HANNU LAHTINEN

SUORITUSKYKYTESTAUKSEN AUTOMATISOINTI JA LIITTÄMINEN JATKUVAAN INTEGRAATIOON

Diplomityö

Tarkastaja:
professori Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
1. maaliskuuta 2017

TIIVISTELMÄ

HANNU LAHTINEN: Suorituskykytestauksen automatisointi ja liittäminen jatkuvaan integraatioon
Tampereen teknillinen yliopisto
Diplomityö, 53 sivua
Toukokuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Pervasive Systems
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: suorituskyky, testaus, automatisointi, jatkuva integraatio

Monivuotisessa ohjelmistoprojektissa havaittiin suorituskykyongelmien tulevan ilmi liian hitaasti. Ongelmat havaittiin monesti jopa kuukausia sen jälkeen, kun ongelman aiheuttanut koodi oli liitetty järjestelmän osaksi. Suorituskykytestaus oli manuaalista, mikä johti siihen, että testausta tehtiin liian harvoin ja tulokset eivät olleet luotettavia. Suorituskykytestauksen automatisointi mahdollistaa päivittäisen suorituskykytestauksen ja automatisoinnilla voidaan poistaa tulosten luotettavuuteen vaikuttaneita tekijöitä. Tämä työ kertoo, kuinka koko suorituskykytestausketju automatisoitiin ja käyttäjän käyttötuntuma-arvio korvattiin ohjelmallisilla mittauksilla.

Testauksen kohteena on asiakas-palvelin-mallin johtamisjärjestelmä, joka saa ulkoisista järjestelmistä tilannetietoja ja koostaa niistä karttapohjalle lähes reaaliaikaisen tilannekuvan. Käyttäjän on mahdollista manipuloida kartan näkymää ja antaa käyttöliittymässä syötteitä, jotka järjestelmä lähettää ulkoisille järjestelmille.

Järjestelmän suorituskykytestaus automatisoitiin toteuttamalla toistettavat kuormitusskenaariot. Skenaariot keskittyvät ominaisuuksiin, jotka eivät muutu järjestelmän kehittyessä, jolloin testituloksia voidaan kerätä ja verrata pitkällä aikavälillä. Kuormitusskenaarioissa toistetaan tilannetietoja sovelluspalvelimelle nauhoitteen ja generaattorin avulla ja operoidaan samalla käyttäjäsovellusta testiautomaatiotyökalun avulla. Kuormituksen ajalta kerätään läpimenoajan ja käyttöliittymäviiveen mittaustuloksia sekä resurssien käyttöastetietoja. Lisäksi toteutettiin jatkuvan integraation palvelimelle useita tehtäviä, joilla hallitaan kuormitusskenaarioiden suoritusta ja mittaustulosten keräystä. Jatkuvan integraation palvelimen tuottamat järjestelmäversiot testataan automaattisesti uusien tehtävien avulla.

ABSTRACT

HANNU LAHTINEN: Automating performance testing and making it a part of continuous integration
Tampere University of Technology
Master of Science Thesis, 53 pages
May 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Professor Hannu-Matti Järvinen

Keywords: performance, testing, automating, automatization, continuous integration

In a large software project, it became apparent that performance problems are not recognized in a timely fashion. Performance problems were recognized months after the code that caused the problem had been merged to the system. Performance testing was done manually, which led to testing being too sparse. Human interaction with the system also caused extra variance in the results. Automating performance testing enables daily testing of the system. It also removes some factors that cause variance in test results. This thesis explains the process of automating performance testing and how user's usability estimation was replaced by software metrics.

The system under test is a client-server model based command and control system. The system receives information from outside sources and compiles the information to form a situation picture for the user. Situation picture is a near real-time representation of the situation on a map surface. A user can manipulate the map view and give inputs based on the situation. Inputs are transmitted through the server to outside systems.

Performance testing was automated by creating repeatable load scenarios. The operations in the scenarios are focused on system operations that are trusted to stay the same as the system develops. This enables the performance data collected to be comparable over long periods of time and multiple versions of the system. In the load scenarios, data, that mimics outside sources, is fed to the server with a generator and a data recording. Client-side application load is produced by test automation tool that performs user inputs. During the load scenarios, the test system measures server throughput time, client's user interface lag and whole system's resource allocation. All test automation is controlled by continuous integration server jobs. This enables the project to automatically test the performance of every nightly build produced by the continuous integration server.

ALKUSANAT

Opinnäytetyö on tehty Tampereen teknillisen yliopiston tietotekniikan laitokselle. Työ toteutettiin Insta Defsec Oy:ssä, jossa olen viimeiset kolme vuotta työskennellyt. Kiitokset sen mahdollistamisesta Meetulle ja Matille, jotka luottivat minuun ja suosittelivat palkkaamaan minut melko aikaisessa vaiheessa opintoja.

Installa diplomityötäni ohjasivat Sami Vuorinen ja Teemu Salmivesi. Kiitokset heille molemmille monista neuvoista sekä itse työn toteutuksessa että diplomityön kirjoituksessa. Kiitos myös työni tarkastajalle professori Hannu-Matti Järviselle, joka neuvoi työn rakenteessa ja kieliopissa. Lisäksi hän suunnitteli kanssani sopivan aikataulun, jotta valmistuminen tänä keväänä oli mahdollista.

Suuri kiitos myös vaimolleni Veelle ja äidilleni Leenalle, jotka tukivat minua koko opintojeni ajan.

Tampereella, 17.5.2017

Hannu Lahtinen

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. SUORITUSKYKYMITTAUKSEN TEORIAA 3
2.1 Suorituskyvyn testaaminen 3
2.2 Suorituskyvyn mittaaminen 4
2.3 Java suorituskykytestauksessa 6
3. TESTATTAVA JÄRJESTELMÄ JA YMPÄRISTÖ 8
3.1 Hajautettu johtamisjärjestelmä 8
3.2 Teknologiat 9
3.3 Ympäristöt 10
3.4 Nykyiset testausmenetelmät 12
3.4.1 Yksikkötestit 12
3.4.2 Automaattitestit 12
3.4.3 Manuaalinen testaus 13
4. NYKYISTEN SUORITUSKYKYTESTIEN ONGELMAT 15
4.1 Hidas palaute 15
4.2 Inhimillinen tekijä 15
4.3 Tulosten tarkkuus 16
4.4 Testidatan generointi 17
4.5 Syyn jäljitys 17
5. VAATIMUKSET JA RAJAUKSET TOTEUTETTAVALLE TESTAUSJÄRJESTELMÄLLE 18
5.1 Vaatimukset 18
5.1.1 Automatisointi 18
5.1.2 Toistettavuus 18
5.1.3 Kehityksen kestävä 19
5.1.4 Laajennettavuus 19
5.1.5 Käyttöliittymäviiveen mittari 19
5.2 Rajaukset 20
5.2.1 Aikataulu 20
5.2.2 Hajautus 20
5.2.3 Käyttäjän liikkeiden jäljittely 20
6. SUORITUSKYKYTESTISKENAARIOIDEN SUUNNITTELU JA AUTOMATISOINTI 22
6.1 Skenaarioiden kuormitus 22
6.2 Skenaarioiden operaatiot 23
6.3 Testityökalut 23
6.3.1 Robot Framework 23
6.3.2 TeamQuest 24
6.4 Järjestelmän suorituskyvyn mittaaminen 25
6.4.1 Käyttöliittymäviiveen mittaus 25

6.4.2 Läpimenoajan mittaus ... 27
6.4.3 Resurssien käytön mittaus ja mittausten koostaminen ... 27
6.5 Automaattisten suorituskykytestien ajoympäristö ... 27
6.6 Robot Frameworkin laajentaminen skenaarioiden automatisointiin soveltuvaksi ... 29
6.7 Käyttäjäsovellusskenaarioiden automatisointi ... 31
6.8 Skenaarioiden suorituksen automatisointi ... 32
7. SUORITUSKYKYMITTAUSTEN AUTOMATISOINNIN TESTAUS JA TYÖN ARVIOINTI ... 38
7.1 Testausympäristön luotettavuuden tutkiminen ... 38
7.1.1 Windowsin etätyöpöytäyhteys väärentää mittaustuloksia ... 38
7.1.2 Ensimmäiset testit käyttöliittymäviiveen mittarille ... 39
7.1.3 Käyttäjäsovelluksen dynaamisen piirto-optimoinnin vaikutus tuloksiin ... 40
7.1.4 Pidemmät skenaarioajot auttavat löytämään mittaustuloksia vääristävän tekijän ... 40
7.1.5 Kolmen kuukauden kehityksen tuottamat erot käyttöliittymän viiveeseen ... 42
7.1.6 Käyttöliittymän tahallinen hidastaminen ... 43
7.1.7 Käyttöliittymäviiveen mittarin tulosten vertailu normaalilla ja hidastetulla versiolla ... 43
7.1.8 Muun suorituskykytestausjärjestelmän toiminta ... 46
7.2 Työn arviointi vaatimusten valossa ... 47
7.3 Jatkokehitys ... 48
8. YHTEENVETO ... 50
LÄHTEET ... 51

LYHENTEET JA MERKINNÄT

AWT	Abstract Window Toolkit on Javan käyttöliittymätoteutuksen perusta
DDS	Data Distribution Service on julkaisija-tilaaja-mallin tietoliityntästandardi.
EDT	Event Dispatch Thread on Javassa erityinen säie, jota käytetään käyttöliittymäelementtien päivitykseen.
OSGi-standardi	Open Services Gateway Initiative -standardi kuvaa Java-toteutusrakenteen, joka mahdollistaa modulaaristen sovellusten kehittämisen ja käyttöönoton.

1. JOHDANTO

Työn tavoitteena on suunnitella ja toteuttaa hajautetun johtamisjärjestelmän suorituskykytestauksen automatisointi. Työssä etsitään mittareita, joiden avulla järjestelmän suorituskykyä voidaan seurata luotettavasti. Lisäksi suunnitellaan pysyvät testiskenaariot, joita on mahdollista ajaa nyt ja testattavan järjestelmän kehittyessä. Tavoite on myös pystyä ajamaan skenaariot vanhemmilla versioilla, jotta voidaan kerätä tietoa suorituskyvyn kehittymisestä. Skenaarioiden ajo automatisoidaan ja lisätään jatkuvaan integraatioon, jonka osana suorituskykymittaukset tehdään joka yö. Mittauksien tulokset kerätään talteen ja niistä tärkeimmät esitetään käyttäjälle muodossa, josta on helppo nähdä järjestelmän suorituskykyisyyden kehitys.

Suorituskyky on olennainen osa mitä tahansa sovellusta [1]. Toimiva sovellus voi olla suorituskyvyltään niin heikko, ettei sitä voida käyttää. Yksinkertaisimmillaan tämä voi tarkoittaa, että laskinsovellus laskee laskut oikein, mutta tuloksen saamisessa kestää niin kauan, että laskin on periaatteessa käyttökelvoton. Tietenkin tänä päivänä laskinsovellukseen saa toteuttaa jo melkoisia solmuja, etteivät nykyajan prosessorit selviytyisi laskuista siedettävässä ajassa. Projektien koon ja monimutkaisuuden kasvaessa suorituskykyongelmien todennäköisyys ja vaikutus usein kasvavat. Useissa isoissa projekteissa suorituskykyongelmat ovat aiheuttaneet miljoonien eurojen vahinkoja [1, 2].

Yksi Suomen tunnetuimmista suorituskyvyn yliarvioinneista ja testauksen riittämättömyydestä lienee VR-Yhtymä Oy:n lipunmyyntijärjestelmä, joka uudistui vuonna 2011. Järjestelmä kaatui ensimmäisen kolmen tunnin sisällä julkaisusta. Lippujärjestelmän ongelmat ratkaistiin, mutta sekä VR-Yhtymä Oy:n että järjestelmän tuottaneiden Tiedon ja Accenturen maineeseen jäi kolhu. [3, 4]

Sovellusten suorituskyvyn tärkeydestä [1, 2, 5] huolimatta interaktiivisen sovelluksen suorituskykytestauksen automatisoinnista löytyy melko vähän kirjallisuutta [6]. Tämän työn perusteella voidaan sanoa, että suorituskykytestauksen automatisointi on työläs toteuttaa ja sen hyödyt tähtäävät pitkälle tulevaisuuteen. Tämän perusteella automatisointi ei välttämättä ole kannattavaa pienissä projekteissa, joiden kehitys kestää ennemmin kuukausia kuin vuosia. On siis ymmärrettävää, että aiheesta ei löytynyt yhtä paljon kirjallisuutta kuin loogisesta automaattitestauksesta. Monet teokset [6, 7] keskittyvät pienempiin osa-alueisiin automaattisessa suorituskykytestauksessa. Tämä diplomityö kertoo niitä korkeammalla abstraktiotasolla koko suorituskykytestauksen automatisointiprojektista.

Luvussa 2 käsitellään suorituskykytestauksen teoriaa. Luvussa selitetään kirjallisuuteen viitaten, miten suorituskykyä tulisi testata ja mitä testauksessa on tärkeää ottaa huomioon.

Koska järjestelmän ohjelmointikielenä on Java, käydään luvussa lisäksi läpi Javan erityispiirteitä, jotka vaikuttavat suorituskykyyn.

Tämä työ on toteutettu osana monivuotista ohjelmistoprojektia, jossa nähtiin automatisoinnille selvät edellytykset. Luvussa 3 käsitellään laajemmin projektia, testattavaa järjestelmää, käytettyjä teknologioita ja nykyisiä testausmenetelmiä. Luku 4 pureutuu manuaalisessa suorituskykytestauksessa esiintyneisiin ongelmiin, jotka tässä työssä halutaan ratkaista.

Työlle oli tärkeää asettaa vaatimukset, jotka testausjärjestelmän on tarpeen täyttää. Luvussa 5 esitellään näitä vaatimuksia. Lisäksi kerrotaan rajauksista, joita testausjärjestelmälle asetettiin. Luku 6 kuvaa, kuinka suorituskykytestauksen automatisointi käytännössä toteutettiin. Luvussa kerrotaan, miten järjestelmää kuormitetaan käyttöliittymäoperaatioilla ja tuottamalla järjestelmään tietopäivityksiä koneellisesti. Luvussa kuvataan ympäristö, jossa automaattiset suorituskykytestit ajetaan, sekä työkalut, joita hyödynnettiin suorituskykyskenaarioiden toistoon ja järjestelmän suorituskyvyn mittaamiseen. Lisäksi luvussa kuvataan, kuinka jatkuvan integraation palvelin hallitsee testiympäristöä ja suorittaa testiskenaariot automaattisesti.

