui liian koville testin takia. Tarkastetaan HAProxyn tilastot testin ajalta valitsemalla Droplets valikosta HAProxy ja sieltä välilehti Graphs. Kuviossa 21 näkyy kyseisen sivun tilastot.



Kuvio 21. HAProxy:n tilastot testin aikana

Kuviota tutkimalla havaitaan, ettei testi aiheuttanut HAProxylle mitään ongelmia prosessorin puolella. Prosessorin käytössä näkyy minimaalista toimintaa samaan aikaan kun julkisesta rajapinnasta on tullut liikennettä.

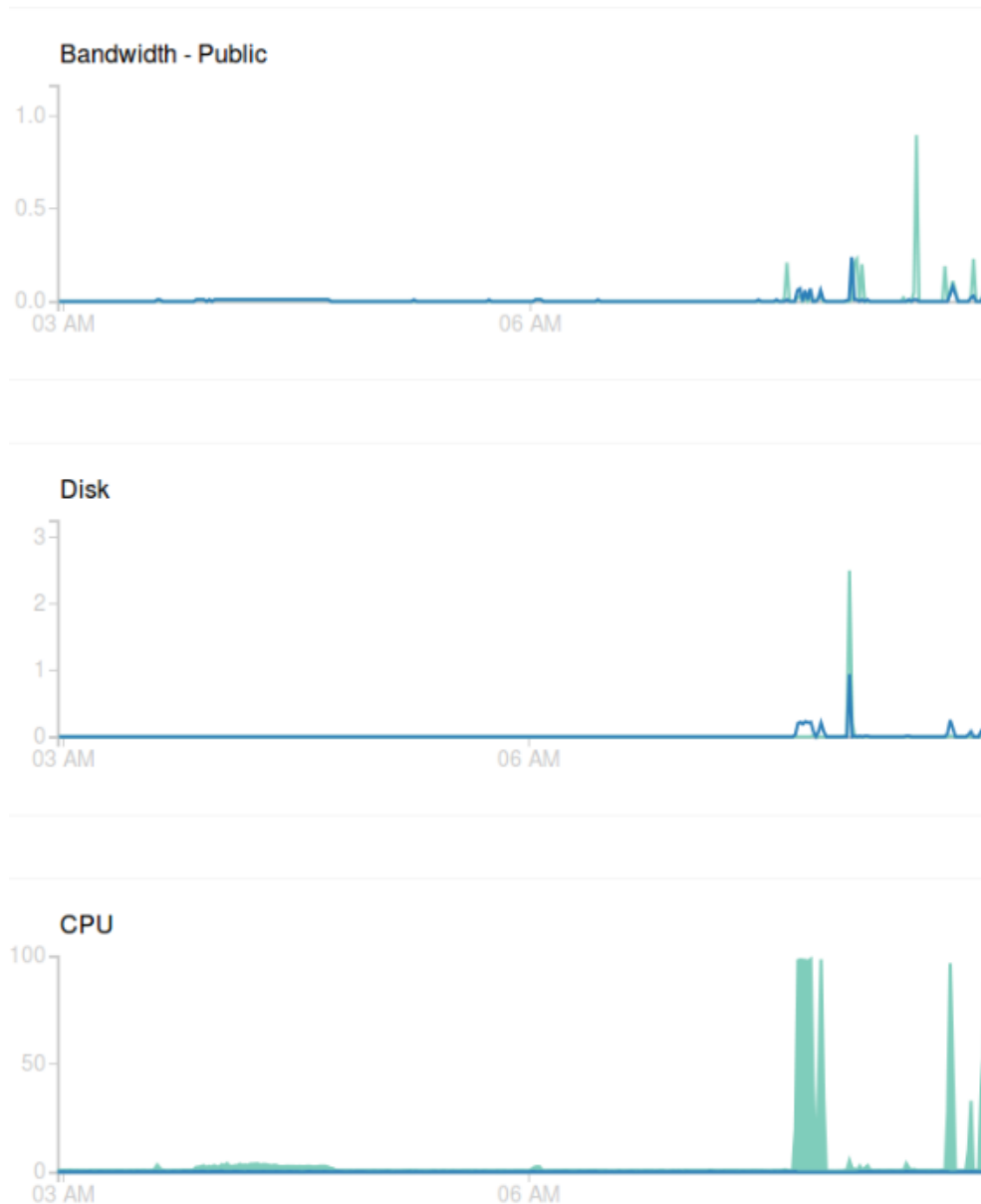
LoadUI- ja SoapUI-Testi

Pelkän SoapUI-testin ja testin tulokset ovat nähtävillä kuviossa 22.

SpamBoard										
Threads: 900					Strategy: Simple		Test Delay: 10		Limit: 1	
							Random: 0.5		Runs per Thread: 100%	
Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
HTTP Test Request	117	19795	8,590.71	10271	900	7.9	573300	5036	0	0
Login	590	19878	9,617.4	12190	900	7.9	49500	434	0	0
getToken	1	63	1.32	1	900	7.9	0	0	0	0
createBoard	39	1449	32.94	44	900	7.9	68744	603	440	48
setBoardID (disabled)	0	0	0	0	0	0	0	0	0	0
createTicket (disabled)	0	0	0	0	0	0	0	0	0	0
Test Case:	747	41185	18,242...	22506	900	7.9	691544	6074	440	48
Show Types: - All - Show Steps: - All -										
time	type	step	message							
2014-06-18 09:22:32.253	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:32.647	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:32.657	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:33.051	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:33.060	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:33.456	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:33.480	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:33.484	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:33.862	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:34.380	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:34.382	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:34.853	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:35.132	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:35.538	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:35.542	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:35.949	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:36.354	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							
2014-06-18 09:22:36.366	Step Status	createBoard	TestStep [createBoard] result status is FAILED; [java.net.Soc...							

Kuvio 22. Kuormitustesti suoraan SoapUI:sta

Kuviosta 23 nähdään DigitalOceanin reaaliaikaisia tilastoja Teamboard-virtuaalikoneesta molempien testien ajalta. Kolme viimeistä palkkia prosessorin käytössä tulevat LoadUI ja SoapUI testeistä; ensimmäinen on LoadUI:ssa ajetusta testistä, toinen palkki tuli kannettavan tietokoneen langattoman yhteyden katkeamisesta kesken testin ja viimeinen palkki on SoapUI-testistä.



Kuvio 23. LoadUI- ja SoapUI-testien aiheuttamat rasitukset Teamboardille

8.5.4 Tulosten analysointi

Testeistä käy ilmi, että kuorman kasvaessa alkaa ilmetä viivettä, hidastumista ja pahimmassa tapauksessa Teamboardin kaatuminen. Jo pelkästään se, että 900 käyttäjää muodostaa yhteyden Teamboard-sivustoon lyhyessä ajassa nostaa viivettä, muttei kuitenkaan kaada järjestelmää. Toisaalta jo pelkästään

900 kirjautumista lyhyessä ajassa kuormittaa järjestelmää liikaa. Testien perusteella voidaan todeta, ettei Teamboard kestä tässä vaiheessa kuormaa, mutta HAProxylla ei ole mitään ongelmia 900 käyttäjän hallinnassa.

Tutkimalla kuviota 22 havaitaan, että kirjautumisessa ei tapahdu virheitä vaan virheet tapahtuvat vasta laudan luomisessa. Virhe suhde on 48 % eli noin puolet lautojen luomisista epäonnistuu. Kuviosta pystytään myös havaitsemaan, että laudan luomisen epäonnistumiset johtuvat java.net.socketista.

Testityökaluista voidaan todeta, että tärkeimmäksi muodostui SoapUI, koska siitä pystytään näkemään tarkempia tietoja epäonnistuneista testien vaiheista, joten sitä ei ole mitään syytä yhdistää LoadUI:n kanssa. LoadUI alkoi kaatamaan aina käynnistämisen yrityksen yhteydessä koko käyttöjärjestelmän koneelta, jossa se oli.

8.6 Mittaus 2: Tietokannan erottaminen

8.6.1 Suunnittelu

Ensimmäisten mittauksien perusteella Teamboard muodostui pullonkaulaksi. Tästä johtuen Teamboardin komponentteja erotetaan eri koneille, jotta jatkossa olisi helpompaa seurata, mikä komponenteista muodostuu pullonkaulaksi. Aiemmin Teamboardin kaikki toiminnot, sekä Teamboardin käyttämä tietokanta MongoDB, olivat samalla koneella. MongoDB asennetaan eri koneelle, sekä se hajautetaan vielä kolmeen osaan: Config-,query,shard-palvelimeksi. Config-palvelimen tehtävänä on hallita kaikkia muita osia ja pitää yllä tietoja niistä. Query-palvelin toimii kuormantasaajana välissä ja ohjaa liikenteen eri shardeille. Shard-serverit sisältävät tietokannat. Kolmeen osaan hajauttaminen tehdään siksi, että skaalautuvuus paranee tulevaisuudessa huomattavasti, koska ei tarvitse lisätä muuta kuin shardeja. Muutosten jälkeen suoritetaan samat mittaukset kuin Mittauksessa 1, tarkoituksena on tutkia kestääkö

Teamboard muutosten jälkeen paremmin kuormaa. Mikäli Teamboard ei kestä kuormaa, pyritään paikallistamaan mikä komponentti aiheuttaa ongelmia.

8.6.2 Toteutus

Asennuksessa käytettiin DigitalOceanista löytyvää MongoDB:n asennus skriptiä:

```
apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
echo "deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
dist 10gen" | tee -a /etc/apt/sources.list.d/10gen.list
apt-get -y update
apt-get -y install mongodb-10gen
```

Kyseinen skripti lisää kustomoidun repositorion, päivittää aptituden ja asentaa MongoDB:n. Nämä komennot tallennettiin tiedostoon mongon_asennus.bash. Kyseinen skripti suoritetaan käskyllä:

```
sudo bash ./mongon_asennus.bash
```

Tarkistetaan vielä, että asennus onnistui ja mongon olevan päällä käskyllä:

```
ps aux | grep mongo
```

Järjestelmä palauttaa tietoja prosessista mongodb, eli kaikki hyvin.

Kyseinen toimenpide tehdään kaikille koneille, joihin tulee mongon osia.

Config-palvelimen asentaminen

Luodaan koneelle uusi kansio johon tallennetaan mongon metadata.

```
mkdir /mongo-metadata
```


Tämän jälkeen käynnistetään mongod-palvelu, joka tarjoaa config-palvelimen. Kerrotaan, että se käyttää porttia 27019 ja metadatan hakemisto:

```
mongod --configsvr --dbpath /mongo-metadata --port 27019
```

Query-palvelimen asentaminen

Seuraavalta koneelta pysäytetään mongodb palvelu käskyllä:

```
service mongodb stop
```

Kuormantasausta ja query-palvelimen tarjoava palvelu on mongos, joka käynnistetään seuraavaksi porttiin 27019 käskyllä:

```
mongos --configdb *Config-palvelimen IP-osoite*:27019
```

Jos käytössä olisi useampi config-palvelin, niin sitten luettelaisiin kaikkien palvelimien IP-osoitteet erottamalla ne toisistaan pilkulla.

Shard-palvelimen asentaminen

Viimeinen kone asennetaan seuraavaksi toimimaan Shard-palvelimena. Ensin koneelta otetaan yhteys query-palvelimeen käskyllä:

```
mongo --host query0.example.com --port 27017
```

Näin saadaan auki mongon oma konsoli, jolle kerrotaan mihin koneeseen/koneisiin halutaan shardit asentaa käskyllä:

```
sh.addShard( "shard0.*halutun koneen IP-osoite*:27017" )
```

Tässä tapauksessa halutun koneen IP-osoite on tämän viimeisen koneen IP-osoite.

MongoDB:n asennuksen testaaminen

Edellä mainittujen asetusten jälkeen luodaan kopio aiemmasta Teamboardista DigitalOceanissa. Kopion luominen tapahtuu sammuttamalla Teamboard virtuaalikone ja painamalla "Take a Snapshot"-nappia. Tämän jälkeen luodaan uusi kone ja valitaan "Select Image" kohdasta juuri luotu snapshot. Seuraavaksi muokataan Teamboardia käyttämään uutta MongoDB:tä, menemällä uuteen Teamboard koneeseen. Koneella muokataan common tiedostoa, joka löytyy hakemistosta ".lib/config/". Aiemmin tässä tiedostossa on asetettu

*host: 'mongodb://localhost:*portti*'*

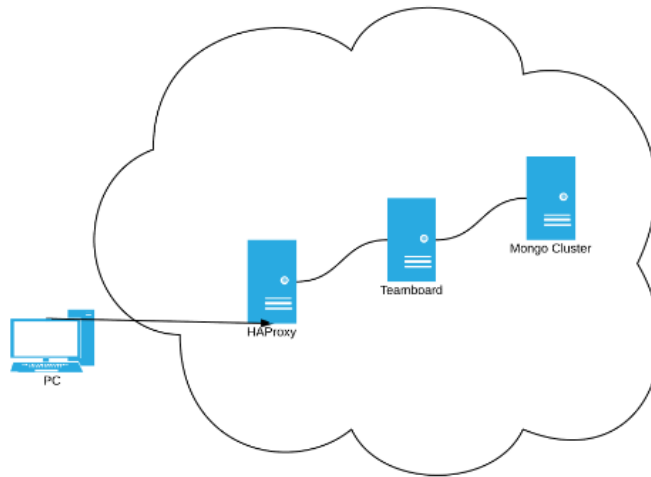
osoitetaan hostiksi nykyinen mongoDB:

host: 'mongodb://188.226.249.6:27017'

Tallennetaan tiedosto, ja tämän jälkeen käynnistetään Teamboard menemällä hakemistoon "deploy@~/node_modules/teamboard" ja antamalla siellä käsky:

forever start lib/main.js

Teamboardin toimivuus testataan menemällä selaimella osoitteeseen 188.226.184.57 ja koska Teamboardin etusivu aukeaa, on kaikki kunnossa. Toiminta varmistetaan luomalla käyttäjä juu@juu.juu salasanalla juu. Käyttäjän luonti onnistuu ja hänellä voi kirjautua sisään, eli kaikki on täysin toiminnassa. Muutosten vaikutus ympäristöön käy ilmi kuviosta 24, jossa tietokanta on erotettuna Teamboard-koneesta.



Kuvio 24. Muutosten vaikutus ympäristöön

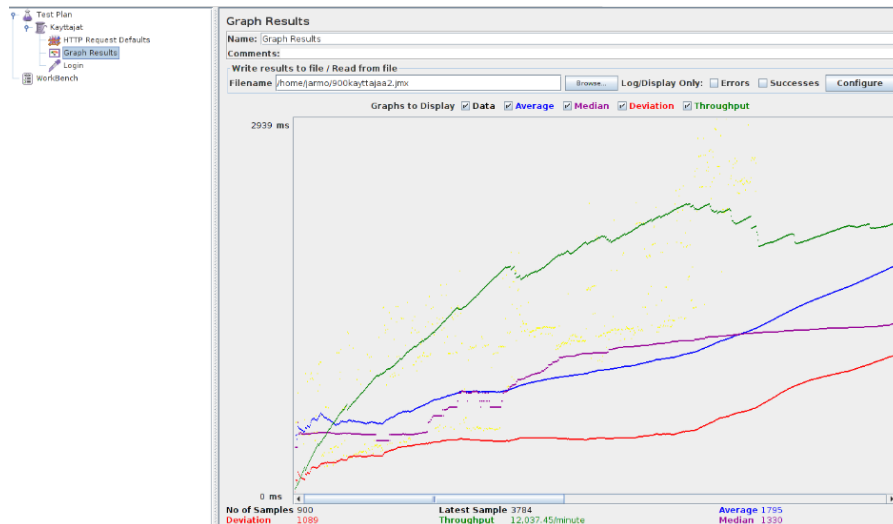
8.6.3 Tulokset

MongoDB:n shard-palvelimen tilastot DigitalOceanista kaikkien testien ajalta ovat kuviossa 25. Tutkimalla kuviota havaitaan, ettei tietokannan prosessorille tule paljoa kuormaa.



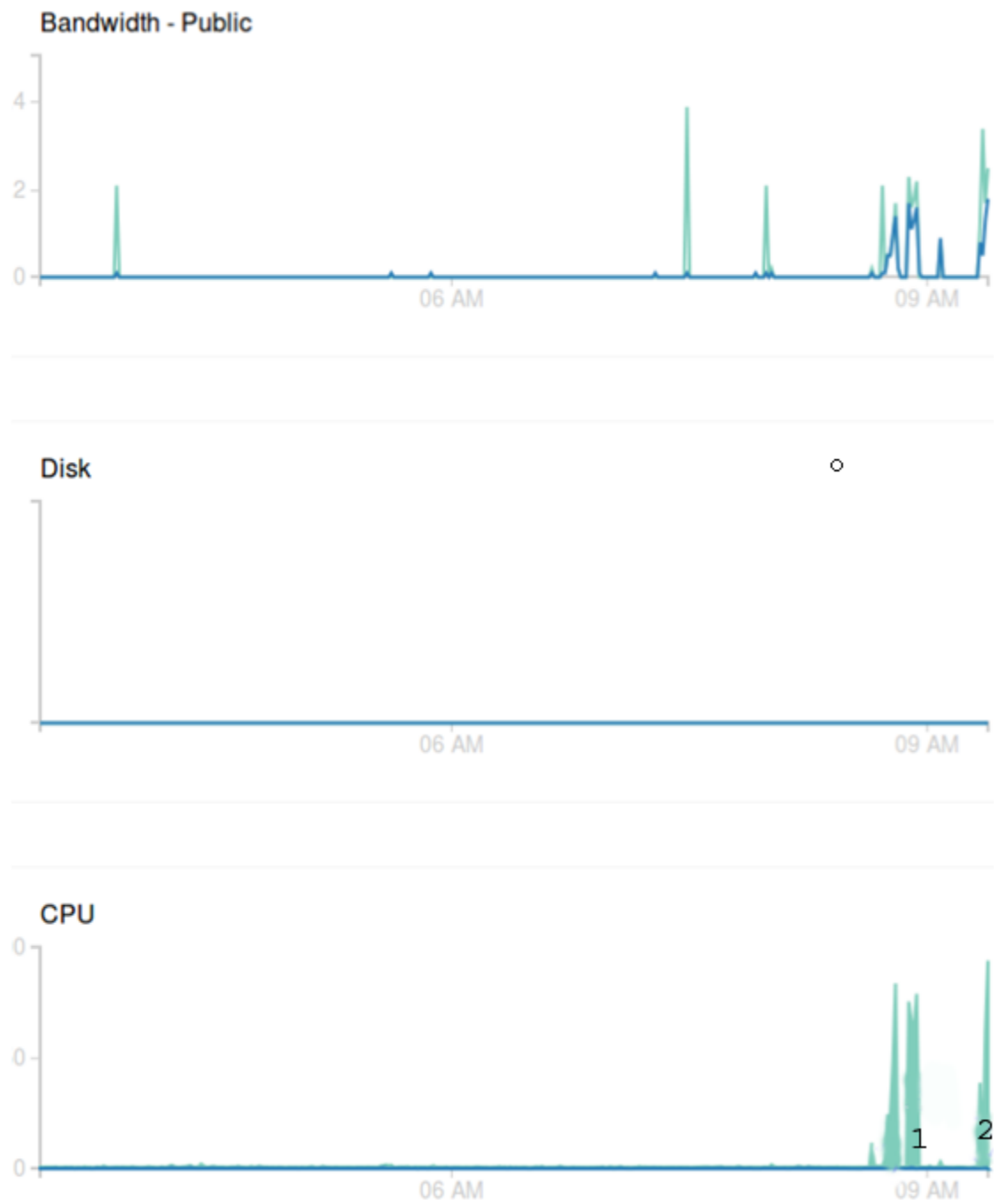
Kuvio 25. Mongo Shardin prosessorin tilastot testin aikana

Jmeterin Login testin ohjelman graafiset tulokset ovat kuviossa 26.



Kuvio 26. Jmeter Login-testin graafinen seuraaminen uudelle Teamboardille

Kuviossa 27 on DigitalOceanin tilastot uudelle Teamboard-konelle testien ajalta. Kuviossa on merkitty eri testien kohdat numeroin: 1. JMeter 2. SoapUI.



Kuvio 27. Testien aiheuttamat rasitukset uudistetulle Teamboardille

8.6.4 Analysointi

Testien perusteella pystytään toteamaan, että tietokantapalvelimen erottaminen auttoi Teamboardia kestämään kuormaa, koska se vähentää Teamboard-koneen prosessorin taakkaa. Uusittu Teamboard pystyi käsittelemään kuormaa jo hieman paremmin, sillä SoapUI testi oli ainoa joka sai prosessorin kuormituksen nousemaan sataan prosenttiin ja jumittamaan hetkellisesti Teamboardin. Muiden testien aikana, sekä niiden jälkeen, Teamboard toimi normaalisti. Ero johtui tietokannan erottamisesta Teamboard-palvelimesta. Testin perusteella voidaan myös todeta, ettei 900 taulun luominen aiheuta mitään ongelmia tietokannalle.

8.7 Mittaus 3: Kuormantasaajan testaaminen

8.7.1 Suunnittelu

Kuormantasaajan hyödyn testaamiseksi asetetaan HAProxy toimimaan kuormantasaajana ja laitetaan se jakamaan kuormaa kahdelle Teamboard koneelle. Nämä kaksi Teamboard konetta eivät osaa vielä kommunikoida keskenään, mutta pystymme havaitsemaan kuinka paljon kuormantasaaja rasittuu tästä sekä kuinka paljon se lisää yhdenaikaisten käyttäjien määrää. Testataan myös kuinka Teamboardin rasitukseen vaikuttaa kuormantasaajan lisäämisen sijasta koneen tehojen lisääminen.

8.7.2 Toteutus

SoapUI:lla ajetaan 900 käyttäjää kuten aiemmissakin testeissä. Ensin yhtä API-konetta vastaan, sitten kahta API-konetta vastaan ja lopuksi testataan kaksi ytimistä API-konetta vastaan. Kuormitusta seurataan DigitalOceanin koneen reaaliaikaisesta näkymästä.