Luvussa 7 arvioidaan toteutettua automaattista testausympäristöä. Luvun alkupuolella keskiöön asettuu testiskenaarioiden, ympäristön ja mittareiden toiminnan todentaminen. Myöhemmin luvussa arvioidaan työn onnistumista kokonaisuutena ja verrataan valmista työtä asetettuihin vaatimuksiin. Luvun lopuksi pohditaan jatkokehitysmahdollisuuksia. Luvussa 8 esitetään yhteenveto työn tuloksista.

2. SUORITUSKYKYMITTAUKSEN TEORIAA

Tässä luvussa kerrotaan yleisesti suorituskyvystä ja sen testauksesta. Lisäksi luvussa pureudutaan yksityiskohtaisemmin tässä työssä olennaisiin suorituskykyaiheisiin.

Suorituskyky perustuu käytännössä ajan ja suoritettujen operaatioiden suhteeseen. Mitä nopeammin operaatio suoritetaan, sitä parempi suorituskyky. Esimerkiksi auton suorituskykyä mitataan muun muassa huippunopeudella, kiihtyvyydellä ja teholla. Nopeushan kuvastaa matkan ja ajan suhdetta, kiihtyvyys taas nopeuden ja ajan suhdetta ja teho tehdyn työn ja ajan suhdetta. Kaikkia näitä suureita voidaan parantaa kasvattamalla auton moottoritehoa tai keventämällä autoa. Auton suorituskykyä parannetaan siis kasvattamalla resursseja tai vähentämällä työn määrää. Sama pätee tietotekniikkaan. Herkästi kuvitellaan, että sovelluksen toimiessa hitaasti olisi syytä lisätä resursseja, kuten prosessoritehoa tai käyttömuistia. Todellisuudessa sovelluksia suunnitellaan usein ympäristöihin, joissa resurssien lisääminen ei ole mahdollista. Sovelluksen käytössä on tietyt resurssit, joilla on pärjättävä. Usein tärkeämpää onkin tutkia, mikä sovelluksessa vie resursseja ja onko sovelluksen toimintaa mahdollista keventää.

Tässä diplomityössä etsitään ratkaisuja, joilla tarkkailla suorituskykyä jatkuvasti ja luotettavasti. Kun suorituskyvyn heikkeneminen havaitaan ajoissa, säästetään aikaa ja rahaa [1].

2.1 Suorituskyvyn testaaminen

Scott Oaks jakaa kirjassaan [5] suorituskykytestauksen kolmeen kategoriaan: mikro-, meso- ja makrotestit. Jako on yksinkertainen ja vastaa jakoa yksikkö-, integraatio- ja järjestelmätesteihin [8].

Mikrotestit vastaavat käytännössä yksikkötestejä. Niissä mitataan ohjelmallisesti mahdollisimman pienen kokonaisuuden suorituskykyä. Mikrotesteissä toistetaan yleensä pientä pätkää koodia tuhansia kertoja ja mitataan suoritukseen kulunut aika. Valitettavasti mikrotesteissä on helppo päätyä mittaamaan asioita väärin. Kehittäjien tulisi ottaa huomioon muun muassa kääntäjän tekemän optimoinnin erityispiirteitä ja mittausten epätarkkuus. Useimpien kehittäjien ei tarvitse tietää näitä asioita osana päivittäistä kehitystä ja he päätyvät tekemään virheellisiä mikrotestejä. Lisäksi mikrotestien tuloksista on vaikea vetää johtopäätöksiä yhtenäisen järjestelmän toiminnasta, koska mikrotestit keskittyvät niin pieniin osa-alueisiin. [5]

Mesotestit siirtyvät askeleen korkeammalle abstraktiotasolle. Niissä voidaan testata esimerkiksi tiedon läpivientiaikaa tietyssä järjestelmän komponentissa. Kokonaisen järjestelmän sijaan mesotesteissä käytetään järjestelmän osakokonaisuutta. Mesotestit voivat

päätyä antamaan vääristyneitä tuloksia, koska ne eivät käsittele tietoa samoin kuin järjestelmäkokonaisuus. [5]

Oaks näkee parhaiksi makrotestit, joissa testit suoritetaan järjestelmässä, joka toimii mahdollisimman realistisessa ympäristössä. Makrotesteistä saatavat tulokset ovat luotettavimpia, kun halutaan tietoa järjestelmän aidosta suorituskyvystä. Oaks tiivistää asian lauseeseen: *"Kokonainen järjestelmä on enemmän kuin sen osien summa."* [5]

Diplomityön käsittelemässä ohjelmistoprojektissa on toteutettu mikro- ja mesotestejä, joilla testataan järjestelmän oikeellisuutta, mutta suorituskykytestauksen automatisoinnissa haluttiin keskittyä makrotestaukseen. Automatisoinnin on tarkoitus korvata manuaalinen suorituskykytestaus, joten tulosten on tärkeää vastata manuaalista testausta, joka tehdään aina koko järjestelmällä.

2.2 Suorituskyvyn mittaaminen

Suorituskykyä voidaan mitata useilla eri tavoilla. Yleisiä mittauksia ovat vasteaika (response time), läpisyöttö (throughput) ja resurssien käyttö [5, 9-11]. Juuri näitä mittareita on hyödynnetty mukaillen myös tämän työn mittauksissa.

Suorituskykymittauksissa vasteajalla tarkoitetaan aikaa, joka järjestelmällä kuluu syötteen saannista siihen, kun järjestelmä saa tuotettua vasteen [11]. Vasteaikaa mitataan ajastamalla aika syötteen ja sen vasteen välillä.

Ajastus voidaan tehdä sovelluksen sisäisesti esimerkiksi lisäämällä koodissa syötteen prosessoinnin alkuun kellonajan tallennus, jota verrataan hetkeen, jolloin vaste on saatu käsiteltyä. Monissa testiautomatisointityökaluissa tämä on mahdollista tehdä ulkoisesti. Testityökalu mittaa ajan joka kului siitä, kun se antoi järjestelmälle syötteen siihen, kun se tunnisti järjestelmän antaneen vasteen syötteeseen.

Suorituskykymittauksessa läpisyötöllä tarkoitetaan operaatioiden määrää, jonka järjestelmä prosessoi tietyssä ajassa. Palvelimen tapauksessa läpisyötön mittarina voi olla, kuinka monta pyyntöä palvelin prosessoi sekunnissa. Prosessorin läpisyöttöä taas mitataan yleensä miljoonina käskyinä sekunnissa ja muistin läpisyöttöä tavuina sekunnissa. [11]

Sovelluksen läpisyöttöä voidaan mitata sovelluksen sisäisesti lisäämällä operaatioiden käsittelijään laskuri, jota kasvatetaan aina, kun operaatio on käsitelty. Laskurin summan avulla voidaan laskea esimerkiksi läpisyöttö sekunnissa.

Resurssien käytöllä tarkoitetaan esimerkiksi sovelluksen käyttämää muistin määrää tai sovelluksen aiheuttamaa prosessorikuormaa. Resursseja voidaan mitata absoluuttisina arvoina, jolloin muistinkäyttö ilmaistaan esimerkiksi megatavuina, tai prosenttiosuuksina

sovellukselle käytössä olevista resursseista, jolloin prosessorin käyttöasteen voitaisiin todeta olevan esimerkiksi 75 prosenttia. Resurssien käytön tarkkailuun voidaan käyttää käyttöjärjestelmän työkaluja, kuten Windowsin resurssienvalvontaa tai Linuxin topia, virtuaalikoneen työkaluja, kuten Javan Visual VM:ää. On myös mahdollista hyödyntää kolmannen osapuolen työkaluja, kuten tämän työn mittauksissa käytettyä TeamQuestia.

On hyvä huomata, että vasteajan kohdalla pienempi luku on yleisesti parempi, koska tämä tarkoittaa käyttäjän saavan vasteen syötteeseensä nopeammin [11]. Jakob Nielsen huomauttaa kuitenkin kirjassaan [12], että tietyissä tapauksissa liian pieni vasteaika voi aiheuttaa huonon käyttökokemuksen käyttäjälle. Tämä diplomityö käsittelee suorituskykyä, eikä ota kantaa liian nopeiden vasteaikojen käyttömukavuuteen, joten pienemmän vasteajan on katsottu olevan aina parempi. Läpisyötössä suurempi luku on parempi, koska järjestelmä on tehokkaampi, mitä enemmän operaatioita se pystyy suorittamaan tietyssä ajassa [11]. Resurssien käyttöasteesta ei voida antaa yksiselitteistä vastausta. Matala käyttöaste tarkoittaa, että järjestelmällä on resursseja, joita ei käytetä ja ne menevät näin hukkaan. Korkea käyttöaste voi johtaa hitaampiin vasteaikoihin. Resurssien käyttöasteessa keskivaiheen arvot ovatkin toivottuja: resursseja hyödynnetään, mutta ne eivät ole loppumaisillaan [11].

Käyttöliittymän suorituskykymittauksessa on hyvä huomioida, että käyttöliittymän tarkoitus on palvella ihmistä. Tällöin suorituskyvyn tulee mukautua ihmisen tarpeisiin. Moni teos [12-16] on viimeisen 50 vuoden aikana käsitellyt, kuinka ihminen kokee käyttöliittymän viiveen. Teoksien yleinen konsensus on, että ihminen tuntee käyttöliittymän vastaavan viiveettä, jos vasteaika on alle 100 millisekuntia. Mainituista teoksista uusimmassa [16] Doherty ja Sorenson vetävät tämän rajan 300 millisekuntiin. 100-300 millisekunnin pisteen jälkeen teoksissa kerrotaan ihmisen havaitsevan viiveen, mutta tuntevan operaation tapahtuvan välittömästi, jos vasteaika on alle sekunnin. Teoksissa katsotaan, että sekunnin ja kymmenen sekunnin välinen viive häiritsee käyttäjää, mutta käyttäjä sietää odotuksen. Sekunnin ja kymmenen sekunnin välisten vasteaikojen operaatioissa neuvotaankin antamaan latauksen aikana käyttäjäpalaute, kuten tiimalasi, jotta käyttäjä ymmärtää sovelluksen suorittavan operaatiota. Yli kymmenen sekunnin vasteaikojen kerrotaan aiheuttavan käyttäjän kiinnostuksen herpaantumisen, jolloin käyttäjä haluaa tehdä muita asioita odottaessaan. Tässä työssä testatut ominaisuudet on tarkoitettu tapahtuviksi välittömästi. Niiden yhteydessä ei anneta väliaikaista käyttäjäpalautetta.

Suorituskykymittaukseen vaikuttaa käytännössä kaikki mahdollinen. Tällä tarkoitetaan sitä, että jokainen resurssi voi periaatteessa loppua ja aiheuttaa suorituskykyongelman. Suorituskykytestauksessa onkin tärkeämpää yrittää huomioida erilaiset vaikuttavat tekijät kuin yrittää sulkea ne kaikki pois. On helpompi mitata häiriötekijöitä kuin estää niitä. Häiriötekijät ovat olemassa loppukäyttäjien maailmassa. Jos ne rajataan testeistä pois, eivät testit vastaa todellisuutta. Esimerkiksi käyttöjärjestelmän sisäiset prosessit ovat osa testausympäristöä ja osa reaalimaailmaa. Prosessit voivat vaikuttaa järjestelmän suorituskykymittaukseen, mutta ne vaikuttavat myös tuotantoympäristön sovellukseen, joten on

tärkeämpää tietää niiden vaikutus kuin sulkea niitä pois testeistä. Jos niiden vaikutus on niin suuri, että ne haittaavat sovelluksen suoritusta, voidaan niiden käyttö estää tuotanto- ja suorituskykytestiympäristössä. [5]

2.3 Java suorituskykytestauksessa

Testattava järjestelmä on toteutettu Javalla. Java on käännettävä ohjelmointikieli, mikä tarkoittaa, että Java-sovelluksen koodi on käännettävä ennen suorittamista. Java-sovellukset suoritetaan Javan virtuaalikoneella. Java-virtuaalikone on toteutettu useille eri alustoille, mutta ne jakavat yhteisen määritelmän, joka kuvaa miten virtuaalikoneen tulee toimia. Koska Java-virtuaalikoneet toimivat saman määritelmän mukaisesti alustasta riippumatta, Javalla toteutetut sovellukset ovat alustariippumattomia. Java-sovelluksen voi toteuttaa, kääntää ja testata yhdellä käyttöjärjestelmällä ja käännetty paketti on suoritettavissa millä tahansa muulla käyttöjärjestelmällä, johon on toteutettu Java-virtuaalikone. Javassa on useita ominaisuuksia, jotka on hyvä ottaa huomioon suorituskykytestauksessa. [17, 18]

Javaa kehittää yritys nimeltä Oracle. Oraclella on oma virtuaalikonetoteutus nimeltä Oracle HotSpot Java Virtual Machine. Javan virtuaalikoneesta on myös useita muita toteutuksia, jotka toteuttavat saman määritelmän ja jotka on testattu yhteensopivaksi alkuperäisen version kanssa [5]. Tämän diplomityön käsittelemässä ohjelmistoprojektissa käytetään Oraclen toteuttamaa virtuaalikonetta ja tämä työ keskittyy sisällössään pelkkään Oracle HotSpot Java -virtuaalikoneeseen.

Prosessoreilla on käskykanta, jota ne ymmärtävät [19]. Staattisesti käännettävissä kielissä, kuten C-ohjelmointikieli, koodi käännetään suoraan prosessorispesifisiksi käskyiksi. Tämä johtaa siihen, että koodi on kääntämisen jälkeen ajettavissa vain prosessoreilla, jotka ymmärtävät kyseiset käskyt. On myös tulkattavia ohjelmointikieliä kuten Python. Tulkattavissa ohjelmointikielissä lähdekoodi tulkataan tulkkisovelluksella konekieleksi tai tavukoodiksi ajoaikaisesti. Näillä kielillä toteutetut sovellukset ovat ajettavissa millä tahansa järjestelmällä, johon tulkkisovellus on toteutettu. Koska käännetyt kielet voivat optimoida käännettävää sovellusta kokonaisuutena ja tulkattavat kielet suorittavat koodia periaatteessa käsky kerrallaan, ovat käännettävät kielet lähtökohtaisesti parempia suorituskyvyltään kuin tulkattavat kielet. [20]

Vaikka Javakin on periaatteessa käännettävä kieli, sen kääntöprosessi eroaa staattisesti käännettävistä kielistä. Java voidaan nähdä tulkatun ja käännettävän kielen välimuotona. Java-kääntäjä kääntää koodin Java-tavukoodiksi, jota Javan virtuaalikone osaa suorittaa. Virtuaalikone voi suorittaa tavukoodia suoraan, kuten tulkki, tai kääntää tavukoodin edelleen prosessorikäskyiksi, kuten käännettävissä kielissä. Virtuaalikone oppii ajoaikaisesti, miten tulkattuna ajettu tavukoodi käyttäytyy ja pystyy optimoimaan tavukoodia ennen sen kääntämistä konekäskyiksi. [5]

Eräajomittaus on hyvä esimerkki siitä, kuinka Javan suoritusprosessi vaikuttaa suorituskykymittauksiin. Eräajomittauksessa toistetaan tiettyä operaatiota tuhansia kertoja ja mitataan kulunut aika. Jakamalla kulunut aika operaatioiden määrällä voidaan laskea yhden operaation suoritukseen kulunut aika. Koska Java optimoi koodia ajoaikaisesti, on sovelluksen suoritus nopeampaa eräajon lopussa kuin heti käynnistyksessä. Tästä syystä Javan suorituskykytesteissä tulee ymmärtää, että virtuaalikone tarvitsee lämpiämisjakson, jonka aikana se optimoi itse suorituskykyään. Jos ajatellaan teoreettista mikrosuorituskykytestiä, joka mittaa yhden metodin suorituksessa kuluvaa aikaa keskiarvona miljoonasta kutsusta, on syytä ensin ajaa metodia ensin esimerkiksi kymmenen tuhatta kertaa ja aloittaa ajan mittaus sadalle tuhannelle kutsulle vasta tämän jälkeen. Näin virtuaalikone ehtii optimoimaan metodin suorituksen ennen kuin itse mittaus aloitetaan. [5]

3. TESTATTAVA JÄRJESTELMÄ JA YMPÄRISTÖ

Tässä luvussa kerrotaan ohjelmistoprojektista, jonka osana tämä työ on toteutettu. Jatkossa kyseiseen ohjelmistoprojektiin viitataan usein pelkällä sanalla projekti. Luvussa kuvataan projektin tuote, eli hajautettu johtamisjärjestelmä, ja sen asettamat erityistarpeet suorituskyvylle. Luvussa kuvataan projektin keskeiset teknologiat ja käsitellään niiden vaikutusta suorituskykytestaukseen. Lisäksi selitetään, kuinka järjestelmää ja sen suorituskykyä testataan.

3.1 Hajautettu johtamisjärjestelmä

Työ toteutetaan osana hajautetun johtamisjärjestelmän kehitystä. Johtamisjärjestelmää käsitellään työssä hyvin korkealla tasolla. Toteutusyksityiskohdat eivät ole suorituskykytestauksen kannalta olennaisia. Järjestelmän kokoa kuvaa se, että järjestelmää on kehitetty vuosia ja se työllistää täyspäiväisesti kymmeniä ihmisiä. Alle on kerätty muutamia hyvin tätä projektia ja sen tavoitteita kuvaavia otoksia yrityksen verkkosivuilta.

”Johto- ja operaatiokeskustason johtamisjärjestelmät tukevat suunnittelua, toimeenpanoa ja reaaliaikaista johtamista puolustushaarakohtaisissa ja yhteisoperaatioissa. Korkea suorituskyky, taistelunkestävyys ja luotettava tietoturva yhdistettynä reaaliaikaiseen tiedon hajauttamiseen tarjoavat ihanteelliset olosuhteet laajamittaisen tilannekuvan muodostamiselle ja ilmapuolustuksen johtamiselle. Kaikki järjestelmän sisältämä tieto, mukaan lukien tunnistettu tilannekuva, on hajautettavissa kaikkiin järjestelmän solmuihin. Lisäksi johtamisjärjestelmän sisällä voidaan luoda reaaliaikaisia tilannekuvia erityistarpeisiin.” [21]

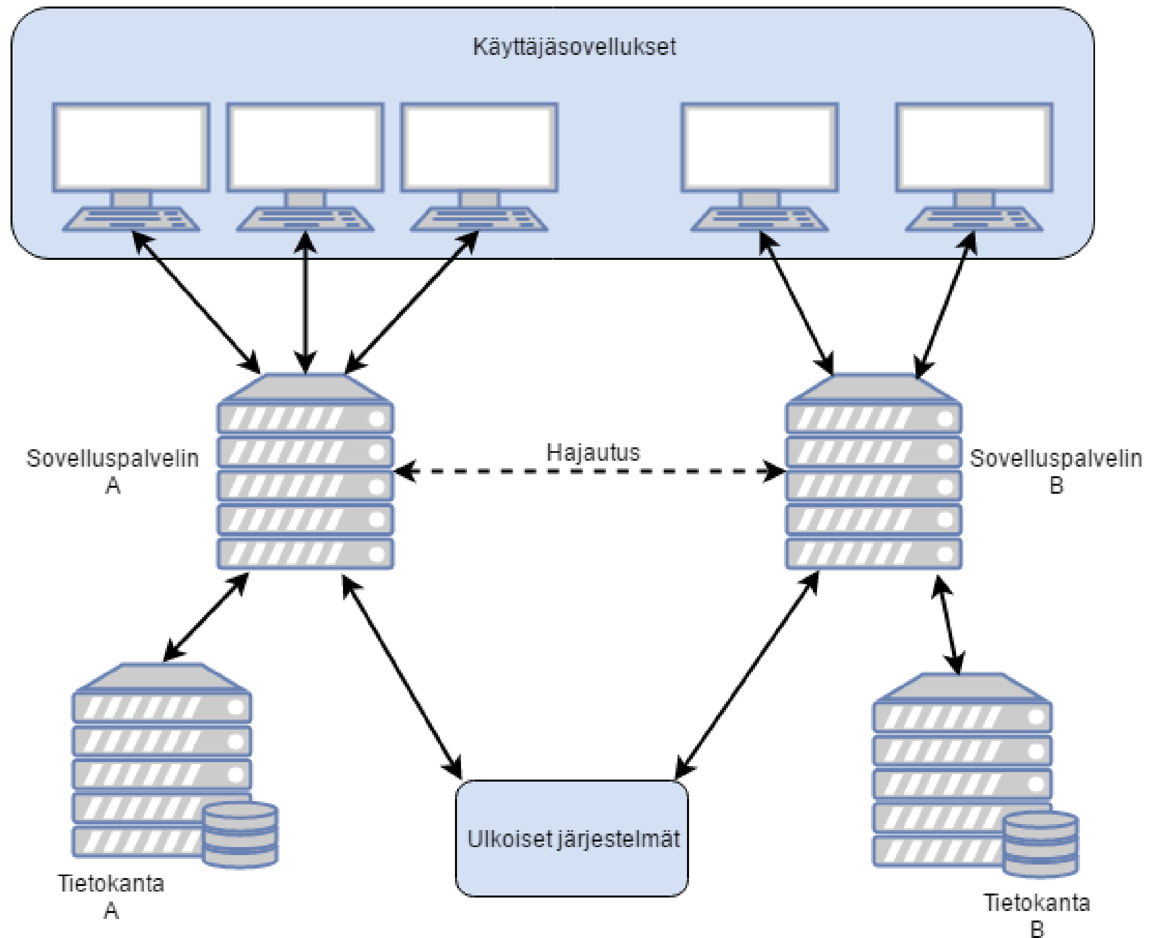
”Koska tilannetieto voi vanhentua muutamissa sekunneissa, on järjestelmän tärkein yksittäinen ominaisuus ajantasaisen tilannetiedon tarjoaminen kentällä oleville joukoille.” [21]

”Tilannetieto saadaan älykkään sanomakäsittelyn avulla siirrettyä reaaliaikaisena hitaista tietoliikenneyhteyksistä huolimatta.” [21]

Näistä lainauksista voi hyvin päätellä, että suorituskyky on olennainen osa testattavaa järjestelmää. Järjestelmälle tärkeitä arvoja ovat tiedon lähes reaaliaikainen käsittely, katkeamaton käyttövalmius sekä vian ja kuorman sietokyky. Järjestelmän tulisi olla aina käyttövalmiudessa ja tiedon välittyä käyttäjille sekunneissa.

Kuva 1 havainnollistaa järjestelmän osia ja niiden välisiä yhteyksiä. Järjestelmä jakaantuu sovelluspalvelimiin, käyttäjäsovelluksiin ja tietokantoihin. Käyttäjäsovellukset ovat verkon yli yhteydessä sovelluspalvelimeen. Sovelluspalvelimet ovat yhteydessä ulkoisiin

järjestelmiin ja hajautustilassa toisiin samanlaisiin sovelluspalvelimiin. Sovelluspalvelimet käyttävät tietokantaa tilannetietojen tallentamiseen ja vanhojen tietojen noutamiseen.



Kuva 1: Johtamisjärjestelmä ja sen osien väliset yhteydet

Käyttäjät voivat itse syöttää tilannetietoja käyttäjäsovelluksesta, jonka lisäksi järjestelmä saa tilannetietoja ulkoisista järjestelmistä ja toisilta käyttäjiltä. Sovelluspalvelin yhdistää saamansa tilannetiedot ja lähettää ne käyttäjäsovelluksille, jotka puolestaan esittävät tiedon käyttäjälle. Tätä yhdisteltyä tietokokonaisuutta kutsutaan tilannekuvaksi. Järjestelmän tilannekuva voidaan mieltää karttana, jonka päällä esitetään mahdollisimman reaaliaikaisia tietoja erilaisina tietosisältöisinä karttaobjekteina. Operaattori voi valita ja muokata karttaobjekteja ja tehdä tyypillisiä karttaoperaatioita, kuten tarkentaa karttanäkymää ja mitata etäisyyksiä.

3.2 Teknologiat

Järjestelmälle on tärkeää olla alustariippumaton, koska käytössä on erilaisia ympäristöjä ja järjestelmällä on pitkä kehitys- ja elinkaari, jonka aikana alusta voi muuttua. Tämä on vaikuttanut päätökseen valita ohjelmointikieleksi Java, joka on alustariippumaton [22]. Muutkin Javan suunnitteluperiaatteet, kuten suorituskyky, luotettavuus ja tietoturva [22],

ovat hyvin linjassa johtamisjärjestelmien vaatimuksien kanssa [21]. Javaa ja sen vaikutusta suorituskykytestaukseen käsiteltiin aliluvussa 2.3.

Sovelluspalvelimena toimii Wildfly, johon on asennettu OSGi-standardin [23] toteuttava alijärjestelmä. Nämä yhdessä mahdollistavat yksittäisten palvelujen käynnistämisen ja pysäyttämisen sekä päivittämisen lennosta [23], jotka ovat tarpeen järjestelmässä, jossa tavoitellaan mahdollisimman vähäisiä käyttökatkoja.

Verkon yli viestintä tapahtuu pääasiassa Data Distribution Service (DDS) -standardin [24] toteuttavalla erillisellä sovelluksella (OpenSplice DDS), jota hallitaan Java-kirjaston kautta. DDS-standardi on suunniteltu luotettavaksi ja suorituskykyiseksi julkaisija-tilaaja-mallin tietoliitynnäksi [24].