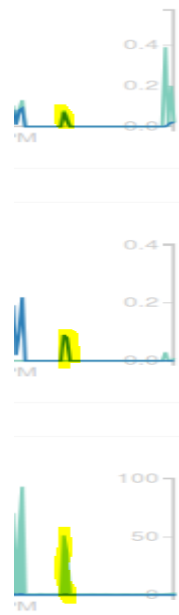
8.7.3 Tulokset

Kuviossa 28 nähdään SoapUI testin aiheuttama raskaus yhdelle koneelle sekä kuviossa 29 täysin saman testin aiheuttama raskaus kahdelle koneelle. Kyseiset testit ovat merkitty kuvioihin keltaisella. Kuviossa 30 näkyy kuinka sama testi aiheuttaa raskautta 2-ytimiselle koneelle, kuvio eroaa kahdesta muusta siksi, että se on käynnistetty juuri kyseistä testiä varten, joten se näyttää tuloksia erittäin lyhyeltä aikaväliltä



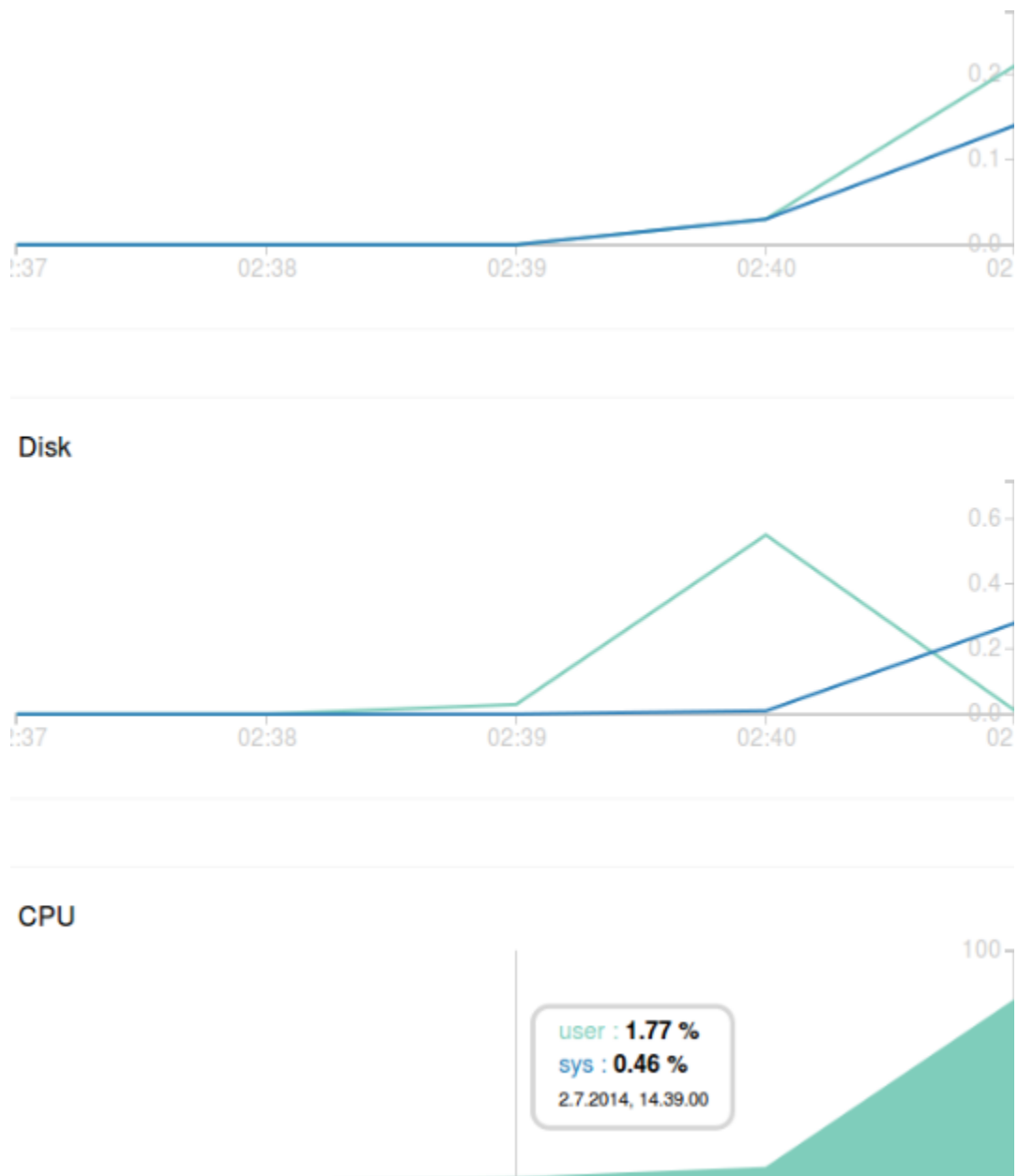
Kuvio 28. Tulokset SoapUI:lla 900 käyttäjää yhtä API-konetta vastaan

Kuviossa 29 näkyy kuinka prosessorin raskaus on vain viidessäkymmenessä prosentissa, kun käytössä on kaksi API-konetta ja käyttäjiä 900.



Kuvio 29. Tulokset SoapUI:lla 900 käyttäjää kahta API-konetta vastaan

Kuviossa 30 näkyy testin aiheuttama rasitus 2-ytimiselle koneelle. Kuviota tutkimalla voidaan havaita, että prosessorin rasitus on huomattavasti yli viidenkymmenen prosentin.



Kuvio 30. Tulokset SoapUI:lla 900 käyttäjää kaksitytimistä API-konetta vastaan

8.7.4 Analysointi

Vertailemalla kuvioita 17 ja 18 voidaan havaita, että jakamalla sama kuorma kahdelle koneelle puolitetaan yhden koneen prosessorin taakka.

Tutkimalla kuviota 19 huomataan, ettei prosessorin käyttö edes puolitu, vaikka ytimien määrä tuplattaisiin. Tämä oli tuloksena yllättävää, sillä Socket.IO:ta pystytään ajamaan useammalla ytimellä. Toisaalta tulos on ymmärrettävissä, koska koneella tapahtuu tässä vaiheessa muutakin kuin Socket.IO:lla tehtäviä toimintoja, kuten sisäänkirjautuminen ja taulujen luonti.

Voidaan myös päätellä, että yhden koneen lisääminen kaksinkertaisti Teamboardin tehot, koska kuorma jaettiin tasaisesti kahdelle koneelle. Samaa ei täysin saisi aikaiseksi nostamalla yhden virtuaalikoneen prosessorien määrän yhdestä kahteen. Tästä johtuen vaikuttaisi olevan järkevää käyttää jatkossa yksiytimisiä koneita.

8.8 Mittaus 4: Sisäänkirjaus vs taulun luonti

8.8.1 Suunnittelu

Tutkitaan tarkemmin, millaista raskautta mikäkin käyttäjän toiminto eli aiemman testin osa aiheuttaa, jotta pystytään tarkemmin määrittelemään pullonkaulaksi muodostuvat asiat. Koska aiemmissa testeissä havaittiin, että 900 käyttäjän testi yhtä API-konetta vastaan aiheuttaa prosessorille 100 %:n kuormituksen, testataan erikseen, kuinka paljon kuormaa aiheuttavat pelkät sisäänkirjautumiset ja pelkät taulunluonnit.

8.8.2 Toteutus

Jotta voidaan testata pelkästään taulunluontia, täytyy tehdä skripti, jolla lisätään oma käyttäjä ja saadaan käyttäjälle pysyvä poletti, jonka avulla sitten voidaan luoda tauluja. Tämä tapahtuu menemällä Teamboard-koneeseen, lisäämällä sinne luopoletti.js tiedosto, joka sisältää seuraavan koodin:

```
var jwt    = require('jsonwebtoken');
var mongoose = require('mongoose');

mongoose.connect('host', 'port');
mongoose.model('user', require('./config/schemas/user'));

var User = mongoose.model('user');
var user = new User({ email: 'testi@testi.com', password: 'testi' });

user.save(function(err, user) {

    if(err) {
        return console.log(err);
    }

    var config = require('./config');

    user.token = jwt.sign({ id: user.id }, config.token_secret);
    user.save(function(err, user) {

        if(err) {
            return console.log(err);
        }
        console.log('access-token:', user.token);
    });
});
```

```

        process.exit(0)
    });
});

```

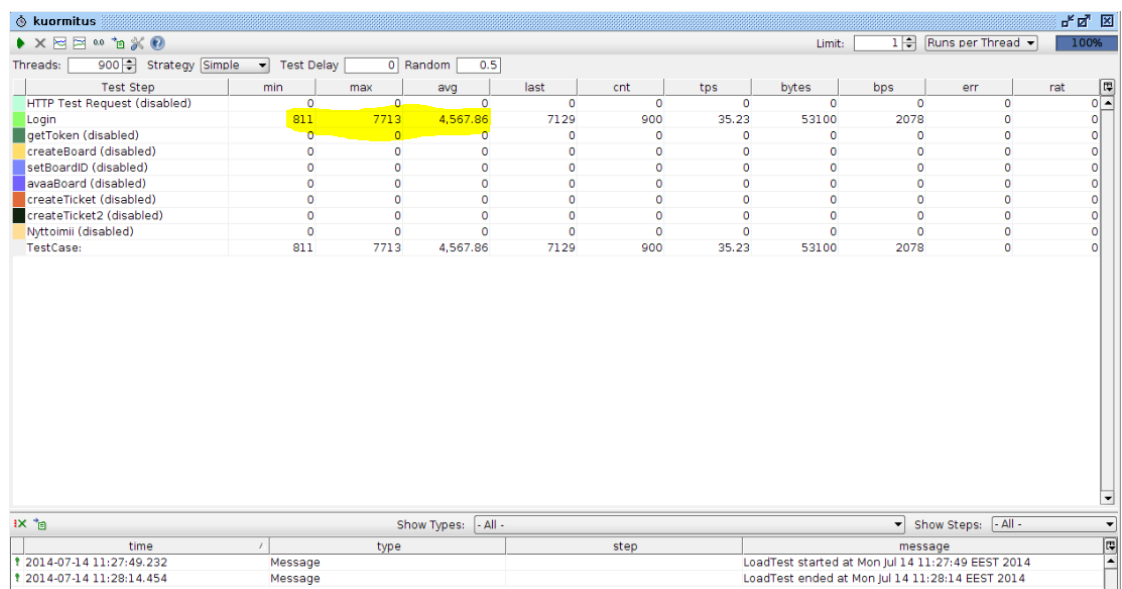
Tämän jälkeen kyseinen skripti suoritetaan käskyllä:

```
node luopoletti.js
```

Skriptin suorittamisen jälkeen tulostuu näytölle poletti, jolla käyttäjä voi tunnistautua. Kyseinen poletti asetetaan lähetettäväksi taulun luonnissa otsikko kentässä Access-token kohdassa.

8.8.3 Tulokset

Kuviossa 31 nähdään kuinka 900 sisäänkirjautumista suoritettiin SoapUI:stä, kuviossa on merkitty keltaisella viiveet.

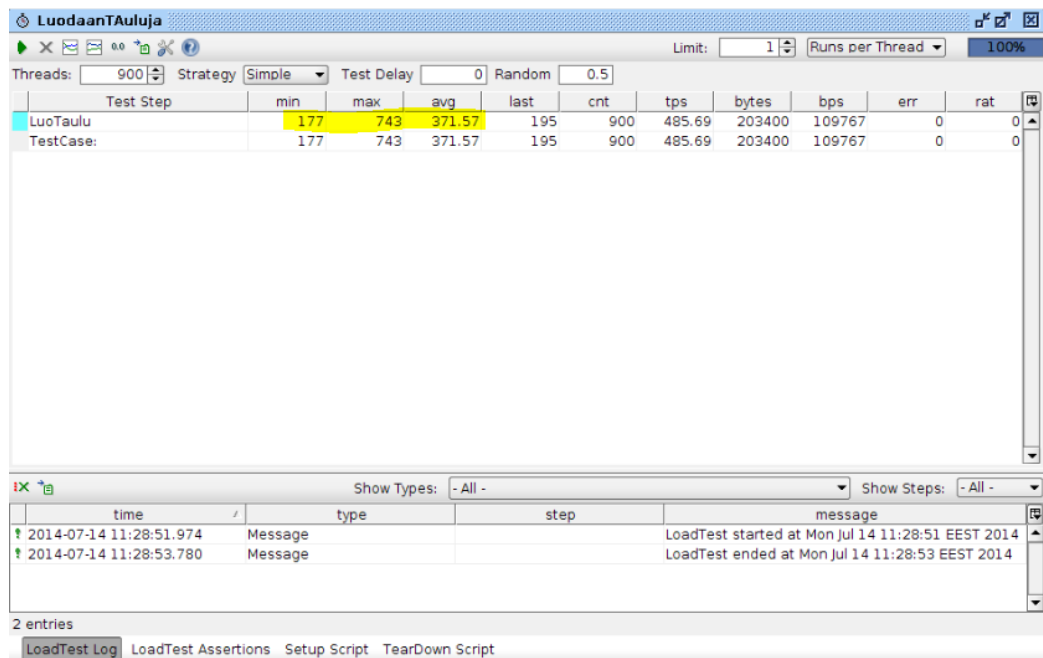


Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
HTTP Test Request (disabled)	0	0	0	0	0	0	0	0	0	0
Login	811	7713	4,567.86	7129	900	35.23	53100	2078	0	0
getToken (disabled)	0	0	0	0	0	0	0	0	0	0
createBoard (disabled)	0	0	0	0	0	0	0	0	0	0
setBoardID (disabled)	0	0	0	0	0	0	0	0	0	0
avaaBoard (disabled)	0	0	0	0	0	0	0	0	0	0
createTicket (disabled)	0	0	0	0	0	0	0	0	0	0
createTicket2 (disabled)	0	0	0	0	0	0	0	0	0	0
Nyttoimii (disabled)	0	0	0	0	0	0	0	0	0	0
Test Case:	811	7713	4,567.86	7129	900	35.23	53100	2078	0	0

time	type	step	message
2014-07-14 11:27:49.232	Message		LoadTest started at Mon Jul 14 11:27:49 EEST 2014
2014-07-14 11:28:14.454	Message		LoadTest ended at Mon Jul 14 11:28:14 EEST 2014

Kuvio 31. SoapUI 900 sisäänkirjautumista

Kuviossa 32 nähdään, kuinka 900 taulunluontia suoritetaan SoapUI:stä, vii-
veet on merkitty keltaisella.



Test Step	min	max	avg	last	cnt	tps	bytes	bps	err	rat
LuoTaulu	177	743	371.57	195	900	485.69	203400	109767	0	0
Test Case:	177	743	371.57	195	900	485.69	203400	109767	0	0

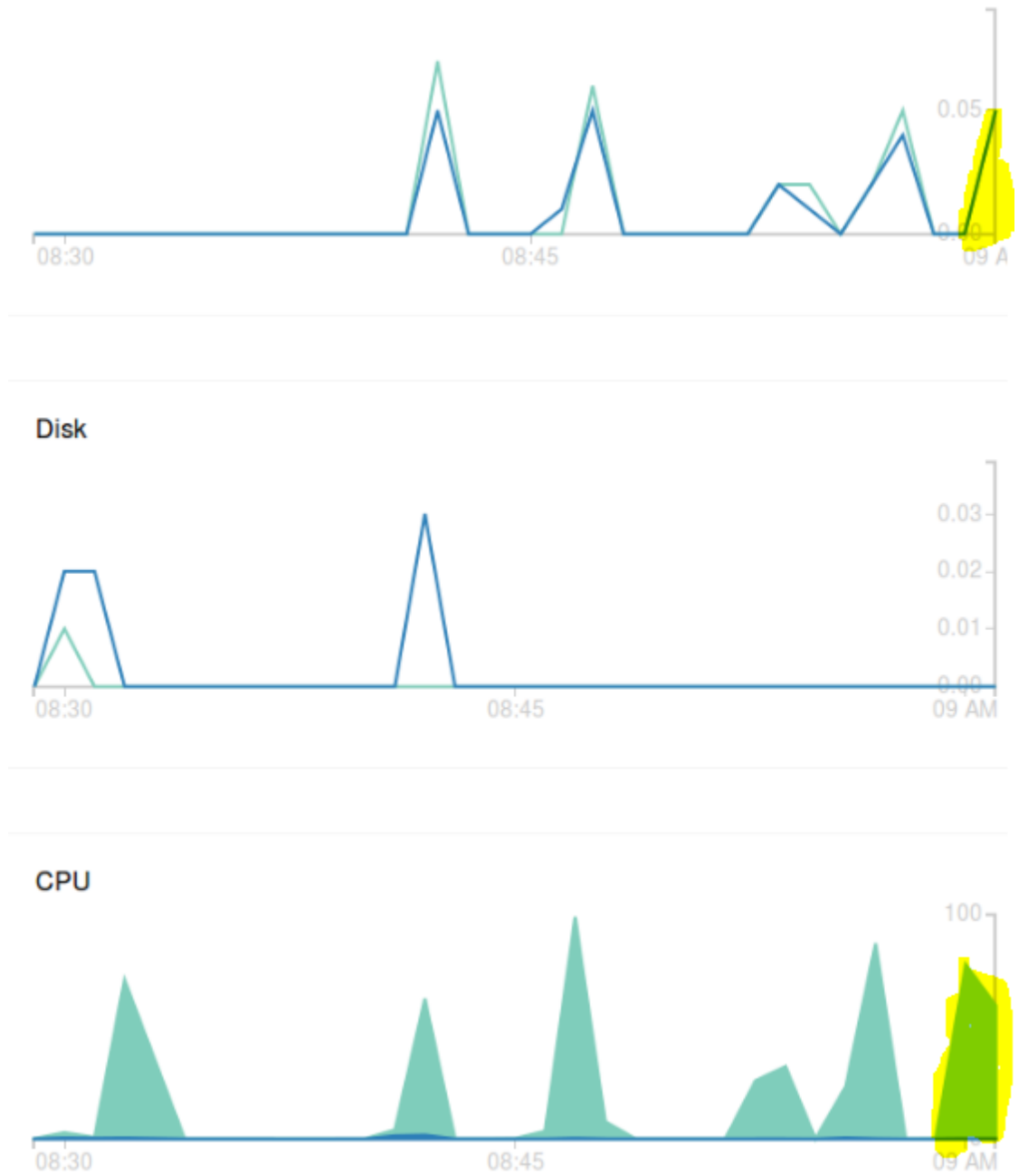
time	type	step	message
2014-07-14 11:28:51.974	Message		LoadTest started at Mon Jul 14 11:28:51 EEST 2014
2014-07-14 11:28:53.780	Message		LoadTest ended at Mon Jul 14 11:28:53 EEST 2014

2 entries

LoadTest Log LoadTest Assertions Setup Script TearDown Script

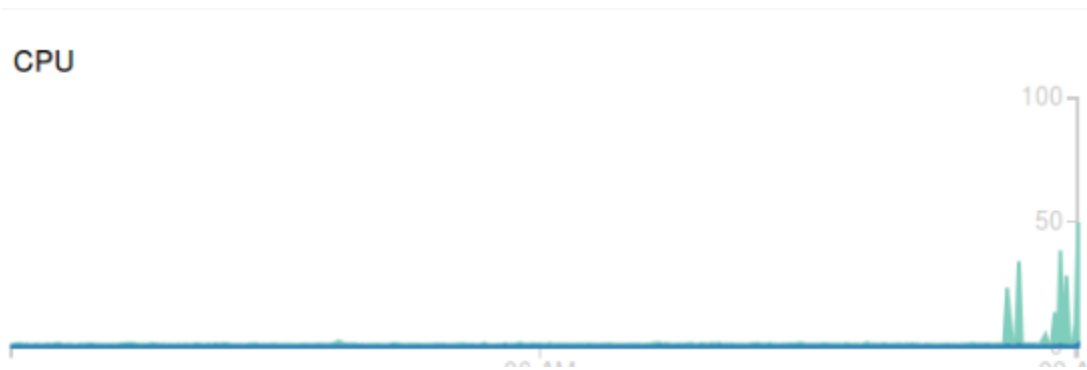
Kuvio 32. SoapUI 900 taulunluontia

Kuviossa 33 nähdään, kuinka paljon kuormitusta pelkästään 900 sisäänkirjau-
tumista aiheuttaa yhdelle koneelle.



Kuvio 33. 900 sisäänkirjautumista yhdelle koneelle

Kuviossa 34 Näkyy kuinka 4000 taulunluontia aiheuttaa rasitusta n.50% yhdelle koneelle.



Kuvio 34. 4000 taulun luonnin rasitus yhdelle koneelle

8.8.4 Analysointi

Tutkimalla viiveitä ja kuormitusta pystytään havaitsemaan sisäänkirjautumisen olevan huomattavasti hitaampi ja rasittavampi toiminto kuin taulun luomisen. Huomataan myös olevan kevyempää luoda 4000 taulua, kuin kirjautua sisään 900 kertaa. Sisäänkirjautumisen raskaus johtuu siitä, että tunnistautumisessa käytetään Bcrypt salausta, joka on hidas operaatio. Tästä voidaan päätellä, että pullonkaulaksi voi myöhemmin muodostua useiden käyttäjien yhtäaikainen sisäänkirjautuminen. Teamboard-koneen kuorma oletettavasti kevenisi huomattavasti, mikäli Bcrypt erotettaisiin omaksi palveluksi.

8.8.5 Jatkotoimenpiteet

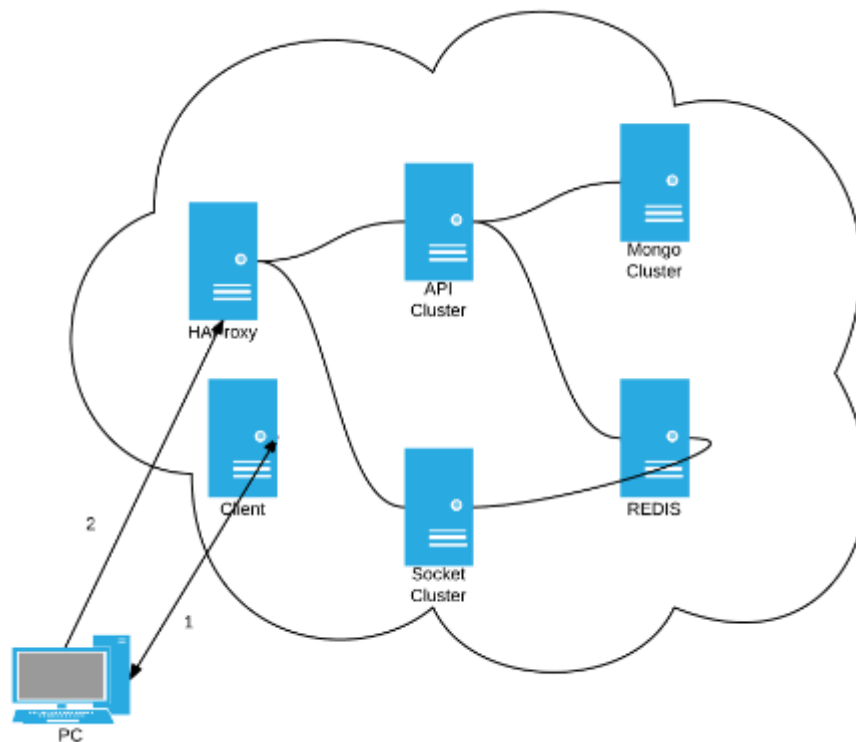
Ongelmien johtuessa itse Teamboardin komponenteista, kuormantasaajan lisääminen ympäristöön on tässä vaiheessa hyödytöntä. Vaikka lisäämällä kuormantasaaja saataisiin lisättyä samanaikaisten käyttäjien määrää, pitäisi myös lisätä Teamboard-koneita samalla. Siitä huolimatta ongelmaksi muodostuisi se, että Teamboardit eivät osaisi jutella keskenään, vaan olisivat omia

instanssejaan, eli jokainen Teamboard olisi erillään muista. Teamboardin on kuitenkin tarkoitus pitää reaaliaikaista yhteyttä kaikkien käyttäjien välillä ja sen on tarkoitus pitää kaikki käyttäjät samassa palvelussa, joten muutoksia täytyy tehdä ennen kuin kuormantasaajasta on hyötyä.

8.9 Mittaus 5: Tehokäyttäjien simulointitestit

8.9.1 Suunnittelu

Kuviossa 35 näkyy kuinka Teamboardin komponentit ovat erotettu toisistaan.



Kuvio 35. Hajautettu Teamboard

Osoite teamboard.n4sjamk.org on asetettu osoittamaan Client-koneelle. Kun asiakas menee kyseiseen osoitteeseen, muodostaa hän kuvion 25 kohdan 1.