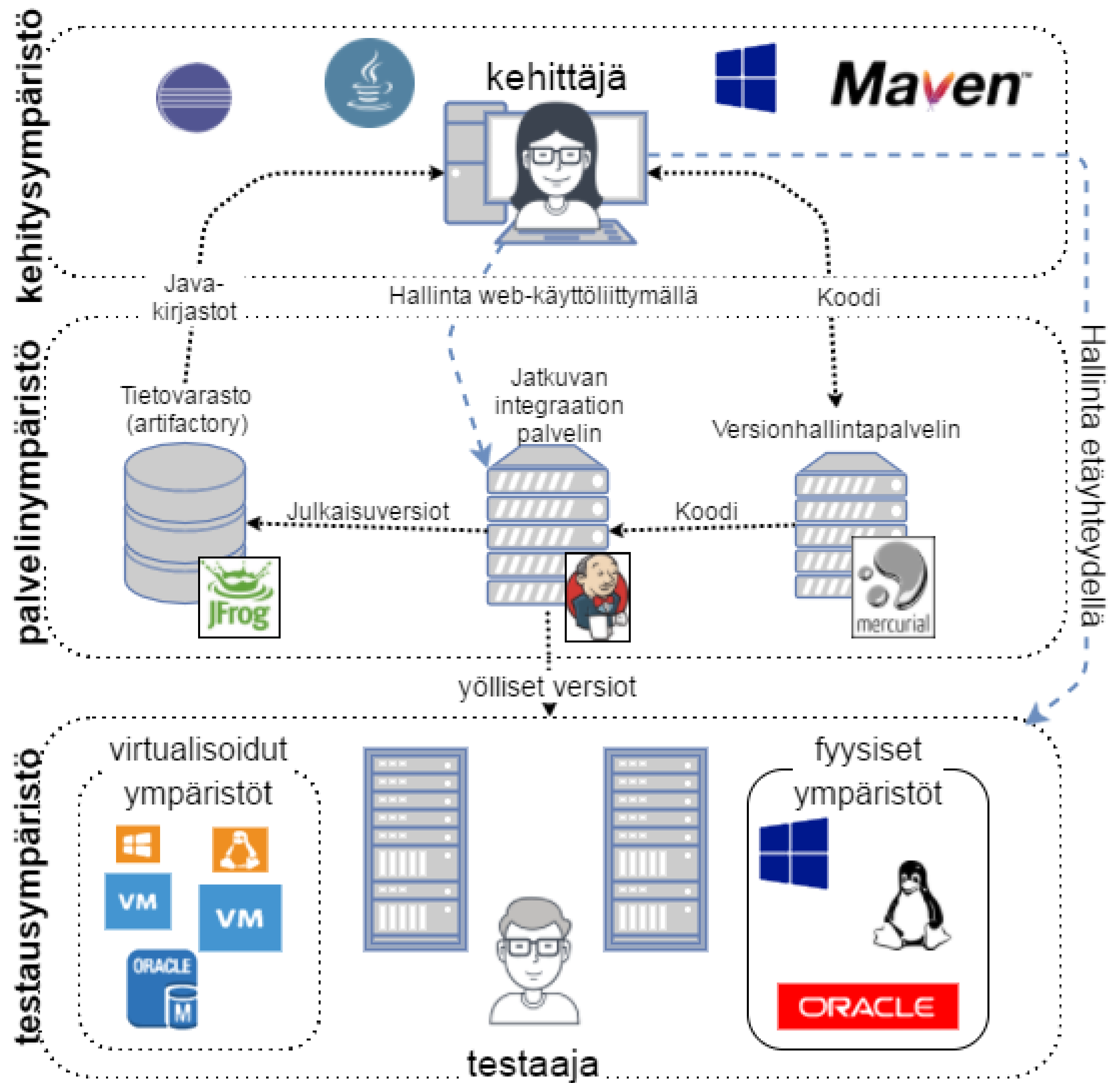
3.3 Ympäristöt

Projektin kehitys ja testaus jakautuvat kahteen ympäristöön. Projektin kehittämistä varten jokaisella kehittäjällä on henkilökohtaisella tietokoneellaan asennettuna ympäristö, jossa he voivat testata tuotoksiaan. Testausta varten on toinen ympäristö, jonka on tarkoitus jäljitellä tuotantoympäristöä. Lisäksi yhdeksi ympäristöksi voidaan mieltää palvelinympäristö, johon varastoidaan tietoa ja joka hoitaa automatisoituja tehtäviä, kuten versioiden tuottamisen. Kuvassa 2 on havainnollistettu ympäristöjen tehtäviä ja niiden välisiä yhteyksiä.

Kehitysympäristössä sovelluspalvelin ja käyttäjäsovellus pyörivät samalla tietokoneella Windows-käyttöjärjestelmässä. Testausympäristössä on useita virtuaalisia ja fyysisiä sovelluspalvelin- ja käyttäjäsovelluskoneita. Sovelluspalvelimet ajetaan Linux-käyttöjärjestelmässä ja käyttäjäsovellukset Windows-käyttöjärjestelmässä.

Jatkuva integraatio on tärkeä osa projektia. Jatkuvalla integraatiolla tarkoitetaan periaatetta, jossa versionhallintaan luodaan päähaara, johon kehittäjät liittävät omat tuotoksensa mahdollisimman usein. Tästä päähaarasta luodaan joka yö versio, jolle suoritetaan yksikkö- ja automaattitestit. Testit eivät ole kattavia, vaan niiden tarkoitus on varmistaa, että järjestelmä kääntyy ja käynnistyy. Testeistä kerrotaan lisää aliluvussa 3.4.

Projektin versionhallintatyökaluna toimii Mercurial. Mercurial perustuu haaroittamiseen ja haarojen liittämiseen [25]. Käsiteltävässä ohjelmistoprojektissa kehittäjät tekevät ominaisuuksia omissa kehityshaaroissaan ja liittävät ne päähaaraan, kun ominaisuus on kehittäjän mielestä valmis.



Kuva 2: Ympäristöt ja kehityksen kulku

Jatkuvan integraation automatisointipalvelimeksi on valittu Jenkins. Jenkins on avoimen lähdekoodin palvelinsovellus, jonka tehtävä on hoitaa automatisoitavia tehtäviä [26]. Tehtäviä on mahdollista määritellä vapaasti ja ajoittaa tarpeen mukaan [26]. Projektissa Jenkinsin ajoitettuihin tehtäviin kuuluu koodin kääntäminen, asennuspakettien tuottaminen testausympäristöön ja automaattitestien suoritus sekä tulosten keräys. Jenkins myös säilöö testitulokset, jotta niitä voidaan verrata myöhemmin. Jenkins ilmoittaa kehittäjille sähköpostilla, jos heidän tekemiensä muutosten päähaaraan liittämisen jälkeen kääntö tai automaattitestit epäonnistuvat. Lisäksi Jenkins toteuttaa muita automatisoituja tehtäviä, kuten luo tuotantoversioita, jotka talletetaan erilliselle tietovarastopalvelimelle (artifactory), sekä suorittaa koodin tyylitarkastuksen, jonka tulokset jaetaan web-palveluna kehittäjien tarkasteltavaksi.

3.4 Nykyiset testausmenetelmät

Projektin käytössä on monenlaisia testejä ja testausmenetelmiä. Aliluvuissa käydään läpi, miten projektia tällä hetkellä testataan, kerrotaan erilaiset testausmuodot ja otetaan kantaa, miten niillä on testattu suorituskykyä. Testaus painottuu selvästi ihmisen tekemään manuaaliseen testaukseen. Projektissa on päätetty siirtää vastuuta testaajilta automatisoiduille testeille, minkä seurauksena tämäkin työ on päätetty toteuttaa.

3.4.1 Yksikkötestit

Yksikkötesteillä tarkoitetaan pienen järjestelmän osan, kuten yksittäisen luokan, toimivuuden testausta ohjelmallisesti. Yksikkötestit ajetaan usein ja niiden tavoite on huomata virheet mahdollisimman aikaisessa vaiheessa [8]. Testattavaan järjestelmään luodaan yksikkötestejä osana normaalia kehitystä. Yksikkötestit tukevat kehitystä ja niitä luodaan järjestelmän kehityksen ohessa kehittäjien parhaan näkemyksen mukaan.

Projektissa yksikkötestien pääpaino on selvästi varmistaa tiedon oikeellisuus. Projektissa on toteutettu myös muutamia suorituskykyä mittaavia yksikkötestejä, mutta niiden tuottamat tulokset ovat useimmiten vain suuntaa antavia. Niissä ei ole huomioitu aliluvuissa 2.1 ja 2.3 mainittuja suorituskykytestaukselle olennaisia asioita, kuten Java-kääntäjän tekemä ajoaikainen optimointi.

Projektissa kehittäjien tulee ajaa yksikkötestit osana kehitystä ja ennen uusien ominaisuuksien tai korjausten liittämistä versionhallinnan päähaaraan. Yksikkötestit ajetaan myös osana jatkuvaa integraatiota. Jos yksikin testi epäonnistuu, ilmoitetaan siitä niille kehittäjille, jotka ovat muuttaneet koodia viimeisimmän onnistuneen testiajon jälkeen.

3.4.2 Automaattitestit

Projektissa käytetään sanaa ”automaattitesti” kuvaamaan automaattisesti suoritettavia testejä, jotka testaavat suurempia kokonaisuuksia kuin yksikkötestit. Tästä käytetään usein myös sanaa integraatiotestaus [8]. Projektissa kehittäjät eivät yleensä aja automaattitestejä, vaan ne on tarkoitettu jatkuvan integraation palvelimen ajettaviksi. Osana jatkuvaa integrointia automaattitestit ajetaan yöllisten versioiden luonnin yhteydessä.

Projektissa automaattitestejä on kahta eri alalajia. Testejä, jotka toimivat yksikkötestien tapaan, mutta niissä testataan melko suuria kokonaisuuksia. Nämä testit on koodattu Javalla ja suoritetaan osana käännösprosessia. Toinen alalaji on testit, jotka suoritetaan käännöksen valmistuttua koko järjestelmällä. Aikaisemmin näitä testejä on ollut vain yksi, jossa testattiin käynnistyvätkö palvelin ja käyttöliittymä, ja saavatko ne yhteyden toisiinsa. Tänä vuonna (2017) koko järjestelmän automaattitestejä on alettu tuottaa lisää. Testeissä käynnistetään koko järjestelmä, suoritetaan käyttöliittymäoperaatioita ja varmennetaan operaatioiden tuottavan toivotut tulokset koko järjestelmässä.

Kuten yksikkötestit, myös projektin automaattitestit keskittyvät tällä hetkellä toiminnallisuuden testaamiseen. Automaattitesteillä varmistetaan, että järjestelmän erilliset moduulit toimivat yhdessä. Automaattitestejä kehitetään harkitusti ja ne keskittyvät vahvasti sovelluspalvelimen palvelujen testaamiseen. Uusien koko järjestelmää testaavien testien on tarkoitus lisätä automaattitestausta myös käyttäjäsovelluksen puolella.

3.4.3 Manuaalinen testaus

Projektissa on monia kokopäiväisiä testaajia. Testaus on pääsääntöisesti tarkasti suunniteltua. Testattavilla kokonaisuuksille luodaan testitapaukset, jotka pohjautuvat asiakasvaatimuksiin. Kehittäjät tarkastavat testitapaukset ennen niiden suorittamista. Testaajat suorittavat testitapaukset ja testin onnistuminen arvostellaan asteikolla hylätty tai hyväksytty. Jos testitapaus on hylätty, havaituista vioista tehdään vikaraportti. Testitapaus voidaan hyväksyä vasta, kun kaikki siihen liittyvät vikaraportit on korjattu ja testattu. Testaustyö jakaantuu tuote-, järjestelmä- ja suorituskykytestaukseen sekä vikakorjausten todentamiseen.

Tuotetestaus tarkoittaa tietyn vaatimuksen täyttävän ominaisuuden kokonaisvaltaista testausta. Vaatimus voi olla esimerkiksi se, että saatu tilannetieto tulee esittää käyttäjälle. Tuotetestauksessa todennetaan, että tieto todellakin esitetään käyttäjälle. Lisäksi testataan ominaisuuden käytettävyyttä, suorituskykyä ja kestääkö ominaisuus virhekäyttöä. Virhekäyttö liittyy interaktiivisiin käyttöliittymäelementteihin, joissa käyttäjä voi käyttää ominaisuutta toisin kuin on suunniteltu. Näissä tilanteissa on tärkeää, että ominaisuudella ei voida aiheuttaa koko järjestelmään vikatilaa. Myös ominaisuuteen liittyvä suorituskyky huomioidaan tuotetestauksessa. Jos esimerkkitapauksessa tiedon esittäminen käyttäjälle kestäisi kohtuuttoman kauan, aiheuttaisi tämäkin testitapauksen hylkäämisen. Tuotetestauksesta ominaisuus voidaan palauttaa kehittäjälle, jos siinä huomataan puutteita. Tässä tapauksessa testaaja kirjaa vikaraportin, jonka lähettää kehittäjälle. Testaajat yleensä neuvottelevat kehittäjän kanssa mahdollisista parannuksista. Tuotetestaus suoritetaan aina mahdollisimman nopeasti sen jälkeen, kun kehitystyö on tehty, jotta saadaan nopeaa palautetta kehittäjälle.

Järjestelmätestauksessa testataan nimensä mukaisesti koko järjestelmää tai vähintään useamman ison komponentin toimintaa yhdessä. Suunnitellaan skenaarioita, jotka käyttävät komponentteja todellisen käytön mukaisesti, ja toteutetaan ne. Havainnoista kirjataan muistio ja mahdollisista vioista vikaraportti. Järjestelmätestejä ei suoriteta jatkuvasti, koska niiden suunnittelu ja suorittaminen vaativat paljon aikaa testaajilta. Järjestelmätestejä suoritetaan vain muutaman kerran vuodessa. Tiheä järjestelmätestaus ei ole mielekästä, koska järjestelmätestiversion halutaan olevan vakaa ja jatkuva kehitys aiheuttaa väistämättä epävakautta. Järjestelmätestejä suoritetaankin vain jäädytetyillä versioilla, joihin ei liitetä uusia ominaisuuksia, ja joiden kaikki ominaisuudet on jo tuotetestattu ja mahdolliset viat korjattu.

Projektin suorituskykytestaus on hyvin samanhenkistä kuin järjestelmätestaus. Siihen on määritelty tarkat skenaariot, joita seurataan aina testejä suoritettaessa. Testit on jaettu kolmeen skenaarioon rasituksen perusteella: kevyt, keskiraskas ja raskas. Järjestelmää rasitetaan generaattoreilla, jotka jäljittelevät ulkoisia järjestelmiä ja tuottavat tilannetietoja järjestelmään. Skenaarioita testataan passiivisella ja aktiivisella testitapauksella. Passiivisissa skenaarioissa ei suoriteta käyttäjäoperaatioita. Järjestelmä vain käsittelee sille tuotetut tilannetiedot. Aktiivisessa testitapauksessa testaaja operoi käyttäjäsovelluksella sen ollessa rasituksessa. Aktiivinen testitapaus on suoritettu vain raskaalla skenaariolla, koska järjestelmän on oltava käyttökelpoinen raskaallakin kuormalla. Testaaja kirjaa testitapausten ajalta keskiarvoina ylös muistin käytön megatavuina, prosessorin käyttöprosentin sekä mitatun tiedon läpivientiajan järjestelmässä. Lisäksi testaaja kirjaa muut havainnot testauksesta, kuten käyttöliittymän vasteen.

Kuten aliluvussa 3.3 kerrottiin, testausympäristöistä osa on virtualisoituja, mutta osa koostuu fyysisistä koneista. Tuote- ja järjestelmätestaus keskittyvät virtualisoituihin ympäristöihin ja fyysiset ympäristöt on tarkoitettu erikoistarkoituksiin. Jos vian syy on vaikea paikantaa, saatetaan se toistaa fyysisessä ympäristössä, jotta voidaan sulkea pois virtualisointi vian aiheuttajana. Suorituskykytestaus on eristetty yhteen fyysiseen koneeseen, koska havaittiin, että virtualikoneet syövät toistensa resursseja. Tällä tarkoitetaan, että yhdellä virtuaalikoneella tehtävä raskas prosessi, kuten version asennus, hidastaa toisen samassa fyysisessä koneessa pyörivän virtuaalikoneen toimintaa.