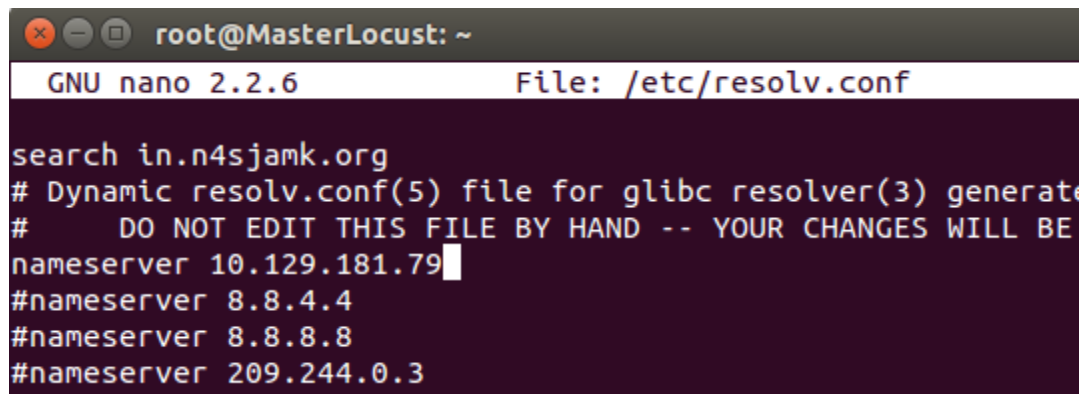
mukaisen yhteyden internetin yli client-koneeseen. Tämän jälkeen client-kone on hänen selaimessaan, joka muodostaa kuviossa näkyvän kohdan 2 mukaisen yhteyden HAProxyyn. HAProxyn kautta asiakas lähettää pyyntöjä API-koneisiin. Socket yhteydet muodostuvat HAProxyn kautta Socket-koneisiin, käyttämällä osoitteita `api.teamboard.n4sjamk.org` ja `ws.teamboard.n4sjamk.org`. HAProxyyn pystytään helposti lisäämään koneita joille kyseiset osoitteen ohjataan. API-koneet hoitavat rekisteröinnin, sisäänkirjautumisen, sekä screenshottien ottamisen Teamboardin taulujen tiloista. API-koneet muodostavat yhteyden MongoDB-koneeseen, joka taas tasaa tietokantaan menevät asiat useammalle MongoShard-koneelle. API-kone lähettää myös muutos ilmoitukset REDIS-koneelle, joka välittää muutokset websocket-yhteyksille.

8.9.2 Testiympäristön valmistelu

Aiemmissa testeissä kävi ilmi, ettei yksikään testityökaluista osaa simuloida useampaa oikeaa käyttäjää yhtäaikaaisesti tai jos osasi, niin yli tuhannen yhteyden muodostaminen yhdeltä koneelta aiheutti ongelmia. Koska tarkoituksena on pystyä simuloimaan useaa yhtäaikaista käyttäjiä sekä pystyä tuottamaan kuormaa enemmän kuin 1000 käyttäjää, toteutetaan seuraavat testit LocustIO-ohjelmalla, sillä sen testit voidaan ajaa useammalla koneella, eikä koneessa tarvitse olla graafista käyttöliittymää.

Eri verkoista johtuvien viiveiden ja vaikutusten välttämiseksi luodaan testejä varten virtuaalikoneita DigitalOceaniin. LocustMaster-koneelta, joka ei suorita testejä vaan tarjoaa websivuston, testejä voidaan hallita ja seurata. Luodaan useampi LocustSlave-kone, jotka muodostavat yhteyden LocustMaster-koneeseen ja suorittavat itse testit. Jottei testit rasita DigitalOceanin reunalaitteita, niin asetetaan sisäverkon nimipalvelimelle osoitteen `api.in.n4sjamk.org` ja `ws.in.n4sjamk.org`, joita käytetään tulevissa Locustilla suoritettavissa testeissä. Samat osoitteet asetetaan HAProxyyn kohtiin, joissa edelliset osoitteet ovat. Jotta Locust koneet osaavat käyttää sisäistä nimipalvelinta, täytyy se

lisätä koneille sekä poistaa ulkoiset nimipalvelimet käytöstä, jottei turhaan mennä ulko verkkoon. Kuvio 36 selviää kuinka tämä tehdään.



```
root@MasterLocust: ~  
GNU nano 2.2.6 File: /etc/resolv.conf  
search in.n4sjamk.org  
# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)  
#     DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE LOST  
nameserver 10.129.181.79  
#nameserver 8.8.4.4  
#nameserver 8.8.8.8  
#nameserver 209.244.0.3
```

Kuvio 36. Sisäisen nimipalvelimen käyttö Locustille

Kyseinen asetus tulee voimaan heti, mutta ongelmana on sen katoaminen, mikäli kone käynnistetään uudelleen. Asetetaan vielä sama asetus pysyvästi, tekemällä sama muutos ”/etc/resolvconf/resolv.conf.d/head” tiedostoon.

8.9.3 Locust-asennus

Locustin asennetaan käskyllä:

```
pip install locustio
```

Jotta Locustia voi käyttää hajautetusti useammalla koneella täytyy asentaa Python ZeroMQ, joka asentuu käskyllä:

```
pip install pyzmq
```

Näiden käskyjen jälkeen Locust on valmis käytettäväksi.

8.9.4 Locust-testin luonti

Luodaan tiedosto `locustfile.py`, Locust lukee kyseisen tiedoston automaattisesti, mikäli se suoritetaan samasta kansioita jossa kyseinen tiedosto on.

Kokonaisuudessa `locustfile.py` tiedosto löytyy kohdasta Liite 3, mutta tässä selitetään tärkeimpiä koodin osia:

Luodaan tehtäväsarja ja määritellään käyttäjää luomaan alussa itselleen taulukot `boards` ja `tickets`. Lisäksi määritetään se luomaan satunnainen käyttäjänimi, salasana `test_password` ja määritetään, ettei polettia ole. Tämän jälkeen määritetään se rekisteröitymään kyseisillä tunnuksilla. On erittäin tärkeää määrittää kyseiset taulut tyhjiksi juuri `on_start` kohdan jälkeen, koska silloin ohjelma ymmärtää, että kyseiset taulukot ovat omat jokaiselle käyttäjälle.

class TeamboardTasks(TaskSet):

```
def on_start(self):
    self.boards = [ ]
    self.tickets = [ ]
    self.user = {
        'email': 'test_' + urandom(16).encode('hex') + '@garnet.red',
        'password': 'test_password'
    }
    self.token = None
    self.client.post('/auth/register', self.user)
```

Seuraavaksi asetetaan käyttäjä kirjautumaan sisään juuri luoduilla tunnuksilla, sekä ottamaan vastauksena tuleva poletti talteen:

```
response = self.client.post('/auth/login', self.user)
self.token = response.headers['x-access-token']
```

Seuraavaksi määritellään toiminto `post_board`, jolla käyttäjä luo uuden taulun. Määritetään, ettei taulua yritetä luoda, ellei polettia ole tai jos tauluja on jo luotu enemmän kuin yksi:

```
@task(1)
def post_board(self):
    if self.token is None: return
    if len(self.boards) > 1: return
```

Taulunluomiseksi lähetetään `/boards` polkuun data kentässä tiedot nimi, info ja `isPublic` arvolla `True` tehdään taulusta julkinen:

```
response = self.client.post('/boards',
    data = {
        'name': 'Botti vauhdissa',
        'info': 'vrrrruuuummm',
        'isPublic': 'true'
    },
```

Jotta taulu voidaan luoda, täytyy otsikossa lähettää aiemmin tallennettu poletti:

```
headers = {
    'authorization': 'bearer ' + self.token + "
})
```

Lisätään vastauksesta taulun tiedot taulukkoon:

```
self.boards.append(response.json())
```

Samalla tavalla haetaan satunnainen oma taulu, mikäli poletti on olemassa, ja `"boards"`-taulukko ei ole tyhjä.

```

@task(2)
def get_board(self):
    if self.token is None: return
    if len(self.boards) is 0: return

    board = random.choice(self.boards)
    self.client.get('/boards/' + board['id'] + ",
        headers = {
            'authorization': 'bearer ' + self.token + "
        })

```

Mikäli poletti on olemassa, haetaan kaikki taulut:

```

@task(1)
def get_boards(self):
    if self.token is None: return

    self.client.get('/boards',
        headers = {
            'authorization': 'bearer ' + self.token + "
        })

```

Luodaan korkeintaan 6 lappua ja tallennetaan ne taulukkoon talteen, jotta niitä voidaan myöhemmin liikuttaa:

```

@task(3)
def post_ticket(self):
    if self.token is None: return
    if len(self.boards) is 0: return
    if len(self.tickets) > 5: return
    board = random.choice(self.boards)
    response = self.client.post('/boards/' + board['id'] + '/tickets',

```

```

        data = {
            'heading': 'Liikuteltavaa',
            'content': 'sisaltoa'
        },
        headers = {
            'authorization': 'bearer ' + self.token + "
        })

    ticket = response.json()
    self.tickets.append({
        'ticket': ticket['id'],
        'board': board['id']
    })

```

Luodaan toiminto, jossa satunnaista omaa lappua liikutetaan satunnaiseen kohtaan:

```

@task(15)
def move_ticket(self):
    if self.token is None: return
    if len(self.tickets) is 0: return

    ticket = random.choice(self.tickets)

    data = { }
    data['position'] = { 'x': 100, 'y': 300, 'z': 0 }

    response = self.client.put('/boards/' + ticket['board'] + '/tickets/' +
        ticket['ticket'] + ",
    data = json.dumps({
        'position': {
            'x': random.randint(0, 712),

```

```

        'y': random.randint(0, 556),
        'z': 0
    }
}),
headers = {
    'content-type': 'application/json',
    'authorization': 'bearer ' + self.token + "
})

```

Asetetaan luokka "TeamboardUser" käyttämään juuri luotua tehtävä sarjaa, sekä määritetään aikaväli, jolla tehtäviä suoritetaan. Alla olevassa esimerkissä odotetaan vähintään yksi sekunti ja korkeintaan kaksi sekuntia:

```

class TeamboardUser(HttpLocust):
    task_set = TeamboardTasks
    min_wait = 1000
    max_wait = 2000

```

Locustille pystyy asettamaan myös useampia luokkia samaan tiedostoon, joita pystyy ajamaan yhtä aikaa. Näin pystyisi esimerkiksi simuloimaan erilaisia käyttäytymismalleja. Eri tehtävät erotellaan toisistaan merkinnällä "@task", jonka jälkeen sulussa voidaan määritellä kuinka usein kyseistä tehtävää suoritetaan. Yllä olevassa konfiguraatiossa esimerkiksi lappujen siirtoa tehdään 15 kertaa enemmän kuin taulun luontia.

Locust käynnistetään käskyllä:

```
locust -H http://api-tb-haproxy.in.n4sjamk.org/api/v1 TeamboardUser
```

Mikäli halutaan käynnistää Locust useammalla koneella, täytyy yksi kone määrittää pääkoneeksi, joka tarjoaa myös web-käyttöliittymän. Tämä tapahtuu käskyllä:

```
locust -H http://api-tb-haproxy.in.n4sjamk.org/api/v1 TeamboardUser --master
```

Muille Locust-koneille täytyy käynnistyksen yhteydessä kertoa pääkoneen osoite. Niiden käynnistäminen tapahtuu käskyllä:

```
locust -H http://api-to-haproxy.in.n4sjamk.org/api/v1 TeamboardUser --slave -  
-master-host=10.129.171.24
```

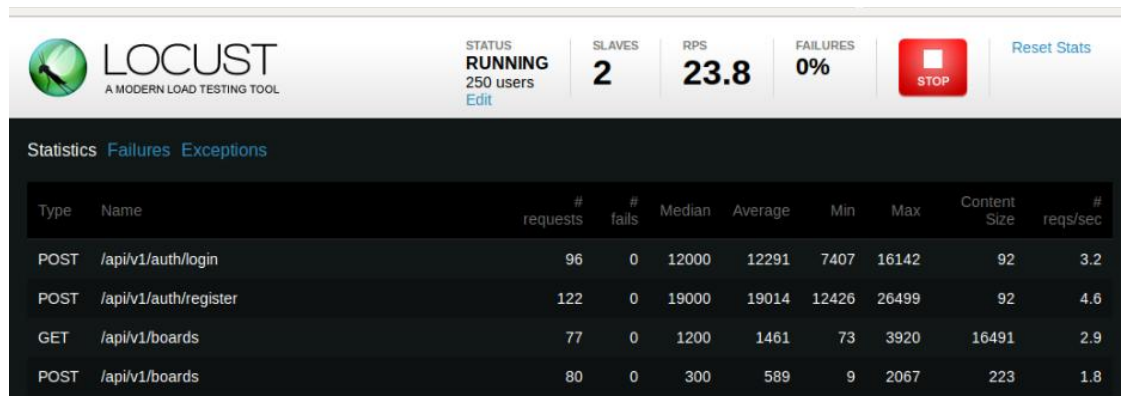
8.9.5 Toteutus

Tutkitaan Locust kuormitustestien avulla kuinka kuorman jakaminen API-koneille vaikuttaa rasitukseen. Testeissä käytetään locust.py tiedostoon luotua TeamboardUser-luokkaa, joka simuloi tehokäyttäjää lähettämällä pyyntöjä sekunnin tai kahden välein. Simuloimalla tehokäyttäjää pystymme kuitenkin tehokkaasti tutkimaan API-koneiden määrän vaikutusta palvelun toimintaan. Tärkeää testeissä on seurata pyyntöjä sekunnissa -arvoa (RPS), sillä se kertoo montako käyttäjää lähettää pyyntöjä yhtä aikaa. Testit suoritetaan 250, 500, sekä 750 käyttäjällä.

8.9.6 Tulokset

250 Käyttäjällä


Kuviossa 37 nähdään kuormitustesti yhdelle API-koneelle. Voidaan havaita ettei virheitä tapahdu, mutta yksi API-kone vastaa todella hitaasti. Kuviossa 41 kyseisen testin rasitus on neljäs palkki oikealta, josta voidaan havaita prosessorin kuormituksen olevan n. 100 %.



Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	96	0	12000	12291	7407	16142	92	3.2
POST	/api/v1/auth/register	122	0	19000	19014	12426	26499	92	4.6
GET	/api/v1/boards	77	0	1200	1461	73	3920	16491	2.9
POST	/api/v1/boards	80	0	300	589	9	2067	223	1.8

Kuvio 37. 250 käyttäjän testi yhdelle API-koneelle

Kuviossa 38 nähdään kuormitustesti kuorman jakautuessa kahdelle API-koneelle. Virheitä ei tapahdu ja palvelu vastaa huomattavasti nopeammin edelliseen testiin verrattuna. Kuviossa 41 tämän testin rasitus on kolmas palkki oikealta, josta voidaan havaita prosessorin kuormituksen olevan n. 90 %.



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

250 users

Edit

SLAVES

2

RPS

122.2

FAILURES

0%

STOP

Reset Stats

Statistics


Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	35	0	1500	1550	885	2276	92	0
POST	/api/v1/auth/register	25	0	2900	2698	2028	3242	92	0
GET	/api/v1/boards	1633	0	230	327	35	1339	85308	6.5
POST	/api/v1/boards	423	0	23	76	7	727	223	0.1

Kuvio 38. 250 käyttäjän testi kahdelle API-koneelle

Kuviossa 39 nähdään kuormitustesti kuorman jakautuessa kolmelle API-koneelle. Virheitä ei tapahdu ja palvelu vastaa huomattavasti nopeammin edelliseen testiin verrattuna. Tuntemattomasta syystä ohjelma ei kuvaa otettaessa lähettänyt pyyntöjä kuin 113,3 kappaletta sekunnissa, joka on hitaampaa kuin edellisessä kahdelle API-koneelle tehdyssä kuormitustestissä. Kuviossa 41 tämän testin rasitus on toinen palkki oikealta, josta voidaan havaita prosessorin kuormituksen olevan noin 60 %.



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

250 users

Edit

SLAVES

2

RPS

113.3

FAILURES

0%

STOP

Reset Stats

Statistics

Failures

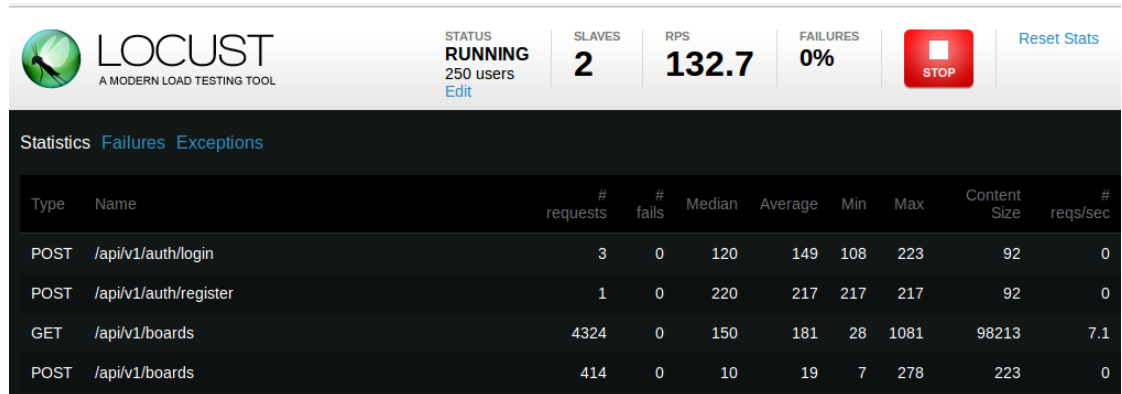
Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	4	0	160	221	112	311	92	0
POST	/api/v1/auth/register	1	0	370	371	371	371	92	0
GET	/api/v1/boards	373	0	79	83	27	379	47017	7.1
POST	/api/v1/boards	248	0	11	15	8	105	223	4

Kuvio 39. 250 käyttäjän testi kolmelle API-koneelle

Kuviossa 40 nähdään kuormitustesti kuorman jakautuessa neljälle API-koneelle. Virheitä ei tapahdu, sekä palvelu vastaa nopeammin kaikkiin edelli-

siin testeihin verrattuna. Kuviossa 41 tämän testin raskaus on ensimmäinen palkki oikealta, josta voidaan havaita prosessorin kuormituksen olevan noin 50 %.



Kuvio 40. 250 käyttäjän testi kolmelle API-koneelle

Kuviossa 41 näkyy neljässä viimeisessä palkissa kaikkien neljän kuormitus-testin aiheuttama raskaus prosessorille. Ensimmäisessä palkissa kuorma kohdistetaan yhdelle API-koneelle, toisessa kahdelle jne. Kuviosta voidaan helposti seurata kuinka kuormanjakaminen useammalle koneelle vähentää prosessorin kohdistuvaa raskautta.



Kuvio 41. Testin tulokset eri API-konemäärillä 250 käyttäjällä

Tulosten seuraamisen helpottamiseksi testien tulokset ovat yhdistetty taulukkoon 1.

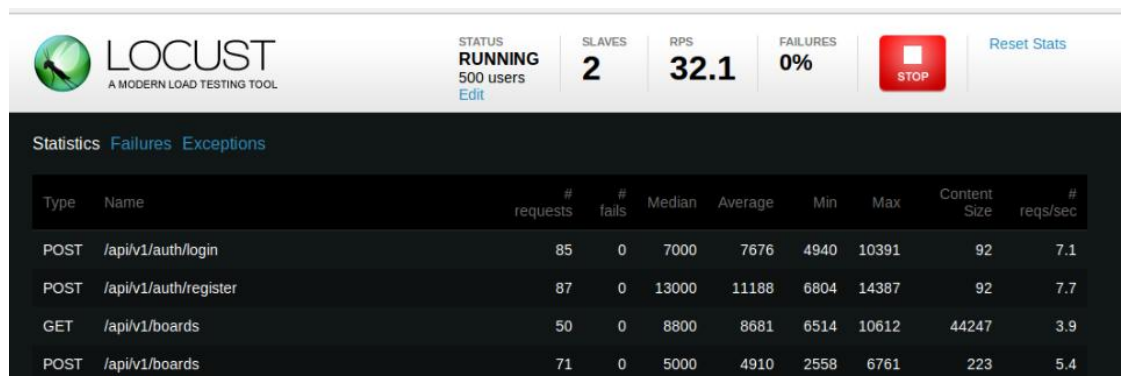
Taulukko 1. 250 tehokäyttäjän kuormitustestin tulokset

250 Teho käyttäjän testi				
API-koneiden määrä	1	2	3	4
Kestikö kuorman	Kyllä	Kyllä	Kyllä	Kyllä
Prossessorin käyttö arvio	100 %	90 %	60 %	50 %
Pyyntöjen määrä sekunnissa	23,8	122,2	113,3	132,7
Login latenssin keskiarvo ms	12291	1550	221	149

500 Käyttäjällä


Yksi API-kone ei kestänyt 500 käyttäjää.

Kuviossa 42 nähdään kuormitustesti kahdelle API-koneelle. Voidaan havaita ettei virheitä tapahdu, mutta kaksi API-konetta vastaa todella hitaasti. Kuviossa 45 kyseisen testin rasitus on ensimmäisenä, josta voidaan havaita prosessorin kuormituksen olevan noin 98 %.



Kuvio 42. 500 käyttäjän testi kahdelle API-koneelle

Kuviossa 43 nähdään kuormitustesti kuorman jakautuessa kolmelle API-koneelle. Virheitä ei tapahdu ja palvelu vastaa huomattavasti nopeammin edelliseen testiin verrattuna. Kuviossa 45 tämän testin rasitus on toisessa pal-kissa oikealta, josta voidaan havaita prosessorin kuormituksen olevan välillä 90 – 98 %.



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

500 users

Edit

SLAVES

2

RPS

154.6

FAILURES

0%

STOP

Reset Stats

Statistics


Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	27	0	1400	1323	120	2483	92	0
POST	/api/v1/auth/register	14	0	1400	1534	469	2417	92	0
GET	/api/v1/boards	411	0	460	639	95	3408	97883	12.5
POST	/api/v1/boards	307	0	190	314	7	1996	223	7.8

Kuvio 43. 500 käyttäjän testi kolmelle API-koneelle

Kuviossa 44 nähdään kuormitustesti kuorman jakautuessa neljälle API-koneelle. Virheitä ei tapahdu ja palvelu vastaa nopeammin muihin tällä käyttäjämäärällä tehtyihin testeihin verrattuna. Kuviossa 45 tämän testin rasitus on kolmas palkki, josta voidaan havaita prosessorin kuormituksen olevan noin 85- 95 %.



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

500 users

Edit

SLAVES

2

RPS

187.5

FAILURES

0%

STOP

[Reset Stats](#)

Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	14	0	390	596	105	1469	92	0
POST	/api/v1/auth/register	6	0	800	907	447	1300	92	0
GET	/api/v1/boards	847	0	360	528	82	2195	126753	11.2
POST	/api/v1/boards	461	0	110	196	7	1149	223	4.6

Kuvio 44.500 käyttäjän testi neljälle API-koneelle

Kuvioon 45 on yhdistetty 500 käyttäjällä tehtyjen testien prosessorin käytön tulokset, mukana tulokset vain niiltä API-konemääriltä jotka kestivät kuorman. Ensimmäisenä 2 API-konetta, keskellä 3 API-konetta ja viimeisenä 4 API-konetta.



Kuvio 45. Prosessorin käyttö eri API-konemäärillä 500 käyttäjällä

Taulukkoon 2 on yhdistetty 500 käyttäjän testin tulokset.