4. NYKYISTEN SUORITUSKYKYTESTIEN ONGELMAT

Suorituskykytestien automatisointi on iso rupeama, joten on syytä tietää mihin ongelmiin sillä haetaan ratkaisuja. Tämä luku käsittelee ongelmia, joita havaittiin manuaalisessa suorituskykytestausprosessissa. Osa havaituista ongelmista johti tämän työn aloittamiseen, osa todettiin työtä valmistellessa ja osa on havaittu työtä toteuttaessa. Työn tulevissa luvuissa palataan näihin ongelmiin ja esitellään, miten ne on ratkaistu.

4.1 Hidas palaute

Suurin yksittäinen ongelma, joka johti tämän työn aloittamiseen, oli hidas palaute. Tällä tarkoitetaan sitä, että projektissa havaittiin suorituskykyongelmat viikkoja tai jopa kuukausia sen jälkeen, kun ne oli aiheutettu. Tämä johti siihen, että kehittäjät olivat ehtineet tekemään satoja koodimuutoksia, jotka oli liitetty versionhallinnassa osaksi päähaaraa, josta yölliset versiot tehdään. Suorituskyvyn heikentymistä etsiessä saatettiin joutua käymään läpi jopa kymmeniä yöllisiä versioita.

4.2 Inhimillinen tekijä

Ihmisen tekemänä suorituskykytestaukseen tulee useita varianssia lisääviä tekijöitä. Tämä varianssi vaikuttaa tuloksiin ja tekee niistä, jos nyt ei vertailukelvottomia, niin ainakin niiden vertailusta vähemmän luotettavaa.

Projektin suorituskykytestauksen valmisteluohje on useamman sivun mittainen. Valmisteluun kuuluu monien asetustiedostojen muuttaminen sekä monien erilaisten työkalujen ja generaattorien käynnistely. Koska ihminen toteuttaa nämä, on mahdollista, että jokin vaihe unohtuu tai menee väärin. Virheellinen valmistelu ei välttämättä estä testien suorittamista, vaan se voi jopa jäädä huomaamatta. Tällöin ei tiedetä tulosten olevan vertailukelvottomia.

Osassa testeistä käyttäjä tekee käyttöliittymäoperaatioita, mutta käyttäjän ei ole mahdollista toistaa tarkalleen samoja operaatioita vaan ajoitukset, hiiren liikkeet ja vastaavat ovat väistämättä erilaisia suorituskertojen välillä.

Kuten aliluvussa 3.4.3 mainittiin, testaaja kirjaa suorituskykytestien jälkeen mielipiteensä käyttöliittymän vasteesta. Tämä on osaksi korvaamatonta, koska käyttökokemusta vastaavaa automatisoitua testiä on hankala toteuttaa. Samalla ihmisen käyttökokemus on aina

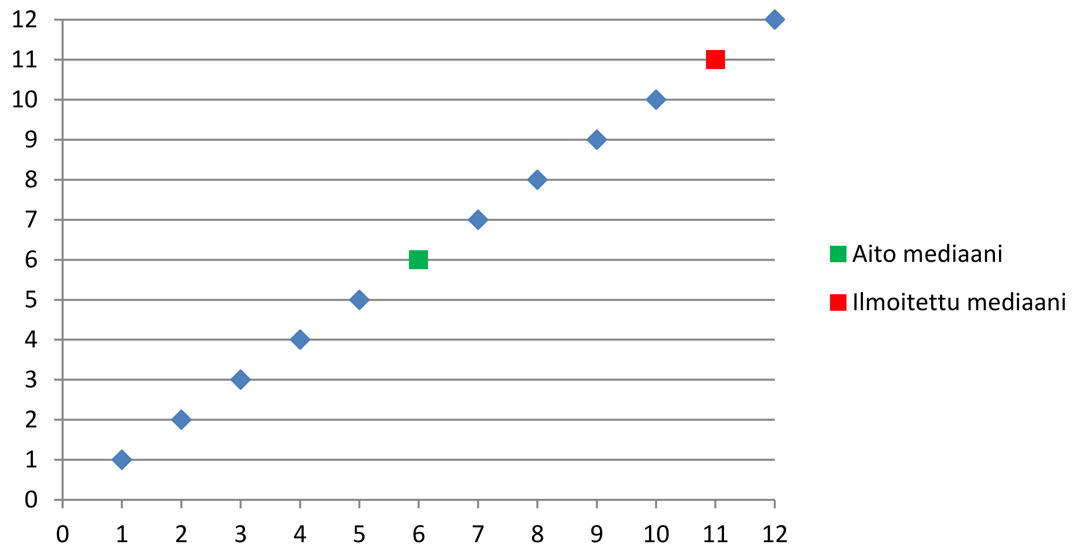
subjektiivista ja vaihtelee suurestikin. Yksi käyttäjä voi tuntea pienen viiveen huomattavasti käyttöä haittaavampana kuin toinen. Lisäksi samankin käyttäjän tuntemus voi muuttua ajan kuluessa ja tottumusten muuttuessa. Jos järjestelmää käyttää joka päivä vuoden ajan ja se hidastuu lineaarisesti ajan kuluessa, käyttäjä voi huomaamattaan alentaa odotuksiaan järjestelmän vasteajasta.

4.3 Tulosten tarkkuus

Suorituskykymittausten tulokset on yleensä kerätty keskiarvoina pitkän suorituskykyskenaarion ajalta. Valitettavasti tämä lähestymistapa tarjoaa hyvin vähän tietoa siitä, mikä järjestelmässä toimi hitaasti. On mahdollista, että 90 prosenttia skenaarion ajoajasta kaikki toimi moitteettomasti, mutta yhden kymmenyksen aikana järjestelmä oli täysin tukossa. Keskiarvosta emme pysty päättelemään, mikä ja missä vaiheessa meni pieleen. Samoin suuret ongelmat voivat jäädä jopa huomiotta, koska minuutin piikki puolen tunnin ajon aikana ei ole heilauttanut keskiarvoa tarpeeksi ylöspäin, että sitä olisi havaittu. Jos puolen tunnin ajosta otettaisiin minuutin välein keskiarvonäyte, voitaisiin näiden graafisesta esityksestä huomata heti jotain tapahtuneen.

Työn toteutuksen alkumetreillä käytiin läpi vanhoja mittareita ja järjestelmän tärkeimmän mittarin, läpimenoajan, mediaanin laskutavassa havaittiin virhe. Ensin mittari tuotti useasta näytteestä mediaanin, joka lähetettiin verkon yli tulosten keräilijälle. Tulosten esitysvaiheessa näistä mediaaneista laskettiin vielä mediaani. Tämä mediaani mediaaneista esitettiin käyttäjälle läpimenoajan mediaanina, jota se ei todellisuudessa ole.

Virheen mahdollinen suuruus on helppo havainnollistaa esimerkillä. Oletetaan, että näytekokoelma A sisälsi 10 näytettä, joiden arvot olivat 1, 2, … 10, ja näytekokoelmat B ja C sisälsivät kukin yhden näytteen arvoilla 11 ja 12. Nyt näytekokoelman A mediaani on 5 ja näytekokoelmien B ja C mediaanit ovat 11 ja 12. Käyttäjälle näytteiden mediaanina esitetään 11, vaikka näytteiden oikea mediaani on 6. Kuvassa 3 esimerkin luvuista on piirretty pistekaavio ja tähän on merkitty sekä aito mediaani että näytekokoelmien mediaanien mediaani. Tämä esimerkki on melko armoton ja järjestelmässä todennäköisempää on, että mediaani on osunut lähemmäs aitoa mediaania. On kuitenkin mahdotonta tietää kuinka lähelle, joten mediaanin osalta kerätyt tulokset voi katsoa saastuneen käyttökelvottomiksi.



Kuva 3: Kuvaaja, joka havainnollistaa mediaanin laskutoimituksen virheen suuruutta tekstin esimerkkitapauksessa.

4.4 Testidatan generointi

Suorituskykytesteissä tuotetaan kartalle objekteja erilaisten generaattorien avulla. Kaikille generaattoreille on voitu määrittää, kuinka monta objektia ne tuottavat ja kuinka usein kyseisiä objekteja päivitetään tiedoilla. Joissain generaattoreissa objektien tiedot on asetettu aina tietyiksi arvoiksi. Toisissa generaattoreissa on tuotettu tiedot käyttäen satunnaisgeneraattoria. Tämä on aiheuttanut varianssia testeihin ja sitä kautta myös tuloksiin.

4.5 Syyn jäljitys

Suorituskykytestausta ei pystytty virtuaalisessa ympäristössä eristämään, vaan muilla koneilla tehty kuorma heijastui suorituskykytestaukseen. Tästä syystä suorituskykytestaus päädyttiin aikanaan eristämään yhdelle fyysiselle tietokoneelle. Nyt samalla tietokoneella pyörivät käyttäjäsovellus, sovelluspalvelin ja tietokanta. Tämäkään ei ole ideaalitilanne, koska havaittaessa suorituskykyongelma, on hankalampi jäljittää hidastumisen syy, koska sovelluspalvelin, käyttöliittymä ja tietokanta voivat hidastaa toisiaan.

5. VAATIMUKSET JA RAJAUKSET TOTEUTETTAVALLE TESTAUSJÄRJESTELMÄLLE

On tärkeää asettaa työlle kriteerit, joihin työn lopputulosta voidaan verrata. Tässä luvussa kuvataan työlle asetetut vaatimukset. Lisäksi luvussa kerrotaan, mitä rajoituksia työlle asetettiin.

5.1 Vaatimukset

Työssä suunnitellulle testausjärjestelmälle asetettiin vaatimukset, joiden toteutuminen on työssä tärkeää. Järjestelmän vaatimukset asetettiin korkealla abstraktiotasolla, koska vaatimuksilla ei haluttu ohjata kehitystä väärille raiteille. Nähtiin tärkeämmäksi mahdollisuus adaptoitua ketterästi toteutuksen aikana kuin tehdä tarkka vaatimuslista ja seurata sitä. Näiden vaatimusten toteutumisella on suora yhteys työn onnistumiseen. Aliluvussa 7.2 pohditaan työn onnistumista peilaten toteutettua kokonaisuutta asetettuihin vaatimuksiin.

5.1.1 Automatisointi

Automatisointiin on viitattu tämän diplomityön johdannosta lähtien. Se oli itsestään selvä vaatimus järjestelmälle, eikä sitä edes kirjattu ylös työtä aloittaessa. Alusta lähtien oli kuitenkin selvää, että testausjärjestelmän tulisi toimia itsenäisesti.

Testausjärjestelmä halutaan toteuttaa siten, että se ei vaadi ihmiseltä toimenpiteitä suorittaakseen testit päivittäin. Tämä voidaan jakaa alivaatimuksiin. Jatkuvan integraation palvelimen tulee tuottaa testaukseen versio joka yö ja signaloida testausjärjestelmälle, että uusi versio on valmiina testaukseen. Testausjärjestelmän tulee pystyä asentamaan ja käynnistämään testattava järjestelmä itsenäisesti. Lisäksi testausjärjestelmän tulee käynnistää mittaustyökalut ja ajaa suunnitellut skenaariot järjestelmälle. Näistä ajoista tulee kerätä mittaustulokset ja tallettaa ne siten, että ihmisen on helppo tarkastella tuloksia.

5.1.2 Toistettavuus

Luvussa 4 käsiteltiin useita ongelmia mittaustulosten tarkkuudessa. Näiden johdosta on ollut hankala vertailla suorituskyvyn kehitystä. Automatisoidut skenaariot on suunniteltava siten, että samalla versiolla skenaarioita toistettaessa mittaustulosten vaihtelu on mahdollisimman vähäistä.

Inhimillinen tekijä poistuu testeistä automatisoinnin myötä. Lisäksi on syytä poistaa satunnaisuus tilannetietoa generoivista apusovelluksista. Mittareiden suunnittelussa on

huomioitava mahdolliset ulkoiset tekijät, kuten muut sovellukset järjestelmässä ja mittareiden aiheuttama lisäviive. Näitä ei kuitenkaan pyritä sulkemaan täysin pois.

5.1.3 Kehityksen kestävä

Kuten aiemmissa luvuissakin on todettu, testattavaa järjestelmää kehitetään edelleen. Siihen luodaan jatkuvasti uusia ominaisuuksia ja jo toteutetut ominaisuudet voivat muuttua. Samaan aikaan suorituskykytestauksen tuloksia halutaan verrata pitkillä aikaväleillä.

Järjestelmän kehitykseen on varauduttava suunnittelemalla skenaarioista niin yksinkertaiset, ettei jatkuva kehitys estä niiden toimintaa. Skenaarioihin toteutettava käyttöliittymäoperointi on keskitettävä ominaisuuksiin, joiden nähdään pysyvän osana järjestelmää koko sen elinkaaren ajan. Samoin ulkoisiin järjestelmiin pohjautuva tilannetieto keskitetään lähteisiin, joiden tiedetään säilyvän osana järjestelmää vuosia eteenpäin.

5.1.4 Laajennettavuus

Vaikka osana tätä työtä toteutetaan vain yksinkertaisia kehityksen kestäviä skenaarioita, itse testausjärjestelmän tulee olla helposti laajennettavissa. Testausjärjestelmään halutaan toteuttaa jatkossa testejä, jotka testaavat yksityiskohtaisempia ominaisuuksia. Näiden testiskenaarioiden ei ole tarpeen toimia kehityksen kestävästi. Skenaarioista saatujen mittaustulosten ei oleteta pysyvän vertailukelpoisina vuodesta toiseen.

Testausjärjestelmän suunnittelussa tulee huomioida mahdolliset laajennukset. Automatisointi toteutetaan siten, että siihen on helppo lisätä skenaarioita jatkossa. Osana laajennettavuutta on huomioitava myös ympäristön muutokset. Järjestelmää ei haluta sitoa tiettyyn laitteistoon. Järjestelmä tulisi olla helposti siirrettävissä virtuaaliympäristöön, jos tämä nähdään jatkossa tarpeelliseksi.

5.1.5 Käyttöliittymäviiveen mittari

Yksityiskohtaisin työlle asetettu vaatimus on käyttöliittymäviiveen mittari. Koska toteutuksella on tarkoitus korvata manuaalinen suorituskykytestaus, halutaan asettaa vaatimus, joka huomioi ihmisen tuoman käyttöliittymätuntuman arvioinnin.

Luvuissa 2.2 ja 4.2 käsiteltiin käyttäjän kokemaa viivettä. Manuaalisessa testauksen perusteella viivettä ei voida suoraan mitata mistään järjestelmän resurssista kuten prosessoriajasta tai muistinkulutuksesta. Tästä syystä viive on syytä mitata sovelluksen toiminnan osana, eikä arvioida viivettä järjestelmän resurssien käytöstä. Jotta viiveen mittariin voidaan luottaa, on se testattava vertaamalla ihmisen käyttökokemusta ja mittarin toimintaa.

5.2 Rajaukset

Työn suunnittelun alkuvaiheessa työlle asetettiin tietyt rajaukset, jotta työ etenisi suunnitellusti. Rajauksilla halutaan rajoittaa työmäärää ja keskittää työ toteuttamaan asetetut vaatimukset.

5.2.1 Aikataulu

Aikataulua voidaan kutsua vaatimukseksi tai rajaukseksi. Se on vaatimus siinä, että työ tulisi toteuttaa arvioidussa ajassa, ja rajaus siinä, että työn laajuus tulee sopeuttaa aikatauluun. Aikataulun perusteella työlle varataan budjetti ja työntekijöiden työaikaa. Työntekijät ovat rajattu resurssi projektille. Työntekijöiden aikaa tarvitaan myös muihin tehtäviin, joten on tärkeää rajata, kuinka paljon heidän aikaansa varataan tiettyyn työhön.

Työn alussa pidettiin palaveri, jossa esiteltiin korkealla tasolla testausjärjestelmä, joka haluttiin toteuttaa, ja tehtiin toteutuksesta työmääräarvio. Työmäärään ei laskettu tätä diplomityötä. Kolme kehittäjää antoi ja perusteli oman arvionsa eri vaiheiden työmäärästä. Arvioiden pohjalta tehtiin kokonaistyömääräarvio. Työ ja arvio esitettiin projektin johdolle, jotka näkivät testausjärjestelmän työmäärän arvoiseksi ja hyväksyivät toteutuksen. Näin työ rajattiin tiettyyn aikatauluun, jonka aikana työn kehitystä seurataan. Rajaus ei ole ehdoton, vaan työtä voidaan jatkaa, jos työn nähdään edenneen, mutta tarvitsevan lisää työtä.

5.2.2 Hajautus

Hajautus tuo haasteita ja lisää kompleksisuutta, joka vaikuttaa myös suorituskykyyn. Testausjärjestelmän monimutkaisuus lisääntyisi huomattavasti, jos testeihin lisättäisiin useita palvelimia ja käyttöliittymätoimipaikkoja. Tämä vaatisi samalla lisäinvestointeja laitteistoon, testiohjelmistojen lisensseihin ja testijärjestelmän ylläpitoon. Tällä hetkellä ei nähdä hajautuksen automaattiselle suorituskykytestaukselle niin suurta tarvetta, että edellä mainittuja investointeja voitaisiin oikeuttaa. Näistä syistä hajautus on päätetty jättää tämän työn ulkopuolelle. Kuten aiemmin tässä luvussa todettiin, laajennettavuudessa huomioidaan virtuaaliset ympäristöt, mikä mahdollistaa hajautuksen testaamisen tulevaisuudessa. Hajautus fyysisillä koneilla vaatii liikaa resursseja, mutta virtuaalisilla koneilla sen toteuttaminen olisi mahdollista.

5.2.3 Käyttäjän liikkeiden jäljittely

Testiskenaarioissa ei yritetä tarkasti jäljitellä ihmistä. On olemassa työkaluja, joilla voidaan nauhoittaa tarkasti ihmisen tekemät liikkeet käyttöliittymälle ja toistaa ne samalla tavalla [6]. Työn kannalta ei ole olennaista, että käyttöliittymäkäskyt jäljittelevät tarkasti

ihmistä. Nauhoitteiden pelätään käyvän helposti vanhoiksi, koska käyttöliittymää kehitetään jatkuvasti [6]. Jos nauhoite joudutaan korvaamaan, tulosten vertailukelpoisuus kärsii, koska nauhoitteet eroavat toisistaan. Työn skenaarioissa keskitytään tarkkoihin koneellisesti toistettuihin käyttäjäsyötteisiin, joita pystytään toistamaan sovelluksen kehittyessä.

6. SUORITUSKYKYTESTISKENAARIOIDEN SUUNNITTELU JA AUTOMATISOINTI

Luvussa käydään läpi valitut suorituskykyskenaariot ja arvioidaan niiden vaikutusta järjestelmän kuormitukseen. Luvussa kuvataan myös skenaarioiden suoritukseen valittu ympäristö ja toteutuksessa ja automatisoinnissa käytetyt työkalut. Luvun lopussa selitetään vielä, miten skenaarioiden suoritus liitettiin jatkuvaan integraatioon.

6.1 Skenaarioiden kuormitus

Skenaarioiden haluttiin perustuvan aliluvussa 3.4.3 kuvattuihin manuaalisiin suorituskykyskenaarioihin. Tästä syystä kuormitusmäärät pidettiin samoina kuin manuaalisissa suorituskykytesteissä. Kuormittajia, eli järjestelmään tietoja tuottavia generaattoreita, oli tarpeen muuttaa, koska osa niistä käytti satunnaislukuja. Satunnaisluvut haluttiin kitkeä pois skenaarioiden kuormituksesta, koska ne aiheuttivat samalla testiskenaarioihin satunnaisuutta, jota pyrittiin välttämään toistettavuuden parantamiseksi.

Manuaalisessa testauksessa tietojen tuottamiseen käytetään kolmea erillistä generaattoria (A, B ja C). Näistä B- ja C-generaattorit käyttivät satunnaislukuja tiedon tuottamiseen. Molemmat päätettiin korvata. A- ja B-generaattorien tuottamat tiedot vastasivat toisiaan, joten B-generaattori pystyttiin helposti korvaamaan kasvattamalla A-generaattorin tuottamaa kuormaa. C-generaattorin tuottamat tiedot jäljittelivät eri ulkoista järjestelmää kuin A, eikä tietojen tuottamista voitu siirtää A-generaattorin vastuulle. C-generaattorin korvaamiseen piti keksiä toinen ratkaisu.

C-generaattori lähettää kuormansa verkon yli järjestelmälle. Tästä syntyi idea nauhoittaa C-generaattorin tuottama verkkoliikenne tietyltä ajalta. Nauhoitus toteutettiin Linux-työkalulla Tcpreplay [27], jolla on mahdollista nauhoittaa verkkoliikennettä ja toistaa nauhoite. Koska nauhoite lähettää aina samat paketit verkkokortille, voidaan olla täysin varmoja, että C-generaattorin tuottama kuorma on joka toistokerralla identtinen. Nauhoitteita tehtiin tässä vaiheessa vain yksi kappale, koska haluttiin saada tuloksia nauhoitteen toiminnasta. Nauhoitteen kuormitustasoksi valittiin C-generaattorin raskas skenaario.

Nauhoitteen ja säilytetyn A-generaattorin asetukset ja käynnistys toteutettiin irrallaan muusta skenaariosta. Tämä mahdollistaa rasitustason määrittämisen skenaarioihin parametrilla, joka helpottaa skenaarioiden ajamista eri rasitustasoilla.

6.2 Skenaarioiden operaatiot

Skenaarioissa tulee suorittaa käyttäjäoperaatioita, jotta käyttöliittymän viivettä voidaan mitata. Operaatioiden tuli olla yksinkertaisia, jotta vaatimus kehityksen kestävyydestä toteutuu. Valittiin kolme yleistä käyttöliittymäoperaatiota, jotka eivät muutu järjestelmän kehittyessä.

Sen sijaan, että samassa skenaariossa toistettaisiin kaikkia operaatioita, ne eriytettiin omiksi käyttäjäsovellusskenaarioikseen. Tällä tarkoitetaan, että käyttäjäsovellus käynnistetään aina skenaarion alussa ja suljetaan skenaarion lopuksi. Skenaarion aikana toistetaan samaa operaatiota pidempi aika, jotta nähdään vaikuttaako operaation pitkäaikainen toistaminen sen suorituskykyisyyteen. Samalla nähdään, vaikuttaako mittaustuloksiin luvussa 2.3 käsitelty Javan tekemä ajoaikainen optimointi.

Ensimmäisenä on syytä käsitellä niin sanottu leposkenaario. Se perustuu suoraan manuaalisiin suorituskykytesteihin. Niissä suoritetaan aina skenaario, jossa käyttöliittymä käynnistetään, mutta sillä ei operoida, vaan käyttöliittymä vain esittää tilannetietoja käyttäjälle. Tällä saadaan mitatuksi, kuinka paljon tilannekuvan esittäminen yhdessä käyttäjäsovelluksessa vie resursseja sovelluspalvelin- ja käyttöliittymäkoneissa. Lisäksi skenaario antaa lähtötason, johon muita käyttöliittymäskenaarioita voidaan verrata. Tämä skenaario toteutettiin ensimmäisenä ja se antoi samalla hyvän rungon, jonka päälle rakentaa muita skenaarioita.

Itse käyttöliittymäoperaatioiksi valittiin kaikkien karttaobjektien valinta hiirellä, näppäimistösyötteet ja kartan tarkennus. Karttaobjektien valinta nähtiin tärkeäksi, koska se on havaittu raskaaksi operaatioksi, jonka tiedetään kuormittavan käyttäjäsovellusta monella tavalla. Karttaobjektien valinta on käyttötuntumaltaan hitain toteutetuista operaatioista. Näppäimistösyötteet eivät ole yhtä raskaita kuin karttaobjektien valinta, mutta ne aiheuttavat paljon tapahtumia käyttäjäsovelluksessa. Kartan tarkennus on yleinen operaatio, joka aiheuttaa muutoksia käyttöliittymän piirrossa. Käyttötuntumaltaan se mielletään hitaammaksi kuin näppäimistösyötteet ja nopeammaksi kuin karttaobjektien valinta.

6.3 Testityökalut

Suorituskykymittauksissa hyödynnettiin kahta kolmannen osapuolen työkalua. Operaatioiden automatisointiin käytettiin Robot Framework -testiautomaatioalustaa ja mittausten tekemiseen ja koostamiseen TeamQuestia.

6.3.1 Robot Framework

Käyttäjäsovellusskenaariot ja tilannetietogeneraattorien hallinta toteutettiin avoimen lähdekoodin Robot Framework -testausautomaatioalustalla, joka on kehitetty alun perin No-

kia Siemens Networksilla [28]. Robot Framework oli valittu käytettäväksi uusien automaattitestien toteutuksessa, joista mainittiin aliluvussa 3.4.2. Robot Framework on käytössä myös yrityksen muissa ohjelmistoprojekteissa ja projektien välillä yritetään käyttää samoja työkaluja, jotta osaamista voidaan hyödyntää työtehtäväkierrolla. Suorituskykyskenaariot vastasivat toiminnaltaan muita automaattitestejä, joten saman työkalun käyttäminen oli luonnollista. Samalla voidaan toteuttaa yhteisiä osuuksia, joita hyödynnetään molemmissa testeissä.

Robot Framework on toteutettu Python-ohjelmointikielellä. Se perustuu yksinkertaisiin avainsanoihin, joita käyttäjä voi yhdistellä suorittaakseen testioperaatioita. Robot Frameworkille annetaan käytännössä lista avainsanoja, jotka se suorittaa järjestyksessä. Se siis toimii kuten tulkattavat ohjelmointikielet. Se on vain toteutettu korkeammalla abstraktiotasolla. Robot Framework mittaa yhden avainsanan suoritukseen kuluneen ajan automaattisesti ja luo suorituksista raporttiverkkosivun. Robot Frameworkin avulla voidaan tehdä käyttöjärjestelmäoperaatioita ja käynnistää sovelluksia. Se osaa kiinnittyä sovelluksiin ja antaa niille komentoja. [28]

6.3.2 TeamQuest

TeamQuest on kaupallinen tuote, joka kerää suuren määrän tietoa resurssien käytöstä. TeamQuest jakaantuu useampaan osakokonaisuuteen, joista tämän työn kannalta olennaisia ovat Teamquest Manager ja TeamQuest View [29].

TeamQuest Manager on TeamQuestin osakokonaisuus, joka hoitaa tiedon keräämisen ja taltioinnin tietokantaan. Manager itsessään koostuu mittausagenteista, tietokannasta ja näiden hallintajärjestelmästä. Mittausagentit ovat Managerin tapa kerätä tietoa. Mittausagentit lukevat tietoja tasaisin väliajoin ja tallentavat ne tietokantaan. Manageriin voi lisäksi luoda omia mittausagentteja, jotka keräävät tietoa komentoriviohjelman tulosteista. [29] Tämä on todella hyödyllistä, koska projektissa on toteutettu sisäisiä työkaluja, jotka tuottavat tietoa, mutta tietoa ei ole kerätty talteen järkevästi. Kuten aliluvussa 4.3 kerrottiin, pitkistä suorituskykyskenaarioista on aiemmin tallennettu tuloksia, joiden hyödyllisyys on sittemmin kyseenalaistettu.

TeamQuestin toinen osakokonaisuus on TeamQuest View -sovellus, jolla tietokannan tiedot esitetään graafisesti käyttäjälle. Kerättyjä tietoja voi tarkastella reaaliaikaisesti tai myöhemmin tarpeen mukaan. TeamQuest View mahdollistaa kerätyn tiedon siirron tietokantoihin, tiedostoon tai taulukkolaskentasovellukseen. Siirron avulla on helppo koostaa tiedoista raportteja. Aliluvun 7.1 kuvaajat on tuotettu TeamQuest View'sta Microsoft Excel -taulukkolaskentaohjelmaan viedyillä tiedoilla. [29]

6.4 Järjestelmän suorituskyvyn mittaaminen

Jotta skenaarioiden suorituksesta olisi hyötyä, on samalla kerättävä mittaustuloksia järjestelmän suorituskyvystä. Alaluvuissa on selitetty, miten tuloksia kerätään.