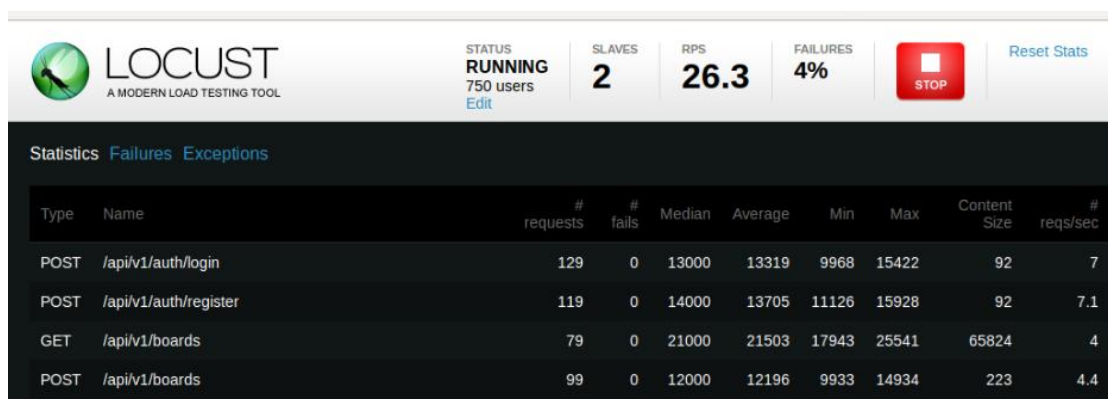
Taulukko 2. 500 tehokäyttäjän kuormitustestien tulokset

500 Teho käyttäjän testi				
Apien määrä	1	2	3	4
Kestikö kuorman	Ei	Kyllä	Kyllä	Kyllä
Prosessorin käyttö arvio	-	98 %	90–98%	85–95%
Pyyntöjen määrä sekunnissa	-	32,1	154,6	187,5
Login latenssin keskiarvo ms	-	7676	1320	596

750 Käyttäjällä

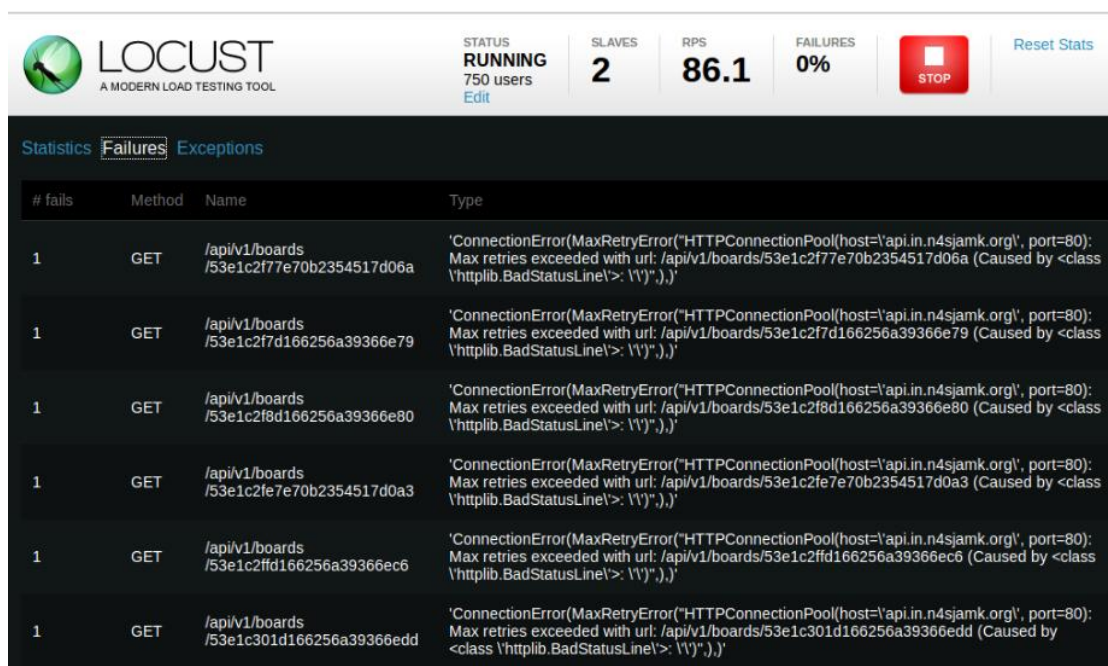
Yksi API-kone ei kestänyt 750 käyttäjää.

Kuviossa 46 nähdään kuinka kahdella API-koneella on vaikeuksia 750 käyttäjän kanssa, virheitä on tullut 4 %.



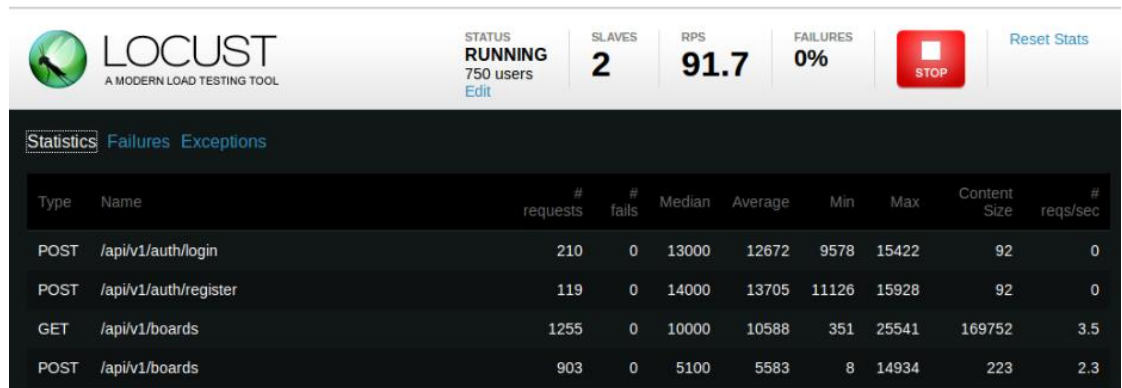
Kuvio 46. 750 käyttäjän testi kahdelle API-koneelle

Kuviosta 47 nähdään ”Failures”-välilehdellä tiedot siitä millaisia virheet olivat. Kaikki virheet olivat tapahtuneet vain yksittäisten lautojen hakemisessa, joten testiä päätettiin jatkaa, jotta nähdään miten palvelu käyttäytyy tämän jälkeen.



Kuvio 47. "Failures"-välilehti; 750 käyttäjää, 2 API-konetta

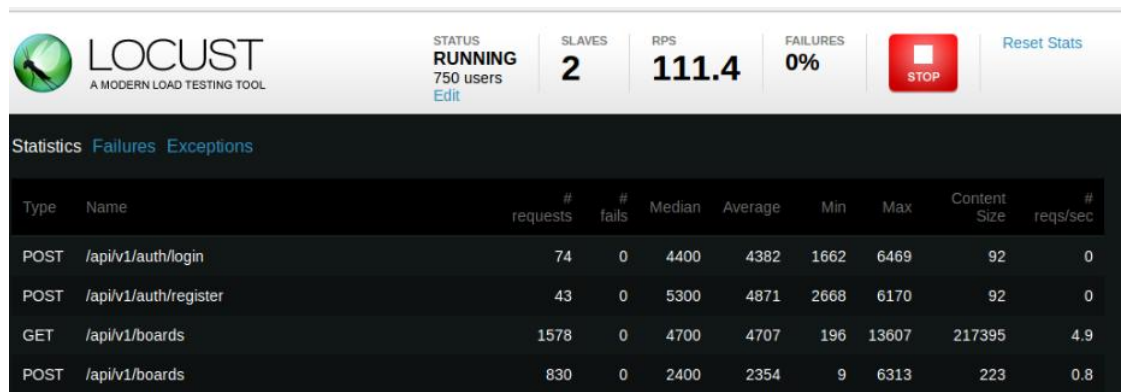
Kuviossa 48 nähdään, kuinka testin jatkuessa pitempään virheitä ei enää tule, joten virheiden määrä laskee Locustissa nollaan prosenttiin.



Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	210	0	13000	12672	9578	15422	92	0
POST	/api/v1/auth/register	119	0	14000	13705	11126	15928	92	0
GET	/api/v1/boards	1255	0	10000	10588	351	25541	169752	3.5
POST	/api/v1/boards	903	0	5100	5583	8	14934	223	2.3

Kuvio 48. 750 käyttäjän testi kahdelle API-koneelle, virheiden jälkeen

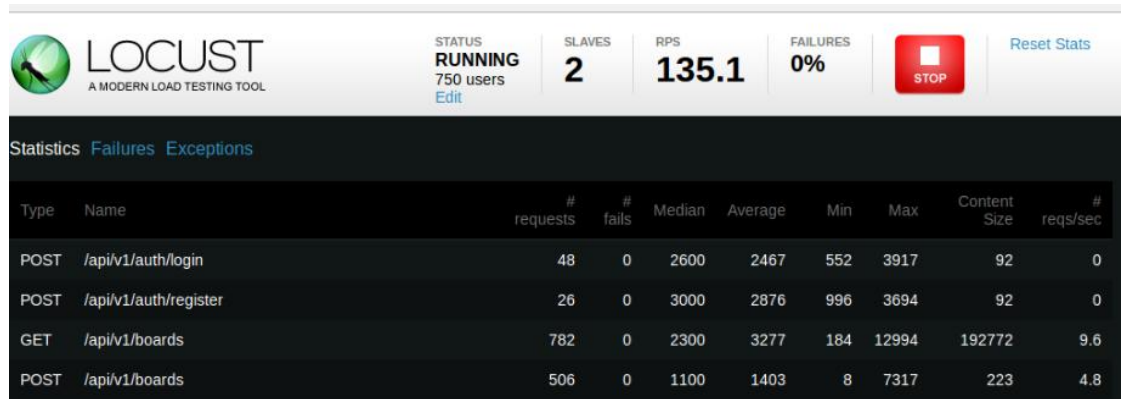
Kuviossa 49 nähdään kolmen API-koneen selviytyvän virheittä 750 käyttäjästä.



Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	74	0	4400	4382	1662	6469	92	0
POST	/api/v1/auth/register	43	0	5300	4871	2668	6170	92	0
GET	/api/v1/boards	1578	0	4700	4707	196	13607	217395	4.9
POST	/api/v1/boards	830	0	2400	2354	9	6313	223	0.8

Kuvio 49. 750 käyttäjän testi kolmelle API-koneelle

Kuviossa 50 nähdään neljän API-koneen selviytyvän virheittä 750 käyttäjästä.



Kuvio 50. 750 käyttäjän testi neljälle API-koneelle

Kuviossa 51 on 750 käyttäjällä tehtyjen testien prosessorin käytön tulokset, mukana tulokset vain niiltä API-kone määriltä jotka kestivät kuorman. Aivan oikealla oleva palkki on neljällä, sitä edeltävä kolmella ja kolmantena oikealta kahdella API-koneella.



Kuvio 51. Prosessorin käyttö eri API-konemäärillä 750 käyttäjällä

Taulukoon 3 on yhdistetty 750 tehokäyttäjän kuormitustestin tulokset

Taulukko 3. 750 tehokäyttäjän kuormitustestin tulokset

750 Teho käyttäjän testi				
Apien määrä	1	2	3	4
Kestikö kuorman	Ei	Ei	Kyllä	Kyllä
Proessorin käyttö arvio	-	-	98–99%	96 %
Pyyntöjen määrä sekunnissa	-	-	111,4	135,1
Login latenssin keskiarvo ms	-	-	4382	2600

8.9.7 Tulosten analysointi

250 Käyttäjällä

Testeistä voidaan havaita palvelun kestävän 250 tehokäyttäjää virheittä, pelkästään yhdellä API-koneella. Palvelu nopeutuu huomattavasti kun kuorma jaetaan useammalle API-koneelle. Kun kuorma jaetaan lopussa neljälle API-koneelle, niin prosessorin käyttö puolittuu siitä, mitä se oli yhdelle API-koneella jaettuna. Vaikka palvelu toimisi yhdellä API-koneella, se toimisi hyvin vain kolmella tai useammalla API-koneella.

500 Käyttäjällä

Testeistä käy ilmi, että palvelu kestää 500 tehokäyttäjää virheittä kuorman ollessa jaettuna kahdelle koneelle. Voidaan myös havaita, että API-koneiden lisääminen ei laske huomattavasti prosessorien kuormitusta, mutta auttaa palvelua toimimaan nopeammin.

750 Käyttäjällä

Kahdelle API-konelle tehdyssä testissä sattui virheitä testin alkupuolella. Testin jatkuessa virheprosentti tippui kuitenkin takaisin nolnaan. Tämän arvellaan johtuvan siitä, että alussa tapahtuva rekisteröinti ja sisäänkirjautuminen ovat raskaita osia testissä, joten ne rasittivat API-koneita niin paljon, ettei käsittelytehoa riittänyt enää kaikille muille pyynnöille. Tätä teoriaa tukee se, että virheitä ei tullut enää myöhemmin kyseisessä testissä. Vaikkei voida sanoa, että

kaksi API-konetta kestäisi 750 käyttäjää, on silti oleellista huomioida alussa pyyntöjä sekunnissa (RPS) olleen 26,3, mutta ongelmien jälkeen 91,7. Näistä voidaan päätellä, että rekisteröinnin ja/tai sisäänkirjautumisen keventämisellä tai muuttamisella voisi saada palvelun kestäämään enemmän yhtäaikaista käyttäjiä.

Voidaan myös havaita ettei prosessorien rasituksessa ole juuri näkyvää muutosta kun lisätään API-koneiden määrä kahdesta kolmeen, näkyvä muutos tapahtuu vasta neljännen koneen lisäyksen jälkeen. Muutokset kuitenkin näkyvät palvelun nopeudessa, joka ei sekään neljällä API-koneella ole hyvä.

Yhteenveto

Kuviossa 41 näkyvän porrasmaisen laskun, API-koneiden määrän noustessa kuviteltiin tapahtuvan kaikilla käyttäjämäärillä. Yllätyksenä tuli käyttäjämäärän noustessa, ettei samanlaista porrasmaista laskua tapahtunutkaan. Kuviossa 45 näkyy, että kaikkien kolmen onnistuneen testin palkit osuvat välillä 85 – 100 % väliin ilman porrasmaista laskua.

Tulosten pohjalta on lähestulkoon mahdotonta laskea, montako konetta tarvittaisiin, jotta palvelu kestäisi 10000 käyttäjää. On kuitenkin oleellista muistaa, että nämä testit simuloivat tehokäyttäjiä, jotka tekivät asioita paljon nopeammin kuin mitä normaali käyttäjä pystyisi tekemään. Yksi API-kone pystyi käsittelemään 23,8 pyyntöä sekunnissa, sen prosessorin käytön ollessa 100 prosenttia, eli palvelu ilman kuormanjakamista kesti alle 24 yhtäaikaista käyttäjää. Kuormanjakamisella saatiin palvelu kestäämään 187,5 yhtäaikaista käyttäjää, jakamalla palvelu neljälle API-koneelle.

8.10 Mittaus 6: Rauhallisten käyttäjien simulointitestit

8.10.1 Suunnittelu

Jotta pystytään tarkemmin näkemään palvelun ongelmakohtia, sekä kuinka palvelu toimisi normaalimmalla käytöllä, täytyy palvelua testata rauhallisemmilla käyttäjillä, jotka simuloivat paremmin normaalia käyttäjää. Kyseistä testiä joudutaan ajamaan hieman pidempään edellisiin testeihin verrattuna, koska kestää kauemmin tehdä samat asiat kuin aiemmin. Palvelun oletetaan kestävän enemmän käyttäjiä, kun pyyntöjä tulee rauhallisemmin.

8.10.2 Toteutus

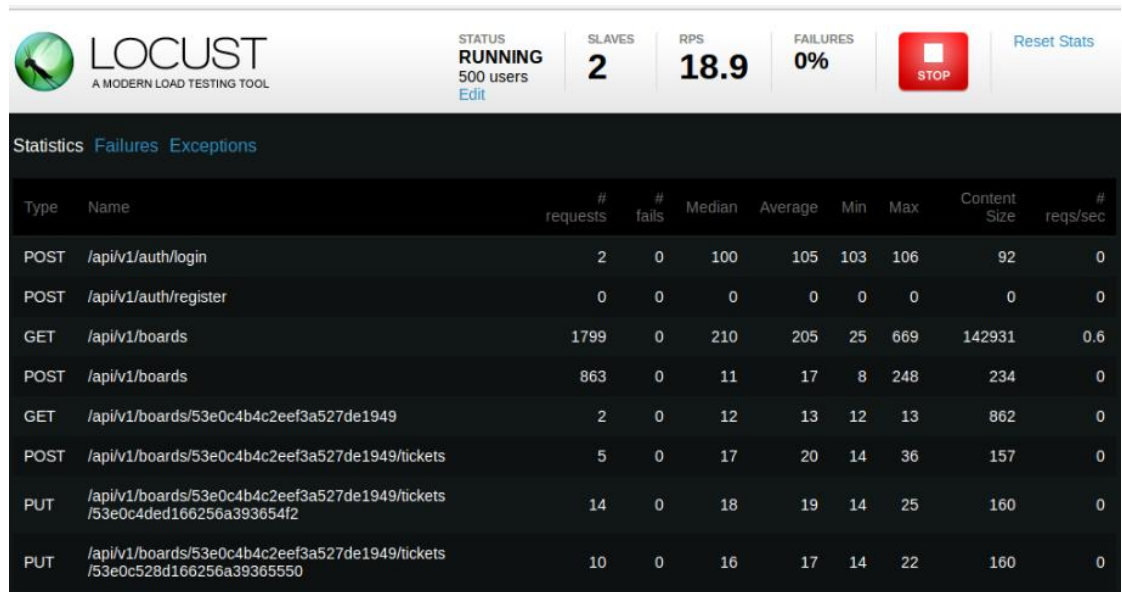
Muokataan aiemmin luotua testikäyttäjää. Asetetaan tehtävien minimi odotus aika olemaan 10 sekuntia sekä maksimi odotusaika 15 sekuntia locust.py tiedostossa:

```
task_set = TeamboardTasks  
min_wait = 10000  
max_wait = 15000
```

Näin saamme luotua rauhallisemman käyttäjä simulaation. Kyseisellä käyttäjällä suoritetaan neljälle API-koneelle testit 500, 750 ja 1000 käyttäjällä.

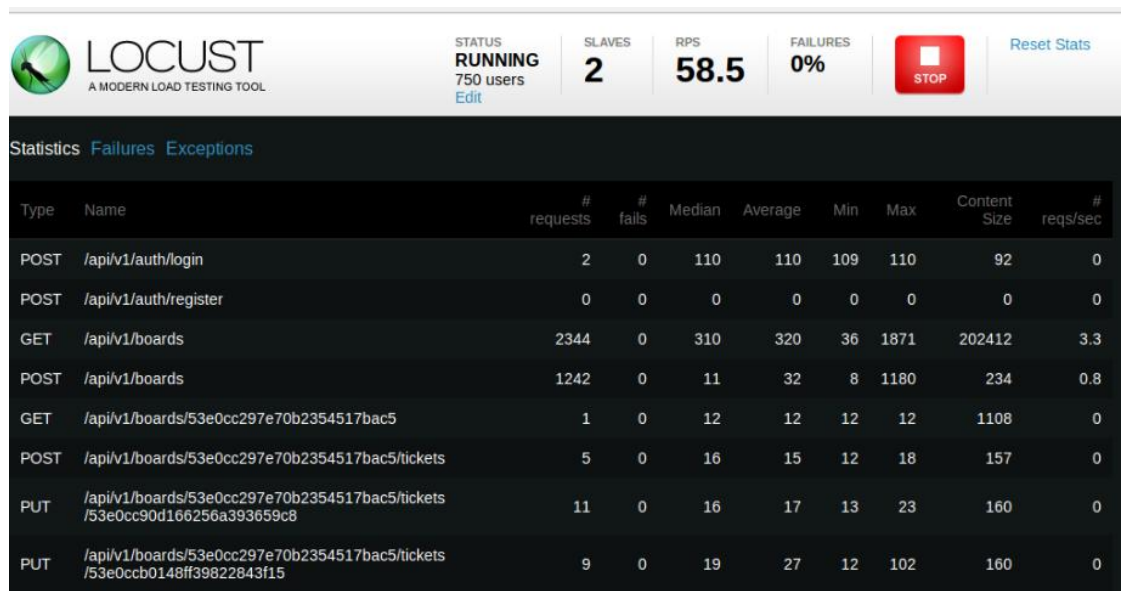
8.10.3 Tulokset

Kuviossa 52 näkyy, kuinka neljä API-konetta kestää 500 rauhallista käyttäjää. Prosessorin rasitus 500 käyttäjälle on kolmantena oikealta kuviossa 55.



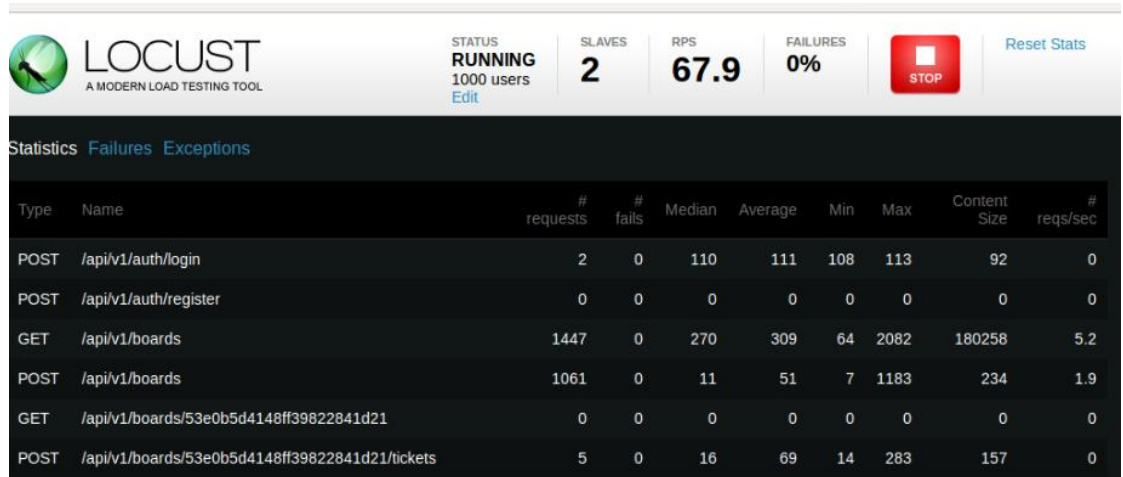
Kuvio 52. 500 rauhallista käyttäjää, 4 API-konetta

Kuviossa 52 näkyy kuinka neljä API-konetta kestää 750 rauhallista käyttäjää. Prosessorin rasitus 750 käyttäjälle on toisena oikealta kuviossa 55.



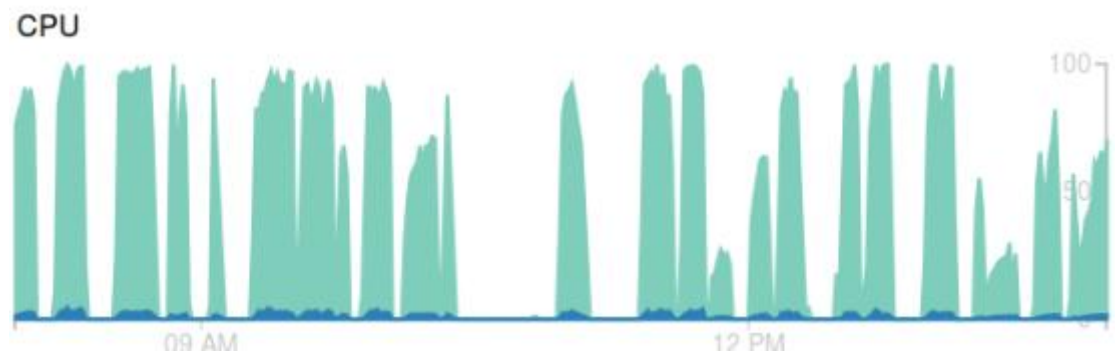
Kuvio 53. 750 rauhallista käyttäjää, 4 API-konetta

Kuviossa 54 nähdään kuinka neljä API-konetta kestää 1000 rauhallista käyttäjää. Prosessorin rasitus 500 käyttäjälle on viimeisenä kuviossa 55.