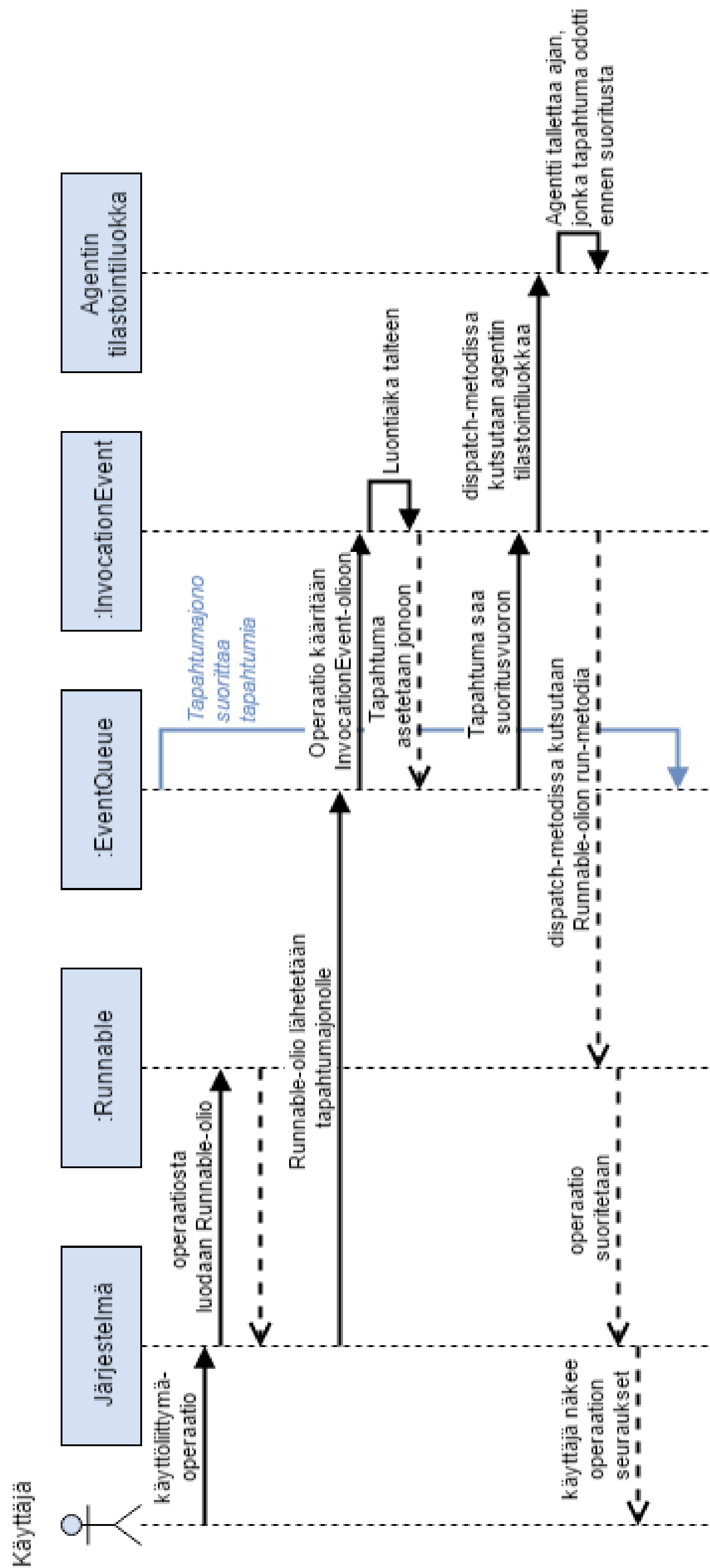
6.4.1 Käyttöliittymäviiveen mittaus

Yksi sisäisesti kehitetyistä työkaluista on Java-agentti, joka voidaan liittää käyttäjäsovellukseen keräämään tilastoja tapahtumajonon (event queue) läpivientiajasta. Tapahtumajonoa käytetään käyttöliittymätapahtumien sarjallistamiseen. Javassa monet säikeet voivat käsitellä käyttöliittymätietoja, mutta käyttöliittymää päivitetään aina tapahtumajonon kautta. Yksi säie purkaa tapahtumajonoa ja suorittaa sinne lisättyjä tapahtumia järjestyksessä. Tätä säiettä kutsutaan Event Dispatch Threadiksi (EDT). Jäljittämällä, kuinka kauan tapahtumalla kestää edetä suoritukseen, voidaan päätellä, kuinka pitkä aika kuluu käyttäjän tekemän operaation ja sen seurauksena tapahtuvan käyttöliittymäpäivityksen välissä.

Java-agentin toiminta perustuu siihen, että se jäljittää, kuinka kauan tapahtuma odottaa ennen sen suoritusta. Teknisemmin kuvattuna: Javan tapahtumajonoon asetettavien olioiden tulee toteuttaa Runnable-rajapinta. Nämä oliot kääritään aina luokan InvocationEvent-instanssiin, kun tapahtuma lisätään tapahtumajonoon (EventQueue-luokka). InvocationEvent-instanssin luonnissa otetaan vakiona talteen olion luontihetken aikaleima. InvocationEvent-instanssi pääsee suoritukseen, ja on näin päässyt tapahtumajonon ensimmäiseksi, kun InvocationEvent-instanssin metodia dispatch kutsutaan. Toteutettu Java-agentti lisää Java-sovellusta käynnistettäessä InvocationEvent-luokan koodiin rivin, joka kutsuu agentin omaa tilastointiluokkaa lähettäen tälle luontihetken ja dispatch-metodin kutsuhetken erotuksen. Tämä vastaa suoraan tapahtumajonon läpivientiaikaa. Kuvassa 4 on esitetty yksittäisen käyttäjäoperaation kulku ja agentin kiinnittymispiste. Kuva havainnollistaa, miten agentti toimii irrallaan mitattavasta järjestelmästä ja kiinnittyy vain Javan sisäisiin luokkiin.

Agentti kerää talteen tapahtumien läpivientiaikoja minuutin ajan. Sen jälkeen se laskee läpivientiajoista keskiarvon, mediaanin, 90-prosenttipisteen, 99-prosenttipisteen sekä mitattujen läpivientiaikojen määrän. Agentti tallentaa nämä tiedot tiedostoon. Tallennuksen jälkeen agentti tyhjää muistinsa ja aloittaa saman syklin alusta.

Agentti kehitettiin suorituskykytestejä varten. Sen manuaalinen testaus on vaikuttanut lupaavalta, mutta se on tarkoitus testata tarkemmin automaattisilla suorituskykyskenaarioilla. Agentin toiminta on tärkeää, koska sen on tarkoitus toteuttaa vaatimus käyttöliittymäviiveen mittarista.



Kuva 4: Käyttöliittymäoperaation kulku Java-sovelluksessa ja viiveen mittausagentin toimintaperiaate.

6.4.2 Läpimenoajan mittaus

Projektin manuaalisten suorituskykytestien tärkein mittari on ollut läpimenoaika. Läpimenoajan mittaus haluttiin ehdottomasti sisällyttää myös automatisoituun suorituskykytestaukseen. Läpimenoajan mittari on toteutettu järjestelmään liitettävänä komponenttina ja se voidaan sovelluspalvelimen käynnistyksen yhteydessä sisällyttää sovelluspalvelimeen tai jättää pois käytöstä.

Läpimenoajalla tarkoitetaan aikaa, joka järjestelmällä kuluu ulkoiselta järjestelmältä saadun tiedon esittämiseen käyttäjälle. Automaattisissa suorituskykytesteissä käytetään täysin samoja asetuksia mittarille kuin manuaalisissa suorituskykytesteissä. Mittari kerää tiedon läpimenoaikoja (näytteitä) jatkuvasti ollessaan käynnissä. Kerran minuutissa se laskee näytteistä lukumäärän, mediaanin, keskiarvon, maksimin ja minimin, ja tallentaa nämä tiedostoon. Tämän jälkeen mittari tyhjää muistinsa ja aloittaa syklin alusta.

Matti Välimäki on kehittänyt alkuperäisen läpimenoajan mittaustyökalun diplomityönään [30]. Työstä on mahdollista lukea lisää läpimenoajan mittarin toiminnasta.

6.4.3 Resurssien käytön mittaus ja mittausten koostaminen

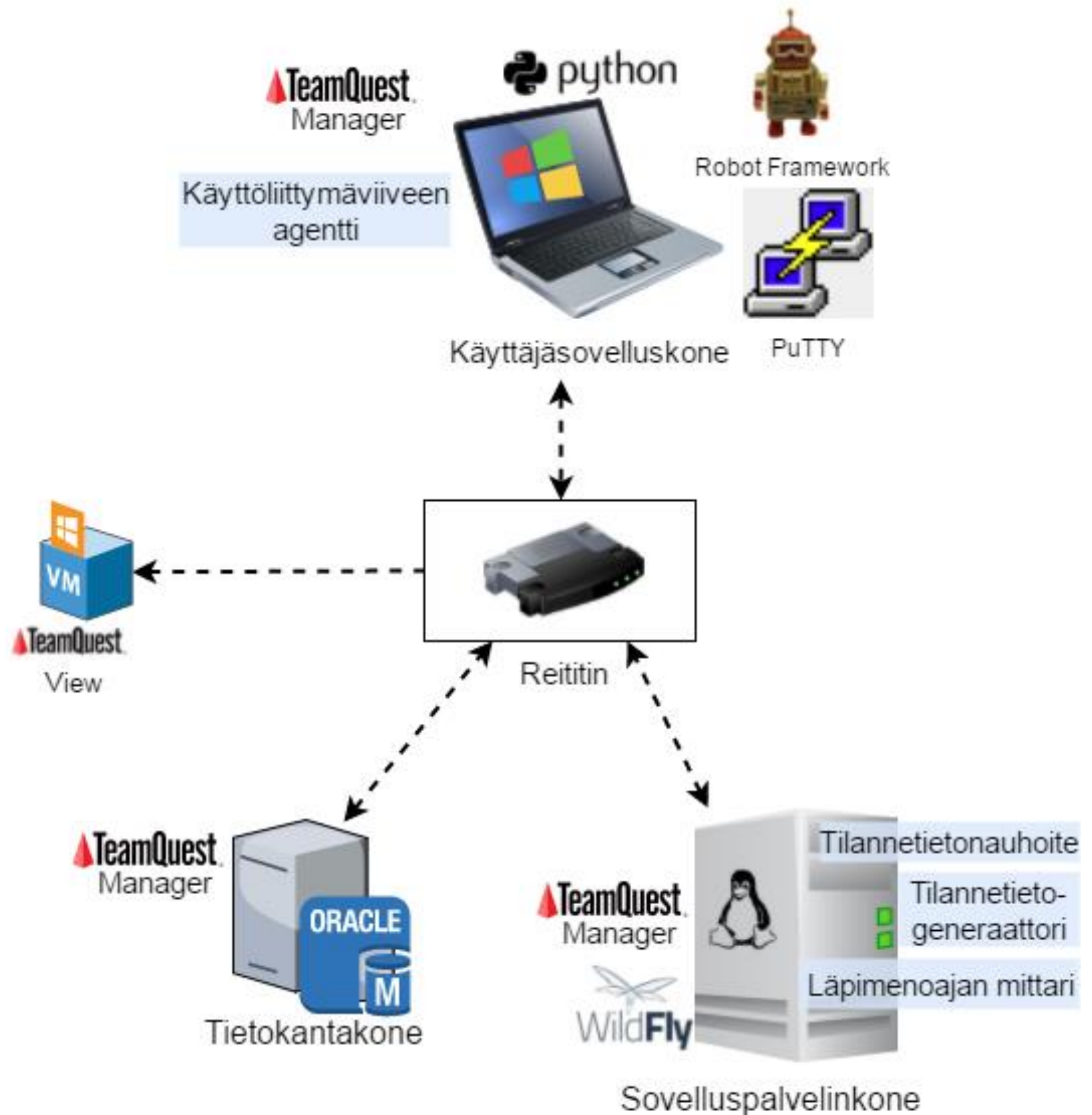
TeamQuest Manager kerää massiivisen määrän tietoja järjestelmän resurssien käytöstä, kuten muistinkäyttö, prosessorikuorma, levyoperaatiot, verkkokuorma, ja johtaa näistä vielä lisää tietoja [29]. TeamQuest Managerilla on mahdollista määritellä tarkasti tiettyjen resurssien, kuten prosessorikuorman, käytön mittaustaso (workload). Mittaustasoksi voidaan määritellä esimerkiksi yksittäinen prosessi. Määrittelyn avulla käyttäjäsovelluksen, testityökalujen ja käyttöjärjestelmän käyttämät resurssit eriytettiin omiksi mittauksikseen. Sama tehtiin sovelluspalvelimelle, nauhoitteelle ja sovelluspalvelimen käyttöjärjestelmälle. Tämä mahdollistaa tarkan seurannan, kuinka paljon itse testattava järjestelmä vie resursseja. Lisäksi voidaan seurata, aiheuttavatko testityökalut merkittävästi kuormaa.

Resurssien käyttömittausten lisäksi TeamQuest Manageriin toteutettiin kaksi omaa mittausagenttia, jotka keräävät läpimenoajan mittarin ja käyttöliittymäviiveen mittarin tallentamat tiedot. TeamQuest Manager on asetettu keräämään tietoja minuutin välein. Näin läpimenoajan mittauskomponentti ja käyttöliittymäviiveen mittari toimivat samassa syklissä TeamQuest Managerin kanssa ja kaikki niiden tuottamat tiedot tallentuvat automaattisesti tietokantaan.

6.5 Automaattisten suorituskykytestien ajoympäristö

Kuten aliluvussa 3.4.3 todettiin, testejä oli ennen ajettu yhdellä fyysisellä koneella. Tämän johdosta suorituskykyongelmat yhdessä järjestelmän osassa heijastuivat toiseen (ks. 4.5). Uudessa suorituskykytestausympäristössä tältä haluttiin välttyä. Käyttäjäsovellukselle, sovelluspalvelimelle ja tietokannalle varattiin kullekin oma fyysinen tietokoneensa.