Kuvio 54. 1000 rauhallista käyttäjää, 4 API-konetta

Kuviossa 55 viimeiset kolme palkkia vasemmalta oikealle järjestyksessä 500, 750 ja 1000 käyttäjää.



Kuvio 55. Kaikkien rauhallisten käyttäjätestien aiheuttamat kuormitukset

8.10.4 Analysointi

Kuvioissa 52, 53 ja 54 voidaan huomata pyyntöjä sekunnissa -määrän olevan pienempi kuin edellisessä mittauksessa. Tämä johtuu siitä, että lisäsimme viivettä pyynnöille. Rekisteröinti -kohdassa pyyntöjen määränä näkyy 0, tämä johtuu siitä, että Locustille kyseiset pyynnöt määriteltiin "on_start", eli alussa suoritettaviksi. Näissä testeissä tehtävien odotusaika on niin suuri, että Locust ehtii hoitaa kyseiset toiminnot ennen kuin aloittaa mitään muita toimintoja. Kuvioista 55 tutkimalla viimeistä kolmea palkkia huomataan, että ensin tulee piikki rasituksessa, tämän oletettavasti aiheuttaa rekisteröinti ja sisäänkirjautuminen. Sitten prosessorin rasitus laskee, koska tauluja ei vielä ole. Tämän jälkeen nousua tapahtuu hiljalleen, oletettavasti koska tauluja tulee koko ajan lisää. Ongelmaksi pitempien testien tekemisessä muodostui ssh-yhteyden katkeileminen Locust-koneisiin, jolloin testi keskeytyi. Olisi ollut kiinnostavaa nähdä mihin asti rasituksen nouseminen olisi jatkunut ja olisiko se lopulta aiheuttanut virhetilanteita. Testitulokset saattavat kuitenkin antaa suuntaa kuinka API-koneen kuormaa voidaan keventää, sillä kuvioista 55 nähdään kuinka alussa suoritettavat tehtävät ja sitten muut tehtävät rasittavat prosessoria eri aikoina suoritettuina. Lyhyemmällä odotusajalla kuten mittauksessa 5, nämä asiat tapahtuvat yhtä aikaa aiheuttaen enemmän hetkellistä kuormitusta. Tästä syystä yksi ratkaisu useamman yhtäaikaisen käyttäjän kestämiseen olisi erottaa autentikointi erilliselle palvelimelle. Jos kuviossa 55 jälkimmäisen hitaasti nousevan kuormituksen aiheuttaa todellakin taulujen määrä, voisi ratkaisuna olla palvelun muuttaminen niin, ettei missään vaiheessa haeta kaikkia luotuja tauluja kerralla.

8.11 Mittaus 7: Tehokäyttäjien simulointi yksityisillä tauluilla

8.11.1 Suunnittelu

Edellisten mittausten perusteella oletetaan, että vähentämällä ladattavien taulujen määrää, voitaisiin vähentää prosessorin raskautta. Teoriassa voidaan pohtia jos 250 käyttäjää luo 2 taulua, on tauluja yhteensä 500. Kun sitten kaikki 250 käyttäjää lataa 500 taulua, tapahtuu 125000 taulun lataamista. Jos käyttäjämäärä kaksinkertaistuu, tauluja luodaan 1000 kappaletta. Kun kaikki 500 käyttäjää lataavat sitten 1000 taulua, tapahtuu 500000 taulun lataamista. Toisin sanoen käyttäjämäärän kaksinkertaistuminen aiheuttaa nelinkertaisen tauluista johtuvan kuormituksen. Seuraavassa testissä testataan saadaanko raskautta laskettua vähentämällä ladattavien taulujen määrää. Mittaustuloksia vertaillaan Mittauksessa 5 saatuihin tuloksiin.

8.11.2 Toteutus

API:ssa oleva "Get Boards"-komento lataa vain kaikki julkiset taulut. Joten uusitaan mittaus 5, mutta muutetaan locust.py tiedostoa niin, että post_board kohdassa lukee:

'isPublic': 'false'

Tällä asetuksella taulut luodaan privaateiksi, jolloin ne ovat olemassa, mutta muut käyttäjät eivät lataa niitä.

8.11.3 Tulokset

Testien tulokset ovat koottuna taulukkoon 4, 5 ja 6. Testien tarkemman tulokset ovat liitteessä 5.

Taulukko 4. 250 käyttäjää privaateilla tauluilla

250 käyttäjää privaateilla tauluilla				
Apien määrä	1	2	3	4
Kestikö kuorman	Kyllä	Kyllä	Kyllä	Kyllä
Prossessorin käyttö arvio	85–95%	47–54%	38 %	27–36%
Pyyntöjen määrä sekunnissa	122,2	120,2	122,8	119,2
Login latenssin keskiarvo ms	11857	1462	206	104

Taulukko 5. 500 käyttäjää privaateilla tauluilla

500 käyttäjää privaateilla tauluilla				
Apien määrä	1	2	3	4
Kestikö kuorman	Ei	Kyllä	Kyllä	Kyllä
Prossessorin käyttö arvio	-	75%	70%	51%
Pyyntöjen määrä sekunnissa	-	243,2	239,2	246,8
Login latenssin keskiarvo ms	-	3539	499	302

Taulukko 6. 750 käyttäjää privaateilla tauluilla

750 käyttäjää privaateilla tauluilla				
Apien määrä	1	2	3	4
Kestikö kuorman	Ei	Kyllä	Kyllä	Kyllä
Prossessorin käyttö arvio	-	98 %	90 %	70-75%
Pyyntöjen määrä sekunnissa	-	281,4	376,3	320,2
Login latenssin keskiarvo ms	-	6611	1295	1126

8.11.4 Analysointi

250 käyttäjää

Verrattaessa edelliseen mittauksiin, huomataan yhden API-koneen pystyvän käsittelemään melkein sata pyyntöä sekunnissa enemmän kuin edeltävässä. Tämä kertoo siitä, että muuttamalla taulukoiden hakumenetelmiä, saadaan Teamboard ilman kuormantasaustakin kestämään 100 yhtäaikaista käyttäjää enemmän. Prosessorin rasitukset ja viiveet laskivat jokaisessa tapauksessa edelliseen verrattuna.

500 käyttäjää

Kaksi API-konetta kesti yli 200 pyyntöä sekunnissa enemmän edelliseen mittaukseen verrattuna. Pyyntö sekunnissa -määrä on noin 240, prosessorin rasituksen ollessa reippaasti alle sadassa prosentissa, joka kertoo siitä, että pystyttäisiin käsittelemään enemmän yhtäaikaista käyttäjiä, mutta 500 käyttäjän testi ei luo niitä enempää. Sama tilanne toistuu muillakin API-kone määrillä.

750 käyttäjää

Kaksi API-konetta kesti 281,4 pyyntöä sekunnissa, oletettavasti tämä on kahden koneen maksimi, koska muilla API-konemäärillä pyyntöjä oli enemmän. Verratessa edelliseen mittaukseen kaksi API-konetta kesti nyt noin 160 pyyntöä sekunnissa enemmän kuin edellisessä. Kolme API-konetta kesti 376,3 pyyntöä sekunnissa, prosessorin rasituksen ollessa noin 90-prosenttia, mistä voidaan päätellä määrään olevan hieman nostettavissa. Neljän API-koneen tulos on hämmäntävä, jostain syystä palvelu pystyi käsittelemään vain 320 pyyntöä sekunnissa, vaikka prosessorin käyttö oli 70 ja 75 prosentin välissä.

Yleistä

Verrattaessa mittaukseen 5 havaitaan, että luomatta julkisia tauluja kuormitetaan jokaisessa tapauksessa vähemmän prosessoria, saadaan latenssi laskemaan ja pyyntöjen määrä sekunnissa kasvamaan. Nämä seikat vaikuttivat siihen, että esimerkiksi kaksi API-konetta kesti 750 käyttäjän kuorman toisin kuin mittauksessa 5. On myös huomattava, että "Get /api/boards" kohdan sisällön koko "Content Size" on tässä mittauksessa joka kohdassa huomattavasti pienempi kuin mittauksessa 5, mikä tarkoittaa sitä, että verkkoa kuormitetaan paljon vähemmän. Kuvioissa käy ilmi myös kaikkien muiden toimintojen latenssin olevan huomattavasti pienempiä kuin login- ja register -toimintojen, josta voidaan edelleen päätellä kirjautumisen ja rekisteröinnin olevan hitaimmat toiminnot. Tulosten perusteella voidaan sanoa, että kolme API-konetta riittäisi kestäämään 750 käyttäjää, muiden pyyntöjen latenssit ovat erittäin matalia eli sisäänkirjautumisessa saattaa kestää hetki, mutta sitten palvelu toimii hyvin. Jos sisäänkirjautuminen saataisiin nopeammaksi tai eroteltua erilliselle palvelimelle, käyttäjämäärää oletettavasti pystyisi nostamaan.

Tutkimalla mittauksen taulukoita voidaan havaita, että prosessorin käyttö kahdella API-koneella 250 käyttäjällä on lähellä samaa kuin neljällä API-koneella 500 käyttäjällä. Sama ilmiö tapahtuu kahden API-koneen 500 käyttäjän ja neljän API-koneen 750 käyttäjän kesken. Tästä ilmiöstä voisi päätellä, että tarvitsee lisätä kaksi API-konetta jokaista 250 käyttäjää kohden, mikäli prosessorin kuormitus halutaan pitää samana. Yhtäaikaisten käyttäjien, eli pyyntöjä sekunnissa -määriä tutkimalla voidaan todeta, että yksi API-kone kestää 122, kaksi API-konetta kestää 281 ja kolme kestää ainakin 376. Tutkimalla HAProxyn rasiusta testien ajalta voidaan todeta, ettei se tule olemaan ongelma näillä kuormamäärillä.

8.12 Mittaus 8: Brypt-palvelin erotettuna

8.12.1 Suunnittelu

Aiemmista tuloksista pystyi päättämään, että rekisteröinti ja kirjautuminen rasittavat API-koneita paljon, tästä syystä erotellaan ne omalle palvelimelle. Ympäristö on muuten sama kuin mittauksesta 5 alkaen tehdyissä mittauksissa, mutta API-koneista on eroteltu yhdeksi omaksi palvelimeksi Bcrypt-palvelin. Bcrypt-palvelin on yhteydessä ainoastaan API-koneisiin ja API-koneet lähettävät sen suoritettavaksi kaikki salaukseen liittyvät toiminnot. Oletuksena on, että kävijämäärän kaksinkertaistaminen kaksinkertaistaa API-koneiden kuormituksen ja koneiden määrän kaksinkertaistaminen puolittaa kuormituksen.

8.12.2 Toteutus

Suoritetaan testejä privaattitauluja luovilla tehokäyttäjillä, jotta nähdään oliko hyödyllistä erottaa Brypt-palvelin erilleen API-koneista. Mittauksia suoritetaan eri käyttäjä- ja konemäärillä, jotta voidaan havainnollistaa kuinka eri muutokset vaikuttavat palvelun toimintaan.

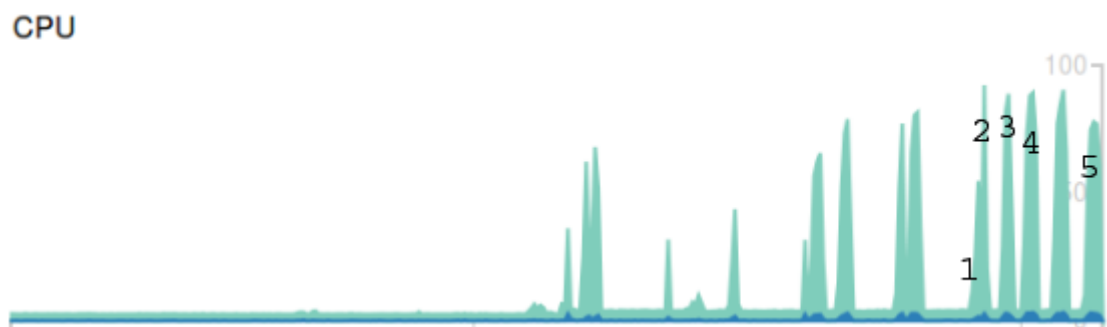
8.12.3 Tulokset

Mittauksen tulokset ovat koottuina taulukossa 7. Testien tarkemman tulokset ovat liitteessä 6.

Taulukko 7. Tulokset Bcryptin ollessa erillisenä palveluna.

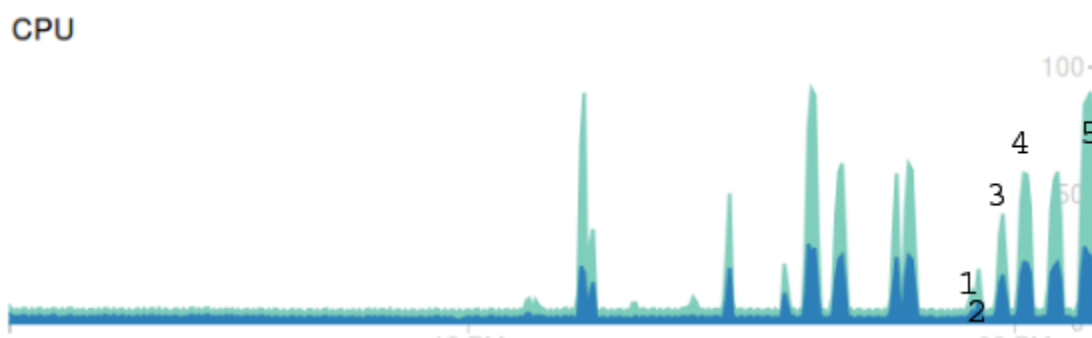
Bcrypt Erillään					
Käyttäjien määrä	250	500	1000	1000	1250
API-koneiden määrä	1	1	2	3	4
Kestikö kuorman	Kyllä	Kyllä	Kyllä	Kyllä	Kyllä
Proessorin käyttö arvio	52 %	96 %	95 %	95 %	78 %
Pyyntöjen määrä sekunnissa	74,3	182	251,4	370,5	394,3

Kuviossa 56 näkyy kaikkien testien aiheuttama rasitus API-koneelle. Kohdassa 1 on 250 käyttäjää yhdelle API-koneelle. Kohdassa 2 on 500 käyttäjää yhdelle API-koneelle. Kohdassa 3 on 1000 käyttäjää kahdelle API-koneelle. Kohdassa 4 on 1000 käyttäjää kolmelle API-koneelle. Kohdassa 5 on 1250 käyttäjää neljälle API-koneelle.



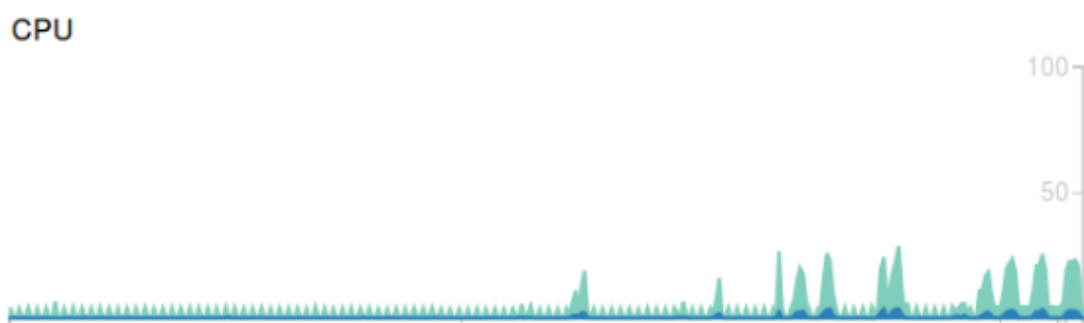
Kuvio 56. Testien aiheuttamat rasitukset API-koneen prosessorille

Kuviossa 57 näkyy kaikkien testien aiheuttama rasitus Shard-koneelle. Kohdassa 1 on 250 käyttäjää yhdelle API-koneelle. Kohdassa 2 on 500 käyttäjää yhdelle API-koneelle. Kohdassa 3 on 1000 käyttäjää kahdelle API-koneelle. Kohdassa 4 on 1000 käyttäjää kolmelle API-koneelle. Kohdassa 5 on 1250 käyttäjää neljälle API-koneelle.



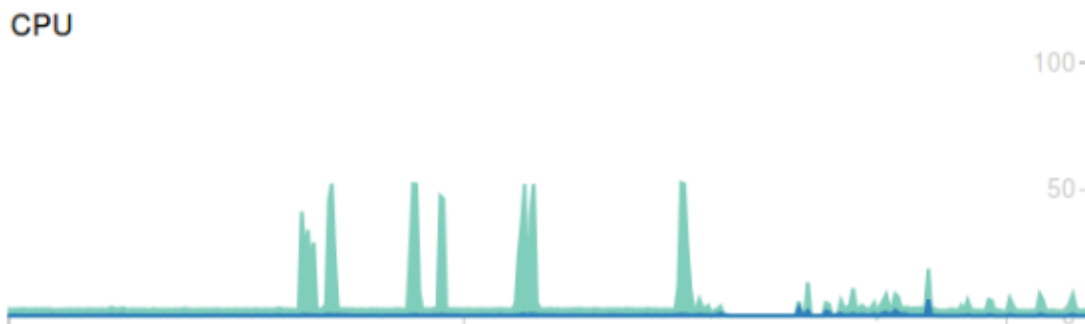
Kuvio 57. Testien aiheuttamat rasitukset Shard-koneen prosessorille

Kuviossa 58 näkyy kaikkien testien aiheuttama rasitus HAProxy-koneelle. Kohdassa 1 on 250 käyttäjää yhdelle API-koneelle. Kohdassa 2 on 500 käyttäjää yhdelle API-koneelle. Kohdassa 3 on 1000 käyttäjää kahdelle API-koneelle. Kohdassa 4 on 1000 käyttäjää kolmelle API-koneelle. Kohdassa 5 on 1250 käyttäjää neljälle API-koneelle.



Kuvio 58. Testien aiheuttamat rasitukset HAProxy-koneen prosessorille

Kuviosta 59 käy ilmi aiempien testien aiheuttamat rasitukset Brypt-koneen prosessorille. Viimeisen kolmanneksen palkit ovat näistä mittauksista.



Kuvio 59. Testien aiheuttamat rasitukset Bcrypt-koneen prosessorille

8.12.4 Analysointi

Kuviosta 56 kohdista 1 ja 2 voidaan havaita, että käyttäjien määrän kaksinkertaistuksessa se ei edes kaksinkertaista prosessorin kuormitusta. Vertailemalla saman kuvion kohtia 2 ja 3 voidaan havaita, että kaksinkertaistamalla sekä kävijämäärä että API-koneiden määrä, prosessorin rasitus jopa pieneni. Tuloksista voidaan päätellä tietokanta-koneen rasituksen nousevan samalla kun pyyntöjä sekunnissa -määrä nousee. Tämä on selitettävissä sillä, että mitä tiuhempaa tahtia pyyntöjä tulee, sitä enemmän tietokannalla on tehtävää. Tulosten perusteella voidaan todeta, että neljä API-konetta kestää 1250 käyttäjää sekä 394,3 pyyntöä sekunnissa, sen prosessorin rasituksen ollessa 78 %. Palvelun toimintaa testattiin myös selaimella testin aikana ja palvelu vastasi viiveettä pyyntöihin.

Kuormantasaajan prosessorin rasitus nousi noin 20 prosenttiin, joten kuormantasaaja ei tule olemaan ongelma vielä hetkeen. Bcrypt-koneen rasitus oli todella matalaa. Tämä oletettavasti johtui siitä, että API-koneesta erottamisen yhteydessä Bcryptiä myös kevennettiin. Tietokanta-koneen prosessorin rasituksen nousu alkaa muodostua ongelmaksi, sillä 4 API-koneelle yritettiin tehdä 1500-käyttäjän testiä, mutta se tuotti paljon virheitä. Tarkemmat tutkimukset osoittivat virheiden johtuvan Shard-koneen prosessorin kuormituksen olemisesta 100 prosentissa.

Pyyntöjä sekunnissa -määrien perusteella voidaan todeta, että Bcrypt palvelun ollessa erotettuna yksi API-kone kestää 182 yhtäaikaista käyttäjää. Tasaamalla kuorma kahdelle API-koneelle, kestää palvelu 251 yhtäaikaista käyttäjää. Kolmelle API-koneelle tasaamalla kestää palvelu 370 yhtäaikaista käyttäjää. Neljälle tasaamalla keitetään ainakin 394 yhtäaikaista käyttäjää, mutta enempää ei voida testata tällä hetkellä koska tietokannan rasitus on liian suuri.

8.13 Mittaus 9: Shardien lisääminen tietokantaan

8.13.1 Suunnittelu

Jotta mittauksia pystytään jatkamaan, täytyy tietokantaa skaalata. Tämä tapahtuu lisäämällä uusi MongoShard-kone, jolle aiempi tietokanta jaetaan. Tietokannan jakaminen MongoQuery-palvelimella tapahtuu antamalla käsky:

```
sh.enableSharding("development")
```

Jotta taulujen arvot jakaantuvat mahdollisimman tasaisesti Shard-koneiden kesken, täytyy vielä antaa käskyt:

```
sh.shardCollection( "development.boards", { _id: "hashed" } )  
sh.shardCollection( "development.users", { _id: "hashed" } )
```

Kyseiset käskyt jakavat taulujen "boards" ja "users" tietueet sen perusteella mikä niiden kentän "_id" arvon tarkistearvo on. Ilman tätä käskyä voi käydä niin, että vaikka tietokanta on jaettuna kahdelle koneelle, dataa menee vain yhdelle.


8.13.2 Toteutus

Suoritetaan testejä privaattitauluja luovilla tehokäyttäjillä, jotta nähdään tietokannan skaalaamisen hyöty. Mikäli palvelu ei jumiutu tietokannan prosessorin kuormituksen johdosta, jatketaan mittauksia siitä mihin edellisessä mittauksessa jäätiin. Jotta saadaan selkeä kuva kuinka API-koneiden lisääminen vaikuttaa, testataan kuinka paljon yhtäaikaaisia käyttäjiä palvelu kestää 1750:llä privaattitauluja luovilla tehokäyttäjillä.

8.13.3 Tulokset

Neljä API-konetta

Kuviossa 60 näkyy kuinka neljä API-konetta kestää muutosten jälkeen jopa 1500 käyttäjää.



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

1500 users

Edit

SLAVES

3

RPS

561.4

FAILURES

0%

STOP

[Reset Stats](#)

Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	1	0	160	157	157	157	92	0
POST	/api/v1/auth/register	0	0	0	0	0	0	0	0
GET	/api/v1/boards	4610	0	110	1007	7	11051	344	32.5
POST	/api/v1/boards	3102	0	52	333	6	4949	158	11.2
GET	/api/v1/boards/53f1eec037e599ed49459b13	1	0	800	796	796	796	714	0
POST	/api/v1/boards/53f1eec037e599ed49459b13/tickets	0	0	0	0	0	0	0	0
PUT	/api/v1/boards/53f1eec037e599ed49459b13/tickets/53f1eec837e599ed49459b6e	9	0	120	211	19	672	162	0.1
PUT	/api/v1/boards/53f1eec037e599ed49459b13/tickets/53f1eed4b6efbae510ee3ece	9	0	61	1142	14	4835	162	0.1

Kuvio 60. Neljä API-konetta, 1500 käyttäjää.

Jotta voidaan olla varmoja siitä, että tietokanta jakautuu useammalle koneelle, tarkastetaan Shard-koneiden prosessorien rasitukset. Kuviossa 61 näkyy viimeisessä palkissa Shard1-koneen prosessorin rasitus.



Kuvio 61. Shard1-koneen prosessorin rasitus

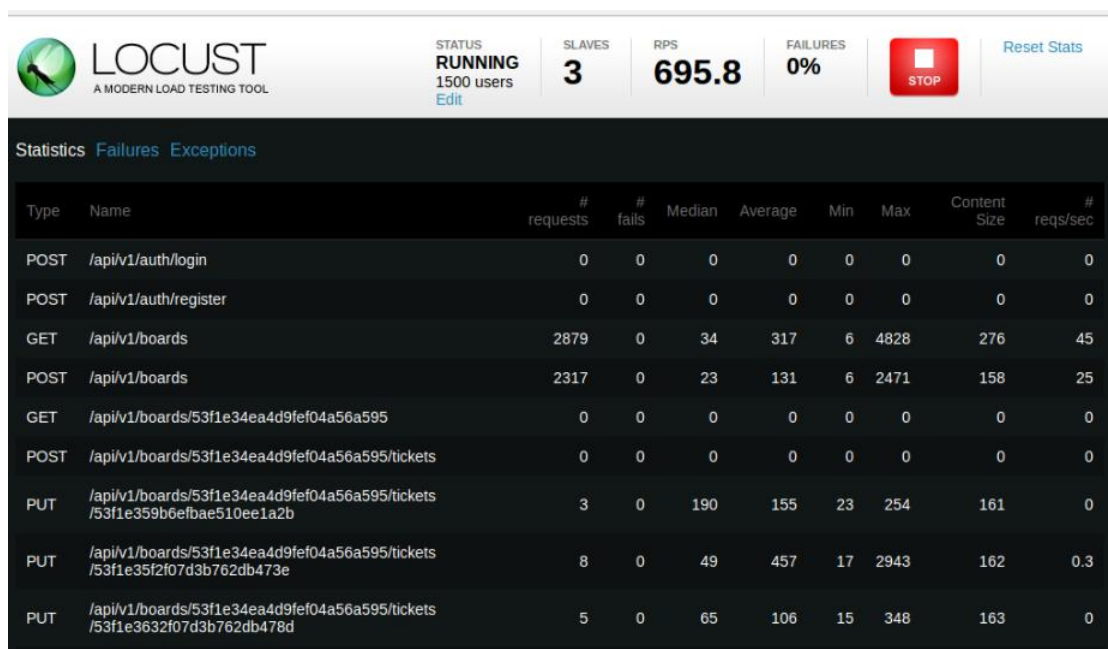
Kuviossa 62 näkyy kuinka myös Shard2-koneen prosessori rasittuu samaan aikaan kuin Shard1-koneen prosessori.



Kuvio 62. Shard2-koneen prosessorin rasitus

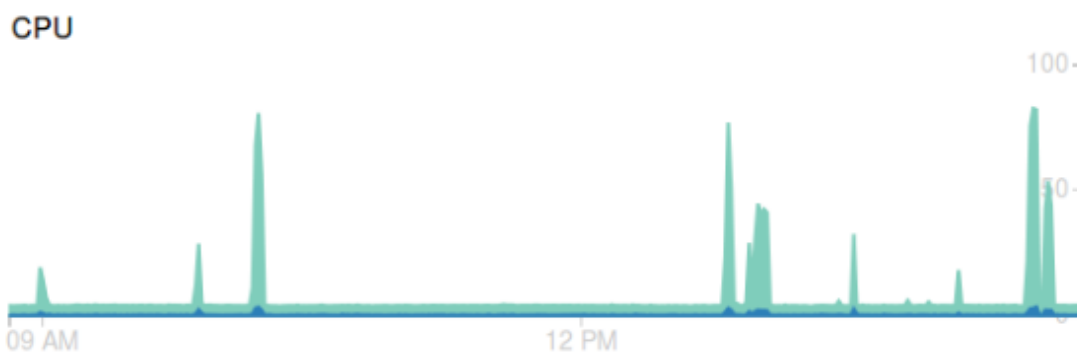
Viisi API-konetta

Kuviossa 63 näkyy kuinka viisi API-konetta 1500 käyttäjällä pystyy käsittelemään enemmän yhtäaikaista käyttäjiä kuin neljä API-konetta.



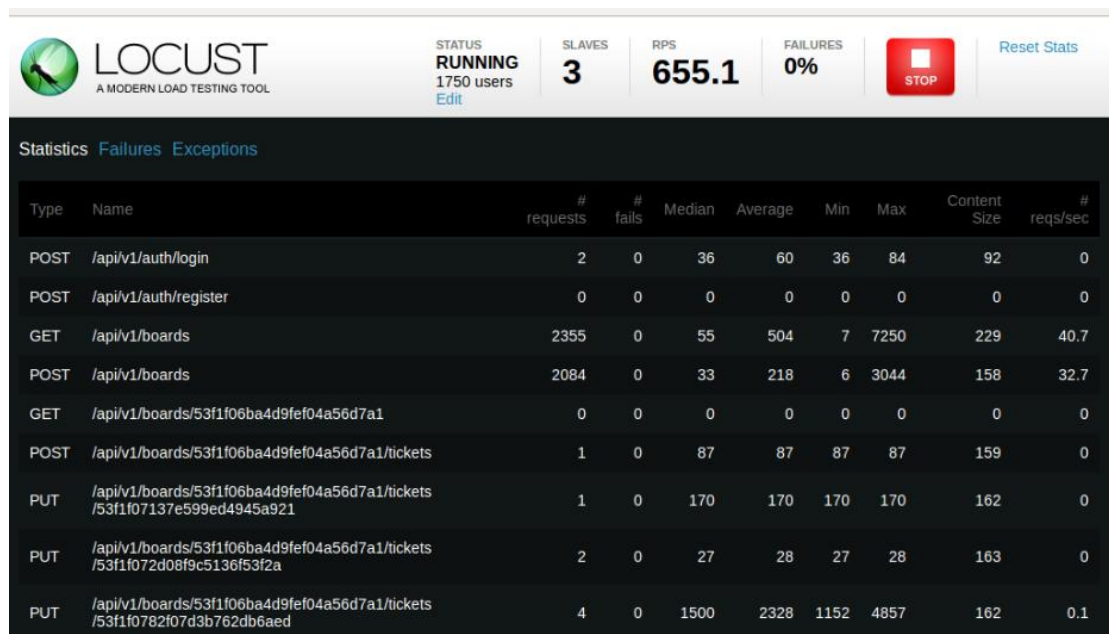
Kuvio 63. Viisi API-konetta, 1500 käyttäjää

Kuviossa 64 nähdään viimeisessä palkissa API-koneen prosessorin rasitus 1500 käyttäjällä kuorman jakaantuessa viidelle API-koneelle. Sitä edeltävässä palkissa sama määrä käyttäjiä, mutta kuorma jakaantuu neljälle API-koneelle.



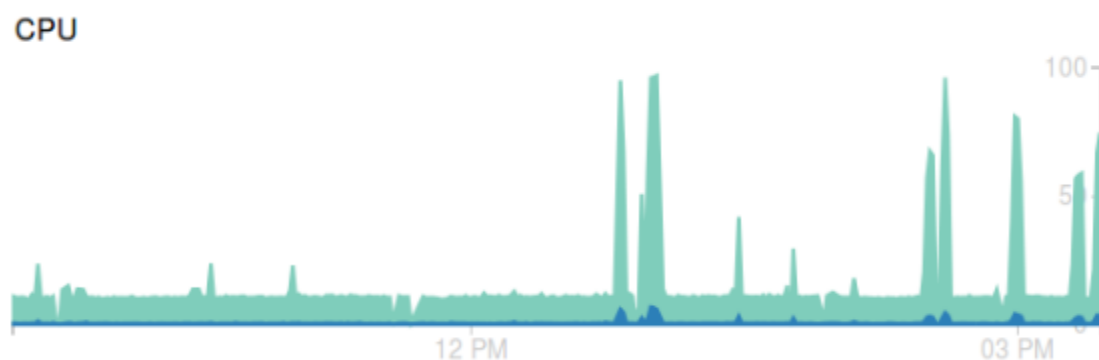
Kuvio 64. 1500 käyttäjän aiheuttama rasitus prosessorille viidellä ja neljällä API-koneella

Seuravaksi testattiin kuinka viisi API-konetta kestää 1750 käyttäjää. Kuviossa 65 näkyy pyyntöjä sekunnissa -määrän olevan pienempi kuin 1500 käyttäjällä. Tämä herätti epäilyksiä siitä, että jossain on oltava vikaa.



Kuvio 65. Viisi API-konetta, 1750 käyttäjää

Testin jatkuttua hetken aikaa alkoi testissä tulla virheitä. Kuviossa 66 näkyy, ettei syy ole Shard1-koneen prosessorin rasituksessa.



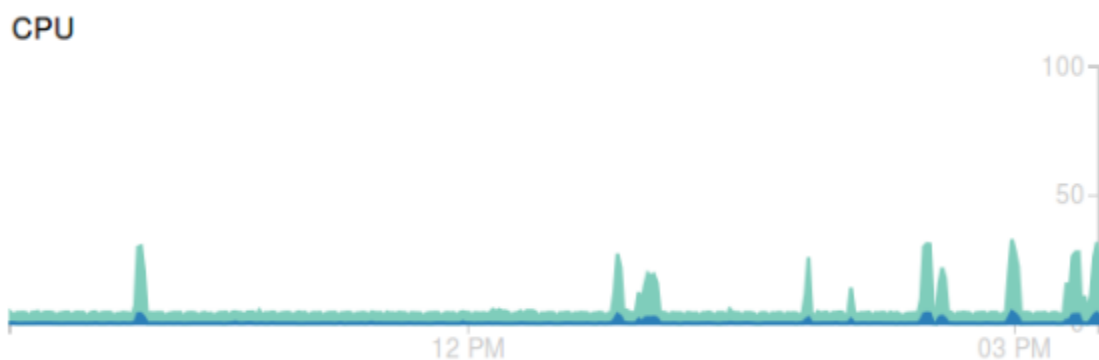
Kuvio 66. Shard1-koneen prosessorin rasitus 1750 käyttäjällä

Myös Shard2 koneen prosessorin rasitus on kunnossa, kuten kuviosta 67 käy ilmi.



Kuvio 67. Shard1-koneen prosessorin rasitus 1750 käyttäjällä

HAProxy-koneen prosessorin rasitus ei voi aiheuttaa vikatilannetta kuten kuviosta 68 käy ilmi.



Kuvio 68. HAProxy-koneen prosessorin rasitus 1750 käyttäjällä

Kuviossa 69 näkyy, ettei API1-koneen prosessorin rasitus ole vielä sadassa prosentissa, joten sekään ei aiheuta vikaa



Kuvio 69. API1-koneen prosessorin rasitus 1750 käyttäjällä

Lopulta vian syy löytyi HAProxyn statistiikkasivuilta, HAProxy ei enää jaa kuormaa tasaisesti, kuten se aiemmin jakoi. Yhteyksien jakautuminen statistiikkasivuilla näkyy kuviossa 70.

API												
	Queue			Session rate			Sessions					
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last
API1	0	0	-	0	180		0	367	2048	41 788	41 788	3m48s
API2	0	0	-	0	180		0	929	2048	41 788	41 788	3m48s
API3	0	0	-	0	180		0	351	2048	41 788	41 788	3m48s
API4	0	0	-	0	180		0	301	2048	41 788	41 788	3m48s
API5	0	0	-	0	180		0	445	2048	41 788	41 788	3m48s
Backend	0	0		0	900		0	1 551	4 096	208 940	208 940	3m48s

Kuvio 70. HAProxyn statistiikkasivu, kuorman jakaantumisen ongelma

8.13.4 Analysointi

Tietokannan jakaminen kahdelle Shard-koneelle auttoi palvelua kestämään enemmän yhtäaikaista käyttäjiä, sillä nyt neljä API-konetta pystyi käsittelemään 1500 käyttäjää, yhtäaikaisten käyttäjien määrän ollessa 564,1.

Yhtäaikaisten käyttäjien määrän noustessa ilmeni uusi ongelma ja tällä kertaa vika oli kuormantasaajassa. Alussa suoritetuissa mittauksissa totuttiin siihen,

että API-koneiden prosessorin kuormitus on sama kaikissa API-koneissa sekä siihen, että kuorma jakautuu tasaisesti kaikkien koneiden kesken. Tuntemattomasta syystä Round Robin-menetelmää käyttäessä API2-koneelle tuli noin kolme kertaa enemmän yhtäaikaista istuntoja, kuin muille koneille. Mittaukset toistettiin useita kertoja virheen aina toistuessa ja aina API2-koneelle. Vaikka API2-kone käynnistettiin uudelleen, sama ongelma jatkui. Vian paikallistamiseksi ja korjaamiseksi päätettiin tehdä Mittaus10.

8.14 Mittaus 10: Kuorman jakaminen tasaisesti

8.14.1 Suunnittelu

Mittauksessa 9 ongelmaksi osoittautui, että API2-koneelle kertyy yli kolme kertaa enemmän yhtäaikaista istuntoja kuin muille API-koneille. Tästä syystä HAProxyn asetuksia muutetaan niin, että se käyttää vähiten yhteyksiä -menetelmää kuormantasaamiseksi. Näin voidaan tutkia tasoittuuko kuorman jakaantuminen, ja samalla nähdään kuinka se vaikuttaa palvelun toimintaan.

8.14.2 Toteutus


HAProxy muutetaan käyttämään vähiten yhteyksiä -menetelmää, vaihtamalla sen konfiguraatio tiedostoon kohtaan "balance roundrobin" teksti "balance leastcon". Tämän jälkeen HAProxy-koneelle annetaan käsky:

```
service haproxy restart
```

Kyseisellä käskyllä HAProxyyn pystytään lataamaan uudet asetukset konfiguraatio tiedostosta, käynnistämättä sitä uudelleen. Tämän jälkeen mittaukset suoritetaan viidelle koneelle 1750:llä privaatti tauluja luovilla tehokäyttäjillä.

8.14.3 Tulokset

Kuviossa 71 näkyy kuinka palvelu kestää 1750 käyttäjää, kun kuorma on tasattuna vähiten yhteyksiä -menetelmällä.



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

1750 users

Edit

SLAVES

3

RPS

720.7

FAILURES

0%

STOP

Reset Stats

Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	0	0	0	0	0	0	0	0
POST	/api/v1/auth/register	0	0	0	0	0	0	0	0
GET	/api/v1/boards	2416	0	60	216	8	2070	243	46.8
POST	/api/v1/boards	2124	0	32	108	7	1573	158	33.1
GET	/api/v1/boards/53f20c28b6efbae510ee7e10	3	0	21	265	9	766	874	0
POST	/api/v1/boards/53f20c28b6efbae510ee7e10/tickets	4	0	15	33	11	64	159	0
PUT	/api/v1/boards/53f20c28b6efbae510ee7e10/tickets/53f20c36b6efbae510ee7eda	7	0	31	226	14	1329	162	0
PUT	/api/v1/boards/53f20c28b6efbae510ee7e10/tickets/53f20c4cb6efbae510ee8115	3	0	500	387	82	584	163	0.2
PUT	/api/v1/boards/53f20c28b6efbae510ee7e10/tickets/53f20c55b6efbae510ee8205	6	0	30	243	21	681	162	0.1

Kuvio 71. Viisi API-konetta, 1750 käyttäjää, vähiten yhteyksiä -menetelmä

Kuviossa 72 nähdään HAProxyn statistiikasta kuinka maksimi istuntojen määrät ovat nyt jakautuneet tasaisesti.

API												
	Queue			Session rate			Sessions					
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last
API1	0	0	-	0	199		0	222	2048	23 732	23 732	17s
API2	0	0	-	0	177		0	223	2048	19 118	19 118	17s
API3	0	0	-	0	221		0	223	2048	24 742	24 742	17s
API4	0	0	-	0	220		0	223	2048	24 054	24 054	17s
API5	0	0	-	0	199		0	223	2048	23 214	23 214	17s
Backend	0	0		0	915		0	1 113	4 096	114 860	114 860	17s

Kuvio 72. HAProxyn statistiikkasivu, maksimi istuntojen jakautuminen

8.14.4 Analysointi

HAProxyn statistiikkaa tutkimalla havaitaan, että yhtäaikaisten istuntojen määrä jakaantuu tasaisemmin kun Round Robin –menetelmästä vaihdettiin vähiten yhteyksiä –menetelmään. Samalla pystytään havaitsemaan palvelun kestävän tällöin enemmän yhtäaikaista käyttäjiä.

9 Pohdinta

9.1 Johtopäätökset

Kuormantasaaminen

Kuormantasaaminen pilviympäristössä eroaa oikeilla laitteilla tehtyyn kuormantasaamiseen vain siltä osin, ettei pilviympäristöön voida lisätä fyysisiä kuormantasaajia. Kuormantasauksen asetuksia säädettäessä on hyvä ottaa huomioon, ettei turhaan kierrätä liikennettä pilvipalveluntarjoajan reunakoneen kautta montaa kertaa, mikäli käytössä on useampi verkko. Esimerkiksi DigitalOceanissa HAProxyn asettaminen ohjaamaan liikennettä Teamboard-koneisiin sisäverkon kautta oli huomattavasti parempi ratkaisu kuin ohjata liikenne ulkoverkon kautta. Tämä johtuu siitä että ohjatessa ulkoverkon osoitteen kautta liikenne kierrätetään uudelleen DigitalOceanin reunalaitteen kautta. Tärkeää oli huomata, ettei kuormantasaaja yksin ratkaise kaikkia ongelmia, vaan palvelua on säädettävä niin, että se toimii järkevästi kuormantasaajan kanssa.

DigitalOceanissa pienin virtuaalikone, jolla Teamboard pyörii tällä hetkellä maksaa 5 \$/kk. Pienin kaksitytiminen kone maksaa 20 \$/kk ja neljäytiminen 40 \$/kk. Hintojen pohjalta voidaan laskea, paljonko Teamboardin pyörittäminen maksaisi eri vaihtoehtoilla taulukon 6 mukaisesti olettaen ytimien määrän nostamisen aiheuttavan saman kuin koneiden lisääminen.

Taulukko 8. Hintavertailu ydinten määrän mukaan

Proessorien määrä	Hinta(\$/kk)			Säästö
	Yksi ydin	Kaksi ydintä	Neljä ytimi- sillä	
2	15	25		10
4	25	45	45	20
6	35	65		30
8	45	85	85	40
10	55	105		50
12	65	125	125	60
14	75	145		70
16	85	165	165	80
18	95	185		90
20	105	205	205	100
22	115	225		110
24	125	245	245	120
26	135	265		130
28	145	285	285	140
30	155	305		150
32	165	325	325	160
34	175	345		170
36	185	365	365	180
38	195	385		190
40	205	405	405	200

Kuukausihinta on laskettu yksiytimisille koneille kaavalla:

*Proessorien määrä * virtuaalikoneen hinta + kuormantasaajan hinta (5 \$).*

Kaksiytimisille koneille kaava on muuten sama, mutta prosessorien määrä on jaettu kahdella, koska tuplaydin vastaa kahta prosessoria eli kaava on:

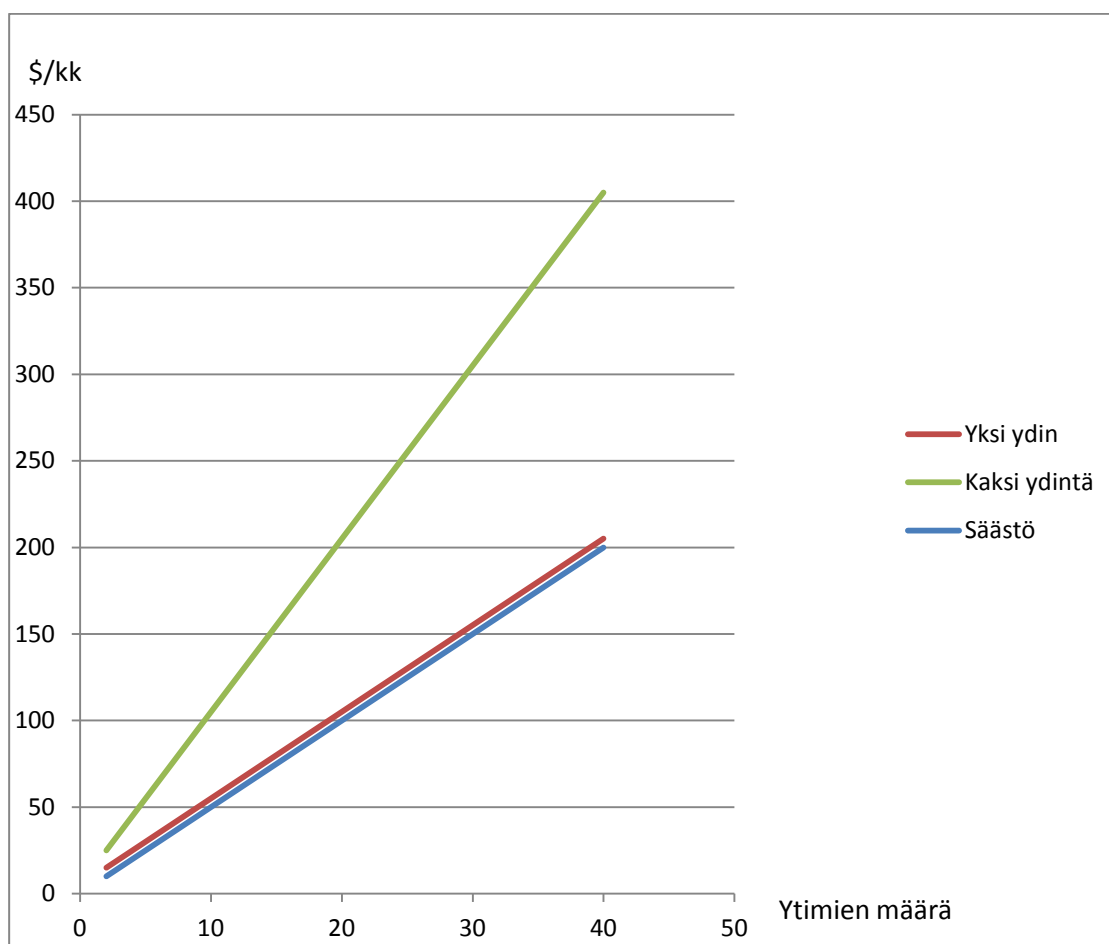
*Proessorien määrä/2 * virtuaalikoneen hinta + kuormantasaajan hinta (5 \$).*

Kaava neljälle samasta syystä:

*Proessorien määrä/4 * virtuaalikoneen hinta + kuormantasaajan hinta (5 \$)*

Lopuksi lasketaan säästö kenttään kuinka paljon edullisemmaksi tulee käyttää yksitytimisiä koneita, kaksitytimisten sijaan kaavalla: Kaksitytimisen hinta – Yksitytimisen hinta.

Taulukon tulosten perusteella voidaan todeta, että on halvempaa lisätä pieniä koneita useampia ja tasata liikenne niille kuormantasaajalla, kuin lisätä koneen ytimiä, varsinkin kun ytimien määrä ei täysin puolita kuormaa. Sama voidaan esittää kuviossa 73 olevan kuvaajan avulla, jossa Y-akselilla on kuukausihinta ja X-akselilla ytimien määrä.



Kuvio 73. Hintavertailu yksi ja kaksitytimisten koneiden välillä

Kuormantestaus

Moni kuormantestaus ohjelmisto oli valmistajan nettisivujen lupauksista huolimatta pettymys. Esimerkiksi LoadUI:n väitettiin olevan tehokas ja skaalautuva, kuitenkin se alkoi pian kaataa käynnistettäessä Linuxin jolla sitä ajettiin. Ohjelman sai toimimaan hetkellisesti oikein asentamalla sen uudelleen, mutta pian se alkoi oireilla samalla tavalla. Tästä syystä se jouduttiin jättämään pois myöhemmistä testeistä.

SoapUI:n ongelmaksi taas muodostui kirjautumisen yhteydessä saatavan poletin tallettaminen muistiin, sillä jos testeissä ei tarvittu polettia tai se asetettiin valmiiksi, ohjelma pystyi suorittamaan useita tuhansia yhteyksiä. Jos ohjelma laitettiin tallentamaan poletti kirjautumisen yhteydessä, niin 800–900 kävijän simulointi kaatoi ohjelmiston muistin loppumisen takia, vaikka testikoneessa oli 16GB muistia. Poletin saaminen talteen on kuitenkin tärkeää, koska Teamboardissa käyttäjä tunnustautuu tällä poletilla.

Parhaaksi tavaksi käyttäjien toimintojen simulointiin osoittautui Locust-ohjelma, jonka avulla pystyi suorittamaan monimutkaisempia toimintoja, simuloimaan useaa yhtä aikaista käyttäjää, sekä luomaan käyttäjiä tarpeeksi. Locust -ohjelmaa voisi parantaa sillä, että siihen voisi suoraan määrittää pyyntöjä sekunnissa -määrän.

9.2 Tulokset

Lähtötilanteessa palvelu kesti yhdellä API-koneella 23,8 yhtäaikaista käyttäjää. Työssä tehtyjen testien perusteella palvelu saatiin kestäämään yhdellä API-koneella 182 yhtäaikaista käyttäjää. Kuormantasaajalla kuorma jaettiin viidelle API-koneelle, jolloin palvelu saatiin kestäämään ainakin 720 yhtäaikaista käyttäjää.

Mittauksien yhteydessä saatiin havaittua useita pullonkauloja Teamboardissa, kuten kaikkien julkisten taulujen lataaminen, rekisteröinnin ja kirjautumisen

hitaus verrattuna muihin Teamboardin toimintoihin. Jakamalla alkuperäisen palvelun komponentit useammalle palvelimelle saatiin lisättyä palvelun kestävyyttä, sekä pystyttiin tutkimaan tarkemmin eri komponenttien kuorman kestävyyttä. Jakaminen mahdollistaa jatkossa ongelmaksi muodostuvien komponenttien skaalaamisen helpommin.

Testien perusteella voidaan arvioida 10000 yhtäaikaisten käyttäjien vaativan parhaassakin tapauksessa vähintään 55 API-konetta palvelun nykyisellä tilalla. Määrä voidaan laskea siitä, että parhaassa tapauksessa yksi API-kone kesti 182 yhtäaikaista käyttäjää ja olettaen että API-koneen lisääminen aina tuplasi käyttäjien määrän. Vaikuttaisi myös että vähintään kuudensadan yhtäaikaisten käyttäjien välein pitäisi lisätä tietokantaan uusi Shard-kone, jotta tietokanta kestäisi kuorman. Otanta on kuitenkin liian pieni siihen, että pystyttäisiin tekemään tarkkoja arvioita kaikkien komponenttien määristä. Mittauksesta 11 voidaan huomata viiden API-koneen kestävän 720 yhtäaikaista käyttäjää, jakamalla käyttäjämäärä API-koneiden määrällä selviää yhden API-koneen kestävän vain 144 yhtäaikaista käyttäjää. Tällä realistisemmalla arvolla 10000 yhtäaikaista käyttäjää vaatisi noin 70 API-konetta, sekä tuntemattoman määrän muita komponentteja.

9.3 Jatkokehitys

Testeissä ilmeni, että tilannekuvien jatkuva tallentaminen poistamatta edellisiä kuvia on sekä raskasta ja se täyttää kiintolevyt. Tästä syystä tulisi suunnitella paremmin kuvien ottamisen tiheys/tarve, sekä entisten kuvien poistaminen, ehkä jopa erotella tilannekuvat omaksi palveluksi. Mittaukset myös osoittivat, että se, ettei kaikkia tauluja ladata kerralla keventäisi palvelua huomattavasti.

Shard-koneita oletettavasti tarvitaan pian, ja onkin erittäin tärkeää seurata tietokannan kuormitusta käyttäjämäärien noustessa.

Mikäli tulevaisuudessa siirrytään käyttämään toista pilvipalvelua, jossa ei maksuteta koneista jotka ovat poissa päältä, rahaa säästettäisiin vaihtamalla kuormantasaus First -menetelmään. First -menetelmän ansioista koneet olisivat päällä vain kun käyttäjiä on niin paljon, että niille on tarve ja heti käyttäjien vähennyttyä voitaisiin taas sulkea ylimääräiset koneet. Tähän voitaisiin hyödyntää tässä työssä saatuja yhtäaikaisten käyttäjien määriä eri API-kone määriin.

Tällä hetkellä kuormantasaaja hoitaa myös sisäverkossa tapahtuvien laitteiden kuormantasaamisen, jatkossa olisi järkevää luoda erillinen kuormantasaaja sisäverkon liikenteelle. Näin saataisiin jo oleva kuormantasaaja kestäämään enemmän liikennettä, koska sen ei tarvitse tasata palvelun komponenttien toisilleen lähettämää liikennettä.

9.4 Saatujen tulosten luotettavuus

Kyseiset tulokset ovat palvelu- ja tilannekohtaisia. Jokainen tulos on riippuvainen Teamboardin sen hetkisestä tilanteesta ja asetuksesta, kuitenkin lähes jokainen mittaus suoritettiin useampaan kertaan, jotta nähtiin ovatko tulokset lähellä toisiaan.

Mittaukset 1–4 suoritettiin koneelta, joka on Jyväskylän Ammattikorkeakoulun verkossa, joten on mahdotonta huomioida kuinka paljon eri verkkolaitteet ja matka DigitalOceaniin vaikuttavat tuloksiin. Mutta toisaalta on mahdotonta myös määritellä millaisilla yhteyksillä käyttäjät käyttävät kyseistä palvelua, siksi mittaukset 5–10 suoritettiin DigitalOceanista käsin, jolloin ulkoverkon vaikutus on eliminoitu.

Tarkoituksena ei kuitenkaan ollut mitata suoraan viivettä, vaan keskittyä kehittämään Teamboardia ja kuormantasaajaa mahdollisimman monen yhtäaikaisten käyttäjän kestämiseksi, sekä määrittämään vaikuttavia tekijöitä. On kuitenkin ymmärrettävä, että vaikka saataisiin tulokseksi Teamboardin kestävän

esimerkiksi 7000 kävijää, keskimääräisen latenssin tällöin ollessa 20000ms, Teamboard kestää kuorman, mutta käyttäjälle palvelun käyttäminen on hidasta. Eli on kaksi täysin eri asiaa kestääkö palvelu kuorman ja onko palvelu järkevästi käytettävissä kyseisellä kuormalla.

Tuloksia katsoessa täytyy myös pohtia mitä tarkoittaa yhtäaikaista käyttäjää, sillä tuloksissa todetaan että palvelu kesti neljällä API-koneella 394,3 yhtäaikaista käyttäjää, joka tässä tapauksessa tarkoittaa käyttäjiä, jotka yhtä aikaa lähettävät pyyntöjä. Kuitenkin Teamboardissa käyttäjä voi olla paikalla, muttei silti lähetä yhtään pyyntöä. Tästä syystä on tärkeää pohtia testin asetuksia ja niissä olleiden käyttäjien määriä. Esimerkiksi kun saavutettiin 393,3 yhtäaikaista käyttäjää, oli paikalla 1250 käyttäjää, jotka lähettivät pyyntöjä 1–2 sekunnin välein, joka on oletettavasti todella paljon useammin kuin mitä oikea käyttäjä tekisi. Oikealla käyttäjällä oletettavasti menee aikaa kun hän kirjoittaa tauluihin ja/tai lappuihin, eikä hän liikuta niitä kerran sekunnissa. Eli kyseisen tuloksen perusteella voidaan puhua joko 393,3 yhtäaikaisesta käyttäjästä tai 1250 tehokäyttäjistä palvelussa.

9.5 Hyöty

Työn hyötyjä olivat useiden pullonkaulojen löytäminen palvelusta ja niiden avulla tehtyjen muutosten tuomat tulokset palvelun kestävyYTEEN. Testien avulla pystyttiin määrittämään eri komponenttien vaikutuksia kuormitukseen sekä tehtyjen muutoksien hyötyjä. Työstä käy ilmi kuinka eri komponentit pystytään asettamaan toimimaan skaalantuvina sekä kuinka kuormaa voidaan tasata käyttäen HAProxya.

Locustilla onnistuttiin simuloimaan todella hyvin oikean käyttäjän toimintoja. Samalla koodilla pystytään demonstroimaan Teamboardin toimintoja, sekä tehdä palvelulle kestävyystestejä, joihin ei normaalilla käyttäjillä helpolla pystyttäisi. Simulaatio viidellä käyttäjällä jätettiin päälle kolmeksi päiväksi, jolloin ilmeni ongelmia tilannekuvien kohdalla. Saman ongelman ilmeneminen ilman

simulointia olisi voinut kestää kuukausia, ellei jopa vuosia. Simulaation ansios-
ta siis huomattiin tilannekuvien lopulta täyttävän kiintolevyn, jonka jälkeen pal-
velu kaatuu.

Käyttäjien määrä saatiin moninkertaistettua alkutilanteesta ja palvelu saatiin
muokattua sellaiseksi, että lisätäkseen myöhemmin maksimi yhtäaikaisten
käyttäjien määrää, tarvitsee vain lisätä tarvittavia komponentteja pilveen, sekä
HAProxyssä olevaan konelistaan niiden osoitteet.

HAProxy:n käyttämistä menetelmistä todettiin vähiten yhteyksiä -menetelmän
olevan sopivin palvelun tämän hetkisellemme kokoonpanolle. HAProxyä ja siihen
työssä käytettyjä asetuksia käytetään pohjana palvelun automatisoidun ympä-
ristön pystytyksen kuormantasaajassa.

Suurin ero alkutilanteeseen verrattuna on se, että alustavissa testeissä Team-
board usein jumiutui täysin, lopputesteissä ainoa tapa kaataa koko palvelu oli
luoda niin paljon tilannekuvia, että kiintolevyt täyttyivät täysin, joka sitten kaa-
toi palvelun. Tämänkin sai kumottua sillä, että poisti tilannekuvat käytöstä,
jolloin Teamboard ei kaatunut suurenkaan taakan alla, vaan alkoi vain toimia
hitaasti. Kuorman rauhoituttua Teamboard toimi taas normaalisti.

Lähteet

Aalto-yliopiston pilvipalveluohje. 2011. Artikkelin Aalto-yliopiston sivuilla. Viitattu 3.8.2014.

https://wiki.aalto.fi/download/attachments/58941866/Aalto_yliopiston_pilvipalveluohje.pdf.

Algorithms. 2014. Dokumentaatiossa Rackspace Cloud Load Balancers Developer Guide - API v1.0 yrityksen Rackspace sivustoilla. Viitattu 10.8.2014. <http://docs.rackspace.com/loadbalancers/api/v1.0/clb-devguide/content/Algorithms-d1e4367.html>.

Anicas, M. 2014. An Introduction to HAProxy and Load Balancing Concepts. Viitattu 8.10.2014. <https://www.digitalocean.com/community/tutorials/an-introduction-to-haproxy-and-load-balancing-concepts>.

Apache JMeter. 2014. Artikkelin yrityksen The Apache Software Foundation sivustoilla. Viitattu 11.8.2014. <http://jmeter.apache.org/>.

Apache Module mod_proxy. 2014. Dokumentaatio yrityksen The Apache Software Foundation sivustoilla. Viitattu 10.8.2014 http://httpd.apache.org/docs/2.2/mod/mod_proxy.html.

Apache Module mod_proxy_balancer. 2014. Dokumentaatio yrityksen The Apache Software Foundation sivustoilla. Viitattu 11.8.2014 http://httpd.apache.org/docs/2.2/mod/mod_proxy_balancer.html

API Load Testing Made Easy. 2014. Artikkelin yrityksen SmartBear Software sivustoilla. Viitattu 11.8.2014. <http://smartbear.com/products/qa-tools/web-service-load-testing/>.

Arregoces, M. & Portolani, M. 2003. Data Center Fundamentals: Understand Data Center Network Design and Infrastructure Architecture, Including Load Balancing, SSL, and Security. Cisco Press. © 2003. Books24x7. Viitattu 17.8.2014. <http://common.books24x7.com.ezproxy.jamk.fi:2048/toc.aspx?bookid=45396>

Davis, J. 2012. Load Balancing with HAProxy. Artikkelin Instant Servers Cloud Home sivustolla. Viitattu 16.8.2014. <http://docs.instantservers.telefonica.com/display/isc2/Load+Balancing+with+HAProxy>.

Hale, C. 2010. How to Safely Store A Password. Artikkelin Coda Hale sivustoilla. Viitattu 10.8.2014. <http://codahale.com/how-to-safely-store-a-password/>

HAProxy. 2014. Artikkelin HAProxy:n kotisivuilla. Viitattu 16.8.2014.
<http://www.haproxy.org/>.

Hassinen, M. 2014. Verkko-ohjelmointi. Verkko-ohjelmointi luennot 3-4, Itä-Suomen Yliopiston sivustoilla. Viitattu 11.8.2014.
<http://www.cs.uku.fi/~mhassine/VOH/Luennot/lu3-4.html>.

Health Checking On Load Balancers: More Art Than Science. 2012. Artikkelin The Data Center Overlords sivustoilla. Viitattu 11.8.2014.
<http://datacenteroverlords.com/2012/03/06/health-checking-load-balancers-more-art-than-science/>.

Heino, P. 2010. Pilvipalvelut. Hämeenlinna: Kariston Kirjapaino.

Heyman, J., Byström, C., Hamrén, J. & Heyman, H. 2014. What is Locust? Viitattu 11.8.2014. <http://docs.locust.io/en/latest/what-is-locust.html>.

Introduction. 2014. Artikkelin yrityksen DigitalOcean sivustoilla. Viitattu 11.8.2014. <https://www.digitalocean.com/help/>.

Introduction to MongoDB. 2014. Artikkelin MongoDB:n sivustoilla. Viitattu 19.8.2014. <http://www.mongodb.org/about/introduction/>

Introduction to Redis. N.d. Artikkelin Redis.IO sivustoilla. Viitattu 20.8.2014.
<http://redis.io/topics/introduction>

Kautto, T. 1996. Ohjelmistotestaus ja siinä käytettävät työkalut. Viitattu 13.9.2014.
<http://www.cs.utu.fi/opinnot/kurssit/Salo/kevat2011/tekniikkakartta.pdf>

Koivisto, M. 2014. TCP- ja UDP-protokollat. Viitattu 10.8.2014.
http://oppimateriaalit.internetix.fi/fi/avoimet/6tekniikkatalous/verkko/tcp_ja_udp_protokollat.

Lesson: A II About Sockets. 2014. Artikkelin yrityksen Oracle sivustoilla. Viitattu 11.8.2014. <http://docs.oracle.com/javase/tutorial/networking/sockets/>.

Load Balancer. N.d. Opetusmateriaali yrityksen f5 sivustolla. Viitattu 3.8.2014.
<https://f5.com/glossary/load-balancer>.

MacVittie, D. 2009. Intro to Load Balancing for Developers – The Algorithms. Viitattu 8.10.2014. <https://devcentral.f5.com/articles/intro-to-load-balancing-for-developers-ndash-the-algorithms>.

Mcheick, H., Mohammed, Z. R. & Lakiss, A. 2011. Evaluation of load balance Algorithms. Software Engineering Research, Management and Applications

(SERA), 9th International Conference on, 104-109.
<http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6065626>.

N4S-ohjelma. 2014. Artikkele yrityksen Digile sivustoilla. Viitattu 5.6.2014.
<http://www.n4s.fi/fi/>.

Nimipalvelimet. 2014. Usein kysytyt kysymykset viestintäviraston sivustoilla. Viitattu 11.8.2014.
[https://domain.fi/info/index/tietoa/useinkysytytkysymykset.html#312-NjNhOWYwZWZWE3YmI5ODA1MDc5NmI2NDIiODU0ODE4NDU\\$61\\$-NXhyWmV6UTY5-0-aeFa6IBb2-aeFb0mxga](https://domain.fi/info/index/tietoa/useinkysytytkysymykset.html#312-NjNhOWYwZWZWE3YmI5ODA1MDc5NmI2NDIiODU0ODE4NDU61-NXhyWmV6UTY5-0-aeFa6IBb2-aeFb0mxga).

Ohjelmiston testaus. N.d. Artikkele yrityksen ScienceSoft sivustoilla. Viitattu 13.9.2014. <http://www.scnsoft.com/fi/palvelut/ohjelmiston-testaus>

OSI-malli. N.d. Artikkele Raahen tekniikan ja talouden yksikön sivustolla. Viitattu 3.8.2014. http://www.ratol.fi/opensource/lahiverkot/fin/yleista/osi_malli.htm.

Pilvipalvelut. 2014a. Artikkele yrityksen Digia sivustolla. Viitattu 3.8.2014.
<https://www.digia.com/fi/Mita-teemme/Tarjoomat/Pilvipalvelut/>.

Pilvipalvelut. 2014b. Artikkele yrityksen IBM sivustoilla. Viitattu 11.8.2014.
<http://www-05.ibm.com/fi/solutions/cloud/>

Pilvipalvelut työntyy IT-ostamiseen. 2014. MARKET-VISIO OY:n julkaisema raportti Tieto- ja viestintätekniikan ammattilaiset ry:n sivustoilla. Viitattu 20.8.2014. <http://www.ttlry.fi/pilvimaailma-ty%C3%B6ntyy-it-ostamiseen-market-visio-oy>

Pilvistrategia suomalaisissa yrityksissä. 2013. Descomin sivuillaan julkaisema tutkimus. Viitattu 20.8.2014. <http://www.descom.fi/wp-content/uploads/2014/05/Pilvitutkimus.pdf>

Protokollatyypit. N.d. Artikkele Raahen tekniikan ja talouden yksikön sivustolla. Viitattu 20.8.2014.
http://www.ratol.fi/opensource/lahiverkot/fin/yleista/protokolla_tyypit.htm

RFC2616. 1999. Hypertext Transfer Protocol -- HTTP/1.1. IETF-organisaation standardi. Viitattu 10.8.2014. <https://www.ietf.org/rfc/rfc2616.txt>.

RFC6455. 2011. The WebSokcet Protocol. . IETF-organisaation standardi. Viitattu 13.9.2014. <http://tools.ietf.org/html/rfc6455>

RFC793. 1981. TRANSMISSION CONTROL PROTOCOL. IETF-organisaation standardi. Viitattu 10.8.2014. <https://www.ietf.org/rfc/rfc793.txt>.

Socket.IO. 2014. Dokumentaatio socket.IO sivustoilla. Viitattu 20.8.2014
<http://socket.io/>

Tarreau, W. 2011. Haproxy Configuration Manual. Viitattu 11.8.2014.
<http://www.haproxy.org/download/1.3/doc/configuration.txt>.

Tarreau, W. 2014a. Making applications scalable with Load Balancing. Viitattu 3.8.2014. http://1wt.eu/articles/2006_lb/index.html.

Tarreau, W. 2014b. Haproxy Configuration Manual. Viitattu 18.8.2014.
<http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-balance>

The OSI Models's Seven Layers Defined and Functions Explained. 2014. Artikkelin yrityksen microsoft sivustoilla. Viitattu 10.8.2014.
<http://support.microsoft.com/kb/103884>.

Understanding Load Balancing. N.d. Artikkelin yrityksen Liquid Web sivustolla. Viitattu 3.8.2014. <http://www.liquidweb.com/kb/understanding-load-balancing/>.

What is a proxy server. N.d. Artikkelin Indiana University sivustoilla. Viitattu 3.8.2014. <https://kb.iu.edu/d/ahoo>.

What is an API? 2012. Dokumentaatio yrityksen 3scale sivustoilla. Viitattu 20.8.2014. <http://www.3scale.net/wp-content/uploads/2012/06/What-is-an-API-1.0.pdf>

What is SoapUI? 2014. Artikkelin yrityksen SmartBear Software sivustoilla. Viitattu 11.8.2014. <http://www.soapui.org/About-SoapUI/what-is-soapui.html>.

What is the difference between Static and Dynamic Load Balancing? N.d. Tietoa kuormantasauksesta yrityksen NetOptics sivustolla. Viitattu 3.8.2014.
<http://www.netoptics.com/portal/kb/faq/29-06-2013/4043>.

Liitteet

Liite 1. HAProxyn asetukset

#Asetetaan perusasetukset siltä varalta, että ne unohtuu määrittää johonkin.

defaults

mode http

fullconn 4096

maxconn 2048

timeout queue 600s

timeout connect 5s

timeout client 600s

timeout server 600s

#Kuunnellaan porttiin 80 tulevia yhteyksiä.

frontend all 0.0.0.0:80

timeout client 86400000

#Luetaan osoitteet johon tullaan, ja sen mukaan laitetaan ACL:ään.

acl is_api hdr_dom(host) dev.api.teamboard.n4sjamk.org

acl is_ws hdr_end(host) dev.io.teamboard.n4sjamk.org

acl is_api_hdr_dom(hdr_dom(host) api-tb-haproxy.in.n4sjamk.org

acl is_ws_hdr_end(hdr_end(host) io-tb-haproxy.in.n4sjamk.org

acl is_crypt_hdr_dom(hdr_dom(host) tb-crypt-haproxy.in.n4sjamk.org

acl is_crypt_hdr_end(hdr_end(host) tb-crypt-haproxy.in.n4sjamk.org

#ACL:n perusteella ohjataan liikenne haluttuun backendiin.

use_backend API if is_api

use_backend WS if is_ws

use_backend CRYPT if is_crypt

#Jos ei tarkempaa osoitetta ohjataan statistiikka sivulle.

default_backend www_stats

#Tarjotaan statistiikka sivu ja tehdään sille tunnuksen

backend www_stats

```
mode http
stats enable
#stats uri /haproxy?stats
stats uri /
stats realm Strictly\ Private
stats auth eioleaitoUSER:eikäSALASANA
```

```
#backend API-koneille
backend API
#Kuormantaus-algoritmi, tässä vaiheessa Round Robin
balance roundrobin
```

```
timeout server 30000
timeout connect 4000
#Backend koneet joille kuorma jaetaan, painoarvo, maksimi yhteysmäärät,
# sekä kerrotaan että tehdään tilatarkastuksia.
server API1 10.129.183.68:9002 weight 1 maxconn 2048 check
server API2 10.129.179.210:9002 weight 1 maxconn 2048 check
server API3 10.129.179.209:9002 weight 1 maxconn 2048 check
server API4 10.129.174.127:9002 weight 1 maxconn 2048 check
server API5 10.129.175.189:9002 weight 1 maxconn 2048 check
```

```
#Soketti yhteyksien backend.
backend WS
#Liikenne ohjataan IP-osoitteen mukaan samalle koneelle jatkuvasti.
balance source
hash-type consistent
stick-table type ip size 1m expire 2h
stick on src
server WebSocket1 10.129.183.72:9001 weight 1 maxconn 2048 check
```

Liite 2. Apachen asetukset

```
#Asetetaan Apache kuuntelemaan porttia 80
<VirtualHost *:80>
#Estetään Apachea toimimasta eteenpäin ohjaavana välityspalvelimena
    ProxyRequests off
#Asetetaan serverin nimi
    ServerName tb.n4sjamk.org
#Luodaan tasaaja //mycluster, johon asetetaan yksi kone, perässä #esimerk-
kinä kuinka lisättäisiin useampi kone: Tarvitsee vain poistaa #-merkki #niin
kuorma jaettaisiin kahdelle koneelle
    <Proxy balancer://mycluster>
        BalancerMember http://188.226.163.232:3000
        #BalancerMember http://10.176.42.148:80

        # Tietoturva asetukset, joilla sallitaan kaikille pääsy sivuille

        Order Deny,Allow
        Deny from none
        Allow from all

        # Asetetaan kuormantasaus toimimaan "Round Robin"
        # -menetelmällä, eli kuorma jaetaan vuorotellen kaikille koneille.

        ProxySet lbmethod=byrequests

    </Proxy>

# Asetetaan web-käyttöliittymä toimimaan palvelimella
#"/balancer- manager" -hakemistossa
```



```
<Location /balancer-manager>
```

```
    SetHandler balancer-manager
```

```
    # Sallitaan sinnekin pääsykaikille, myöhemmin tässä kohdassa
```

```
    # olisi järkevää rajoittaa pääsy hallintaan vain tietyille käyttäjille.
```

```
    Order deny,allow
```

```
    Allow from all
```

```
</Location>
```

```
# Kerrotaan mitä halutaan tasata, alla olevilla konffeilla siis kaikki muu
```

```
# kuin "/balancer-manager"-hakemisto, ja varmuuden vuoksi vielä
```

```
# mycluster.
```

```
ProxyPass /balancer-manager !
```

```
ProxyPass / balancer://mycluster/
```

```
</VirtualHost>
```

Liite 3. Locust testin asetukset

```

from os import urandom
from locust import HttpLocust, TaskSet, task

import json
import random

class TeamboardTasks(TaskSet):

    def on_start(self):
        self.boards = [ ]
        self.tickets = [ ]

        self.user = {
            'email': 'test_' + urandom(16).encode('hex') + '@garnet.red',
            'password': 'test_password'
        }
        self.token = None

        self.client.post('/auth/register', self.user)

    #@task(1)
    #def login(self):
        response = self.client.post('/auth/login', self.user)
        self.token = response.headers['x-access-token']

    @task(1)
    def post_board(self):

```

```

if self.token is None: return
if len(self.boards) > 1: return
response = self.client.post('/boards',
    data = {
        'name': 'Botti vauhdissa',
        'info': 'vrrrruuuummm',
        'isPublic': 'true'
    },
    headers = {
        'authorization': 'bearer ' + self.token + "
    })
self.boards.append(response.json())

```

```
@task(2)
```

```

def get_board(self):
    if self.token is None: return
    if len(self.boards) is 0: return

    board = random.choice(self.boards)
    self.client.get('/boards/' + board['id'] + ",
        headers = {
            'authorization': 'bearer ' + self.token + "
        })

```

```
@task(1)
```

```

def get_boards(self):
    if self.token is None: return

    self.client.get('/boards',
        headers = {
            'authorization': 'bearer ' + self.token + "
        })

```

```

@task(3)
def post_ticket(self):
    if self.token is None: return
    if len(self.boards) is 0: return
    if len(self.tickets) > 5: return
    board = random.choice(self.boards)

    response = self.client.post('/boards/' + board['id'] + '/tickets',
        data = {
            'heading': 'Liikuteltavaa',
            'content': 'sisaltoa'
        },
        headers = {
            'authorization': 'bearer ' + self.token + "
        })

    ticket = response.json()
    self.tickets.append({
        'ticket': ticket['id'],
        'board': board['id']
    })

@task(15)
def move_ticket(self):
    if self.token is None: return
    if len(self.tickets) is 0: return

    ticket = random.choice(self.tickets)

    data = { }
    data['position'] = { 'x': 100, 'y': 300, 'z': 0 }

```

```

response = self.client.put('/boards/' + ticket['board'] + '/tickets/' +
    ticket['ticket'] + ",
data = json.dumps({
    'position': {
        'x': random.randint(0, 712),
        'y': random.randint(0, 556),
        'z': 0
    }
}),
headers = {
    'content-type': 'application/json',
    'authorization': 'bearer ' + self.token + "
})

```

```

class TeamboardUser(HttpLocust):
    task_set = TeamboardTasks
    min_wait = 1000
    max_wait = 2000

```


Liite 4. Mittausten 5-9 tulokset koottuna

250 Teho-käyttäjän testi					250 käyttäjää privaateilla tauluilla				
API-koneiden määrä	1	2	3	4	Apien määrä	1	2	3	4
Kestikö kuorman	Kyllä	Kyllä	Kyllä	Kyllä	Kestikö kuorman	Kyllä	Kyllä	Kyllä	Kyllä
Prosesessorin käyttö arvio	100 %	90 %	60 %	50 %	Prosesessorin käyttö arvio	85-95%	47-54%	38 %	27-36%
Pyyntöjen määrä sekunn	23,8	122,2	113,3	132,7	Pyyntöjen määrä sekunnis	122,2	120,2	122,8	119,2
Login latenssin keskiarvo	12291	1550	221	149	Login latenssin keskiarvo r	11857	1462	206	104
500 Teho-käyttäjän testi					500 käyttäjää privaateilla tauluilla				
Apien määrä	1	2	3	4	Apien määrä	1	2	3	4
Kestikö kuorman	Ei	Kyllä	Kyllä	Kyllä	Kestikö kuorman	Ei	Kyllä	Kyllä	Kyllä
Prosesessorin käyttö arvio	-	98 %	90-98%	85-95%	Prosesessorin käyttö arvio	-	75 %	70 %	51 %
Pyyntöjen määrä sekunn	-	32,1	154,6	187,5	Pyyntöjen määrä sekunnis	-	243,2	239,2	246,8
Login latenssin keskiarvo	-	7676	1320	596	Login latenssin keskiarvo r	-	3539	499	302
750 Teho-käyttäjän testi					750 käyttäjää privaateilla tauluilla				
Apien määrä	1	2	3	4	Apien määrä	1	2	3	4
Kestikö kuorman	Ei	Ei	Kyllä	Kyllä	Kestikö kuorman	Ei	Kyllä	Kyllä	Kyllä
Prosesessorin käyttö arvio	-	-	98-99%	96 %	Prosesessorin käyttö arvio	-	98 %	90 %	70-75%
Pyyntöjen määrä sekunn	-	-	111,4	135,1	Pyyntöjen määrä sekunnis	-	281,4	376,3	320,2
Login latenssin keskiarvo	-	-	4382	2600	Login latenssin keskiarvo r	-	6611	1295	1126
Bcrypt Erillään									
Käyttäjien määrä	250	500	1000	1000	1250				
API-koneiden määrä	1	1	2	3	4				
Kestikö kuorman	Kyllä	Kyllä	Kyllä	Kyllä	Kyllä				
Prosesessorin käyttö arvio	52 %	96 %	95 %	95 %	78 %				
Pyyntöjen määrä sekunnissa	74,3	182	251,4	370,5	394,3				

Liite 5. Mittauksen 7 tulokset

250 käyttäjällä

Yhdellä API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

250 users

Edit

SLAVES

2

RPS

122.2

FAILURES

0%

STOP

Reset Stats


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	156	0	12000	11857	7609	15828	92	0
POST	/api/v1/auth/register	121	0	19000	18460	12086	24723	92	0
GET	/api/v1/boards	1171	0	35	147	13	2118	741	7.5
POST	/api/v1/boards	652	0	18	99	7	1426	224	1.3
POST	/api/v1/boards/53e339ad7e70b23545180a3b/tickets	2	0	14	34	14	53	159	0
PUT	/api/v1/boards/53e339ad7e70b23545180a3b/tickets/53e33a107e70b23545180fe9	9	0	30	50	14	199	162	0.1

Kahdella API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

250 users

Edit

SLAVES

2

RPS

120.2

FAILURES

0%

STOP

[Reset Stats](#)


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	31	0	1400	1462	114	2618	92	0
POST	/api/v1/auth/register	17	0	3800	3205	573	4096	92	0
GET	/api/v1/boards	2277	0	24	26	14	265	892	6.8
POST	/api/v1/boards	653	0	11	13	7	198	224	0
GET	/api/v1/boards/53e33c04c2eef3a527de601d	0	0	0	0	0	0	0	0
POST	/api/v1/boards/53e33c04c2eef3a527de601d/tickets	2	0	17	21	17	25	159	0

Kolmella API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

250 users

Edit

SLAVES

2

RPS

122.8

FAILURES

0%

STOP

Reset Stats


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	4	0	110	206	107	304	92	0
POST	/api/v1/auth/register	1	0	280	277	277	277	92	0
GET	/api/v1/boards	2533	0	10	11	6	124	533	7
POST	/api/v1/boards	644	0	10	11	7	102	224	0
GET	/api/v1/boards/53e33f6b7e70b235451818b9	0	0	0	0	0	0	0	0
POST	/api/v1/boards/53e33f6b7e70b235451818b9/tickets	1	0	18	18	18	18	159	0

Neljällä API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

250 users

Edit

SLAVES

2

RPS

119.2

FAILURES

0%

STOP

Reset Stats

Statistics


Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	2	0	100	104	104	104	92	0
POST	/api/v1/auth/register	0	0	0	0	0	0	0	0
GET	/api/v1/boards	1327	0	10	11	6	54	459	7.6
POST	/api/v1/boards	643	0	10	11	6	126	224	0.4
GET	/api/v1/boards/53e342042b7345dc04163d35	4	0	14	16	12	20	1271	0
POST	/api/v1/boards/53e342042b7345dc04163d35/tickets	2	0	14	16	14	18	159	0

500 käyttäjällä

Yhdellä API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

485 users

Edit

SLAVES

2

RPS

36

FAILURES

9%

STOP

Reset Stats


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	33	15	17000	17272	15756	19182	92	3
POST	/api/v1/auth/register	44	26	28000	27851	0	29989	92	4.4
GET	/api/v1/boards	52	0	1300	1499	844	2525	156	4.4
POST	/api/v1/boards	43	0	830	842	373	1651	224	4
GET	/api/v1/boards/53e34a217e70b23545181ea3	1	0	2100	2124	2124	2124	943	0.1
POST	/api/v1/boards/53e34a217e70b23545181ea3/tickets	0	0	0	0	0	0	0	0

Kahdella API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

500 users

Edit

SLAVES

2

RPS

243.2

FAILURES

0%

STOP

Reset Stats


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	86	0	3400	3593	137	5589	92	0
POST	/api/v1/auth/register	54	0	6800	6409	4319	8816	92	0
GET	/api/v1/boards	2038	0	29	265	8	4136	426	14.1
POST	/api/v1/boards	1082	0	19	164	7	2752	224	1.5
POST	/api/v1/boards/53e351457e70b23545182994/tickets	1	0	14	14	14	14	159	0
PUT	/api/v1/boards/53e351457e70b23545182994/tickets/53e35185c2eef3a527de6d9c	12	0	42	99	11	373	162	0.3

Kolmella API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

500 users

Edit

SLAVES

2

RPS

239.2

FAILURES

0%

STOP

Reset Stats


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	10	0	250	499	119	1041	92	0
POST	/api/v1/auth/register	4	0	970	1023	925	1104	92	0
GET	/api/v1/boards	3664	0	14	22	7	1008	518	17.2
POST	/api/v1/boards	1114	0	11	16	6	604	224	0.2
GET	/api/v1/boards/53e358d9148ff398228478a9	2	0	13	13	13	13	1108	0
POST	/api/v1/boards/53e358d9148ff398228478a9/tickets	0	0	0	0	0	0	0	0

Neljällä API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

500 users

Edit

SLAVES

2

RPS

246.8

FAILURES

0%

STOP

Reset Stats

Statistics


Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	5	0	280	302	126	543	92	0
POST	/api/v1/auth/register	4	0	560	589	529	692	92	0
GET	/api/v1/boards	1470	0	11	15	7	321	394	13.8
POST	/api/v1/boards	874	0	11	13	6	160	224	4.6
POST	/api/v1/boards/53e35db4148ff398228485d5/tickets	0	0	0	0	0	0	0	0
PUT	/api/v1/boards/53e35db4148ff398228485d5/tickets/53e35dc7148ff39822848628	11	0	21	21	13	30	162	0.2

750 käyttäjällä

Kahdella API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

750 users

Edit

SLAVES

2

RPS

281.4

FAILURES

0%

STOP

Reset Stats


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	167	0	6300	6611	119	12605	92	0
POST	/api/v1/auth/register	99	0	16000	13155	6838	17708	92	0
GET	/api/v1/boards	1594	0	350	1203	7	11530	310	16.5
POST	/api/v1/boards	1233	0	160	666	7	7155	224	8.6
GET	/api/v1/boards/53e361c47e70b2354518451e	3	0	370	5157	36	15069	1154	0
POST	/api/v1/boards/53e361c47e70b2354518451e/tickets	1	0	110	108	108	108	159	0
PUT	/api/v1/boards/53e361c47e70b2354518451e/tickets/53e361c6c2eef3a527de8648	2	0	25	221	25	416	163	0

Kolmella API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

750 users

[Edit](#)

SLAVES

2

RPS

376.3

FAILURES

0%

STOP

[Reset Stats](#)


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	23	0	1500	1295	128	1910	92	0
POST	/api/v1/auth/register	11	0	1700	1597	291	1893	92	0
GET	/api/v1/boards	1531	0	22	87	7	2495	373	23.2
POST	/api/v1/boards	1059	0	18	67	7	1466	224	10
GET	/api/v1/boards/53e366117e70b235451853b1	1	0	14	14	14	14	1273	0
POST	/api/v1/boards/53e366117e70b235451853b1/tickets	0	0	0	0	0	0	0	0
PUT	/api/v1/boards/53e366117e70b235451853b1/tickets/53e36615148ff39822848aae	4	0	16	38	16	93	163	0.1

Neljällä API-koneella:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

750 users

[Edit](#)

SLAVES


2

RPS

320.2

FAILURES

0%



STOP

Reset Stats

Statistics



Failures

Exceptions



Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	14	0	470	1126	127	2378	92	0
POST	/api/v1/auth/register	8	0	2100	1591	273	2421	92	0
GET	/api/v1/boards	1222	0	16	251	7	4240	332	21.4
POST	/api/v1/boards	868	0	15	148	7	2231	224	10.5
GET	/api/v1/boards/53e36afe7e70b23545185d80	4	0	15	82	14	218	1267	0.1
POST	/api/v1/boards/53e36afe7e70b23545185d80/tickets	1	0	71	71	71	71	159	0

Liite 6. Mittauksen 8 tulokset


Yksi API-kone 250 käyttäjää:

<div>  LOCUST <small>A MODERN LOAD TESTING TOOL</small> </div> <div> STATUS RUNNING 250 users Edit </div> <div> SLAVES 2 </div> <div> RPS 74.3 </div> <div> FAILURES 0% </div> <div>  </div> <div> Reset Stats </div>									
Statistics Failures Exceptions									
Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	0	0	0	0	0	0	0	0
POST	/api/v1/auth/register	0	0	0	0	0	0	0	0
GET	/api/v1/boards	146	0	10	13	6	104	100	8
POST	/api/v1/boards	172	0	10	13	5	89	158	8.7
GET	/api/v1/boards/53e8ac833f41a376447085fb	2	0	7	7	7	7	551	0.1
POST	/api/v1/boards/53e8ac833f41a376447085fb/tickets	0	0	0	0	0	0	0	0
PUT	/api/v1/boards/53e8ac833f41a376447085fb/tickets/53e8ac8a3f41a3764470864c	3	0	13	13	12	14	162	0.1

Yksi API-kone 500 käyttäjää:

<div>  LOCUST <small>A MODERN LOAD TESTING TOOL</small> </div> <div> STATUS RUNNING 500 users Edit </div> <div> SLAVES 2 </div> <div> RPS 182 </div> <div> FAILURES 0% </div> <div>  </div> <div> Reset Stats </div>									
Statistics Failures Exceptions									
Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	0	0	0	0	0	0	0	0
POST	/api/v1/auth/register	0	0	0	0	0	0	0	0
GET	/api/v1/boards	412	0	89	164	6	819	139	13.3
POST	/api/v1/boards	376	0	49	93	5	460	158	11.4
POST	/api/v1/boards/53e8acfe3f41a37644708c73/tickets	0	0	0	0	0	0	0	0
PUT	/api/v1/boards/53e8acfe3f41a37644708c73/tickets/53e8ad203f41a37644708f65	6	0	14	120	9	616	163	0
PUT	/api/v1/boards/53e8acfe3f41a37644708c73/tickets/53e8ad263f41a37644709033	2	0	26	31	26	36	163	0
GET	/api/v1/boards/53e8acfe3f41a37644708c76	0	0	0	0	0	0	0	0
POST	/api/v1/boards/53e8acfe3f41a37644708c76/tickets	0	0	0	0	0	0	0	0
PUT	/api/v1/boards/53e8acfe3f41a37644708c76/tickets/53e8ad0a3f41a37644708d2c	3	0	320	226	16	342	162	0.2

Kaksi API-konetta 1000 käyttäjää:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

1000 users

Edit

SLAVES

2

RPS

251.4

FAILURES

0%

STOP

Reset Stats


Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	7	0	870	877	702	979	92	0
POST	/api/v1/auth/register	0	0	0	0	0	0	0	0
GET	/api/v1/boards	1210	0	1300	1412	7	6387	231	15.7
POST	/api/v1/boards	888	0	640	699	6	3680	158	8.9
GET	/api/v1/boards/53e8ae973f41a376447098ec	0	0	0	0	0	0	0	0
POST	/api/v1/boards/53e8ae973f41a376447098ec/tickets	0	0	0	0	0	0	0	0
PUT	/api/v1/boards/53e8ae973f41a376447098ec/tickets/53e8ae990aaffc694b7e2b95	2	0	14	1362	14	2709	163	0
PUT	/api/v1/boards/53e8ae973f41a376447098ec/tickets/53e8aea10aaffc694b7e2bd4	2	0	130	715	125	1305	162	0.1

Kolme API-konetta 1000 käyttäjää:



LOCUST

A MODERN LOAD TESTING TOOL

STATUS

RUNNING

1000 users

Edit

SLAVES

2

RPS

370.5

FAILURES

0%

STOP

Reset Stats

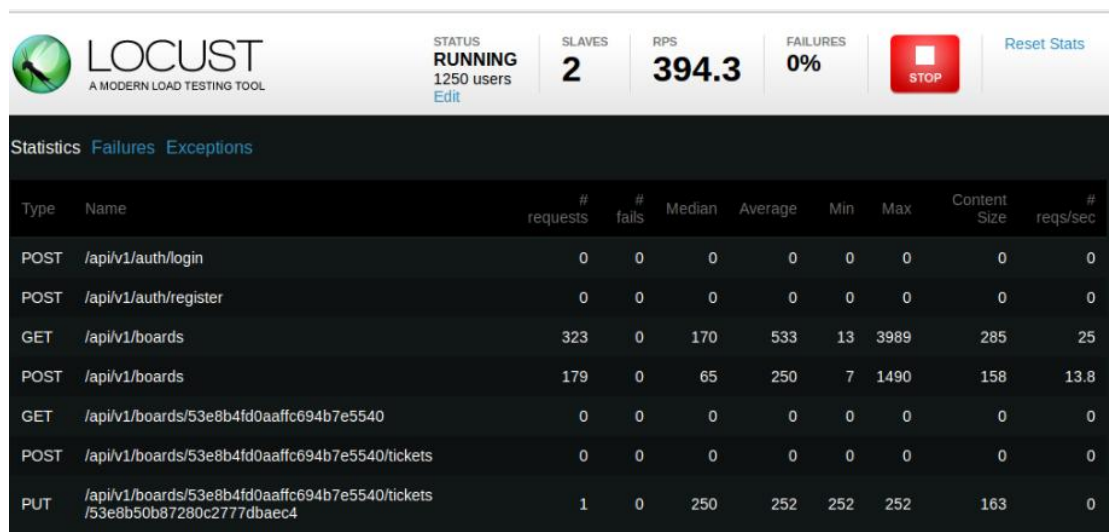
Statistics

Failures

Exceptions

Type	Name	# requests	# fails	Median	Average	Min	Max	Content Size	# reqs/sec
POST	/api/v1/auth/login	0	0	0	0	0	0	0	0
POST	/api/v1/auth/register	0	0	0	0	0	0	0	0
GET	/api/v1/boards	1835	0	120	458	7	4762	275	21.6
POST	/api/v1/boards	1063	0	82	242	5	2771	158	8.5
GET	/api/v1/boards/53e8b0313f41a3764470a8ce	0	0	0	0	0	0	0	0
POST	/api/v1/boards/53e8b0313f41a3764470a8ce/tickets	0	0	0	0	0	0	0	0
PUT	/api/v1/boards/53e8b0313f41a3764470a8ce/tickets/53e8b0353f41a3764470a8e0	2	0	230	896	225	1566	163	0.1
PUT	/api/v1/boards/53e8b0313f41a3764470a8ce/tickets/53e8b0430aaffc694b7e3bc5	5	0	14	258	11	1240	162	0

Neljä API-konetta 1250 käyttäjää:

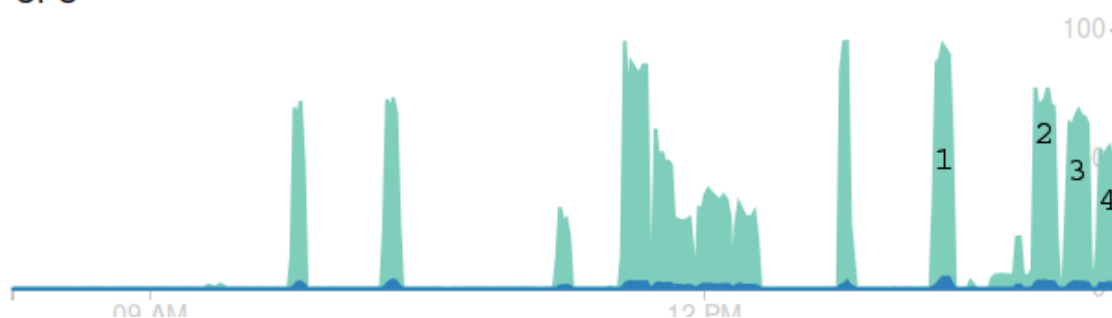


Prosessorin rasitukset kaikista 250 käyttäjän testeistä. Neljä viimeistä palkkia ovat näistä testeistä. Palkit ovat samassa järjestyksessä kuin testitkin:



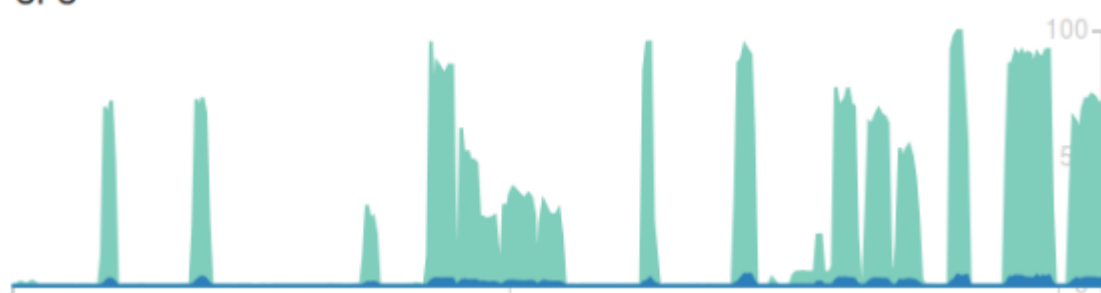
Kaikkien 500 käyttäjällä tehtyjen testien prosessorille aiheuttamat rasitukset eri API-kone määriille. Kuvioon on merkitty numeroilla eri API-koneiden määrät. Kohdassa kaksi onnistunut testi on kokonainen palkki, sitä edeltävä osuus tulee katkenneesta yhteydestä locust-koneeseen:

CPU



Kolme viimeistä palkkia ovat 750 käyttäjän testeistä. Ensin kahdella, sitten kolmella ja viimeisenä neljällä API-koneella:

CPU



HAProxy-koneen, eli kuormantasaajan prosessorin rasitus kaikkien mittausten ajalta:

CPU