Samalla mahdollistui verkkoliikenteen tarkempi eroteltavuus, kun jokainen tietokone käyttää omaa verkkokorttiaan ja kommunikoi reitittimen läpi. Kuva 5 esittää suorituskykytestausympäristön tietokoneet ja tärkeimmät niihin asennetut sovellukset.



Kuva 5: Automaattisten suorituskykytestien ajoympäristö

Kaikkiin kolmeen koneeseen asennettiin TeamQuest Manager, jotta jokaiselta koneelta saadaan kerättyä mittaustietoja. Testausympäristön pääpalvelimelle luotiin erillinen virtuaalikone, johon asennettiin TeamQuest View. TeamQuest Viewin kautta saadaan verkon yli yhteys eri suorituskykytestausympäristön koneiden TeamQuest Manager -tietokantoihin ja voidaan tarkastella mittaustuloksia testiympäristön ulkopuolelta. Näin testejä pystytään seuraamaan reaaliajassa häiritsemättä niiden suoritusta.

Käyttäjäsovelluskoneena toimii kannettava tietokone, jossa on Windows-käyttöjärjestelmä. Robot Frameworkin suoritus tapahtuu käyttäjäsovelluskoneella, joten koneeseen

asennettiin Python ja Robot Framework. Lisäksi Windows-koneelle asennettiin Putty-sovellus, jolla voidaan ottaa SSH-yhteys sovelluspalvelinkoneeseen [31].

Sovelluspalvelinkoneena toimii Linux-palvelinkone, jossa ei ollut graafista käyttöliittymää. Konetta hallitaan SSH-yhteyden ylitse. Testattavan järjestelmän asennuksessa palvelinkone toimii samalla tietovarastona, josta käyttäjäsovelluskone voi ladata käyttäjäsovelluksen.

Projektissa tietokantana käytetään Oraclea, joka on yleinen [32] relaatiotietokanta. Tietokantakoneelle TeamQuest Managerissa otettiin käyttöön TeamQuestin Oracle-laajennos, joka kerää tietoja tietokannan käytöstä. Laajennoksella voidaan jäljittää esimerkiksi, mitkä tietokantakutsut ovat hitaita samoin kuin mitä kutsuja suoritetaan eniten. Lisäksi TeamQuest kerää tietokantakoneella kaikki samat resurssien käyttötiedot kuin muillakin koneilla. Tietokantakoneen suorituskyvyn tarkkailu on jätetty tässä diplomityössä taka-alalle, koska sitä ei pidetty suorituskykytestauksen automatisoinnin kannalta kiinnostavana. Kaikki tietokantakoneella suoritettavat sovellukset ovat kolmansilta osapuolilta ja työn aikana vain todettiin niiden toimivan.

6.6 Robot Frameworkin laajentaminen skenaarioiden automatisointiin soveltuvaksi

Kuten aliluvussa 6.3.1 mainittiin käyttäjäsovelluksen, tilannetietogeneraattorin ja nauhoitteen hallintaan käytetään Robot Frameworkia. Robot Frameworkiin tarvittiin muutamia laajennoksia, jotta tilannetietogeneraattorin ja käyttäjäsovelluksen hallinta oli mahdollista.

Aliluvussa 6.1 kuvattu tilannetietogeneraattori A toimii sovelluspalvelimella, mutta sitä oli tarve hallita Robot Frameworkin kautta, joka suoritetaan käyttäjäsovelluskoneella. Automaattitestien toteutuksessa oli otettu käyttöön Jrobotremoteserver, joka voitiin liittää osaksi sovelluspalvelinta. Jrobotremoteserverille voidaan toteuttaa suoraan Java-kirjastoina avainsanoja (ks. 6.3.1), joita on mahdollista kutsua verkon yli Robot Frameworkilla. [33] Automaattitestien osana generaattorin A hallintaan oli toteutettu jo avainsanat, joten generaattori oli suoraan käyttövalmis suorituskykytestaukseen. Generaattori voitiin käynnistää, määrittäen samalla rasitustaso ja sammuttaa suoraan Robot Frameworkista.

Luvussa 6.1 kuvailtiin myös nauhoite, joka korvasi generaattorin C. Nauhoite toistetaan tcpreplay-sovelluksella, jota on ajettava sovelluspalvelinkoneella. Nauhoitteen hallinta tapahtuu SSH-yhteyden ylitse käyttäjäsovelluskoneelta. Sovelluspalvelinkoneella on komentoriviltä hallittava bash-komentosarjakielellä [34] toteutettu sovellus, jolla nauhoite voidaan käynnistää ja sammuttaa. Nauhoite on noin puolen tunnin mittainen ja komentorivisovellus aloittaa nauhoitteen aina alusta, jos sitä toistetaan pidempään kuin puoli tuntia.

Käyttäjäsovellukseen kiinnittymiseen käytettiin Robot Frameworkin laajennuskirjastoa RemoteSwingLibrary. RemoteSwingLibrary perustuu toiseen kirjastoon nimeltä SwingLibrary. SwingLibrary toimii vain Javan virtuaalikoneelle toteutetulla Python-tulkilla (Jython). Tämä aiheutti ongelmia muissa automaattitesteissä, joten niiden kehityksen alkuvaiheessa oli käytettäväksi valittu RemoteSwingLibrary, jota voitiin ajaa alkuperäisellä Python-tulkilla.

RemoteSwingLibrary käytännössä käärii SwingLibraryn sisälleen, joten alla kuvattu SwingLibraryn toiminnallisuus kuvaa samalla RemoteSwingLibraryn toiminnallisuutta. SwingLibrary on suunniteltu Javan käyttöliittymäelementtien käsittelyyn. Sillä voi esimerkiksi painaa painiketta, syöttää tekstiä tekstikenttään ja vastaavia toimintoja. Sen heikkoudeksi osoittautui hankala tapa, jolla eri käyttöliittymäelementteihin päästiin käsiksi. Javan Swing-kirjastolla toteutetut käyttöliittymäsovellukset rakentuvat sisäkkäisistä säiliöistä, jotka sisältävät käyttöliittymäelementtejä ja toisia säiliöitä [35]. Jokaiselle säiliölle ja käyttöliittymäelementille on mahdollista, mutta ei pakollista, antaa nimi. SwingLibrary löytää käyttöliittymäelementit näiden nimien perusteella. Valitettavasti käyttöliittymäelementtien nimeäminen ei ole ollut tarpeen järjestelmän käyttäjäsovelluksen kehityksessä ja nimeämistä on harrastettu lähinnä kehittäjän tottumusten mukaan. Osa käyttöliittymäelementeistä on nimetty ja osa ei. Nyt testitapauksia luodessa jouduttiin muuttamaan sovelluksen omaa koodia ja nimeämään käyttöliittymäelementtejä, jotta niihin päästiin käsiksi. Tämä rikkoi testien taaksepäin yhteensopivuuden. Asia ei ole ongelma muissa automaattitesteissä, mutta suorituskykytestauksessa oli tavoitteena pystyä testaamaan myös vanhempia versioita järjestelmästä.

Suorituskykytestausautomaation kehityksessä havaittiin tarve laajentaa RemoteSwingLibrarya. SwingLibraryn laajentaminen on helppoa [36], mutta sama ei päde RemoteSwingLibraryyn. Asiaa tutkiessa havaittiin, että RemoteSwingLibraryn toteutuksessa SwingLibraryn laajennettavuus on peittynyt, koska RemoteSwingLibraryyn ei ole toteutettu ominaisuutta, jolla laajennettavuutta voitaisiin käyttää. Myös muissa projektin automaattitesteissä oli ilmennyt tarve laajentaa RemoteSwingLibraryn toiminnallisuutta, joten päätettiin selvittää, onko RemoteSwingLibraryn laajentaminen mahdollista toteuttaa sisäisesti. Koska RemoteSwingLibrary on avointa lähdekoodia, pystyttiin ominaisuus toteuttamaan suoraan RemoteSwingLibraryn lähdekoodiin, jonka jälkeen lähdekoodista käännettiin projektin käyttöön uusi versio.

Teknisesti toteutus oli melko suoraviivainen. Robot Frameworkin skenaariomäärittelytiedostoissa on määriteltävä käytettävät kirjastot. RemoteSwingLibraryn määritykseen lisättiin valinnainen parametri, johon voi antaa Java-luokkapolkuja, jotka sisältävät SwingLibraryn hyväksymiä Java-avainsanaluokkia [36]. RemoteSwingLibraryssa luokkapolut tarjottiin edelleen sen sisäisesti käyttämälle SwingLibrarylle, joka osasi käsitellä niitä. Lisäksi Robot Framework vaatii laajennuskirjastoilta yhtenäisen listan avainsanoista, joita ne tarjoavat käyttäjälle. RemoteSwingLibraryn toteutus sisälsi avainsanoja lukevan Java-luokan, jota käytettiin vain käännösvaiheessa lukemaan SwingLibraryn avainsanat.

Tämä luokka muutettiin toimimaan dynaamisesti. Ladattaessa kirjastomäärittely Robot Frameworkissa Java-luokka lukee parametreina annetuista luokkapoluista löytyvät avainsanat ja lisää ne RemoteSwingLibraryn avainsanalistaan. Näin käyttäjän on mahdollista laajentaa RemoteSwingLibrarya helposti parametrilla. Käyttäjän tulee vain asettaa toteuttamansa käännetyt avainsanaluokkatiedostot Java-sovelluksen luokkapolulle ja antaa Robot Frameworkille kirjastomäärittelyn parametrina luokkapolku, josta tiedosto löytyy.

Projektin sisäinen yhteistyö kantoi hedelmää. Suorituskykytesteissä päästiin hyödyntämään automaattitestien osana kehitettyjä generaattoriavainsanoja ja automaattitesteissä voitiin nyt hyödyntää RemoteSwingLibrary-laajennuksia. Yrityksen sisäisesti RemoteSwingLibraryn laajennus on otettu käyttöön jo toisessakin projektissa, ja siitä uskotaan olevan hyötyä myös muille. Tästä syystä toteutuksen lähdekoodi on tarkoitus jakaa RemoteSwingLibraryn kehittäjille ja käyttäjille. Valitettavasti koodin siivoaminen, yrityksen sisäinen hyväksyntä ja hyväksyttäminen RemoteSwingLibraryn kehittäjillä eivät mahtuneet tämän työn aikatauluun, joten lähdekoodi jaetaan vasta työn valmistuttua.

6.7 Käyttäjäsovellusskenaarioiden automatisointi

RemoteSwingLibraryn laajentaminen avasi ovet koko käyttäjäsovelluksen hallitsemiseen Robot Framework -avainsanoilla. Nyt pystyttiin toteuttamaan Java-kirjastoja, jotka hallitsevat suoraan karttaa ja karttaobjekteja. Koska karttaan päästiin käsiksi vanhojen rajapintojen avulla ilman komponenttien uudelleennimeämistä, kartan hallitseminen pystyttiin toteuttamaan taaksepäin yhteensopivasti. Tämä mahdollisti vanhojen versioiden testauksen. Karttaobjektien hallinnan mahdollistamiseksi itse sovelluksen koodiin jouduttiin toteuttamaan yksi kahden koodirivin muutos. Tästä syystä karttaobjektien hallintaan tarkoitettuja avainsanoja ei voi käyttää vanhoissa sovellusversioissa.

Ensimmäisenä toteutettiin leposkenaario, johon muut skenaariot pohjaavat. Skenaariossa käynnistetään ensin käyttäjäsovellus ja tämän jälkeen tilannetietogeneraattorit. Käyttöliittymään asetetaan tietty kartan tarkennustaso, jossa tiedetään näkyvän kaikki karttaobjektit. Tarkennustason määritystä varten toteutettiin RemoteSwingLibrary-laajennus, joka mahdollistaa käyttöliittymänäkymän tarkan muokkaamisen ohjelmallisesti. Tämä laajennus toimii myös vanhojen sovellusversioiden kanssa. Lisäksi leposkenaarion aikana toteutettiin silmukka, johon voidaan määritellä avainsana ja kesto. Silmukka toistaa avainsanaa keston ajan. Kesto voi olla toistokertojen määrä tai aika, jonka ajan avainsanaa toistetaan. Leposkenaariossa silmukkaa ei ensin käytetty ollenkaan, vaan käytettiin pelkkää lepoavainsanaa (sleep). Myöhemmin huomattiin, että leposkenaario ei tässä tapauksessa huomaa, jos käyttäjäsovellus kaatuu tai sammutetaan, vaan odottaa määritellyn ajan joka tapauksessa. Niinpä myös leposkenaariossa otettiin käyttöön sama silmukka. Silmukassa tarkistetaan puolen minuutin välein, että sovellus on elossa. Tämä onnistui RemoteSwingLibraryn avainsanalla, jolla valitaan tietyn niminen sovellus. Jos sovellusta ei

löydy, testi keskeytyy epäonnistuneena. Leposkenaarion lopuksi sammutetaan generaattorit ja käyttäjäsovellus. Muut skenaariot toteutettiin samaan runkoon. Käyttäjäsovelluksen ja generaattorien käynnistys ja sammutus tehdään samalla tavalla joka skenaariossa.

Robot Frameworkissa on suoraan avainsana, jolla voi tehdä näppäimistösyötteitä. Näppäimistösyöteskenaariossa käytettiin tuota avainsanaa syöttämään käyttäjäsovellukselle tietyt näppäimet. Nyt aiemmin selitettyyn silmukkaan asetettiin avainsana, joka toisti näppäinten painalluksia. Avainsanan alkuun asetettiin käyttäjäsovelluksen valinta aktiiviseksi sovellukseksi, jotta voidaan olla varmoja, että näppäinsyötteet menevät sovellukselle. Valinta oli mahdollista Robot Frameworkin omalla avainsanalla.

Kaikkien karttaobjektien valintaskenaario toteutettiin hiiren liikuttamiseen perustuvalla avainsanalla. Skenaarion kohdalla harkittiin toteuttaa valinta ohjelmallisesti, mutta hiiren liikutus haluttiin sisällyttää skenaarioon, koska ohjelmallisessa toteutuksessa jäisi pois hiirenliikkeistä ja painalluksista aiheutuvia käyttöliittymätapahtumia. Hiiren toiminnan manipulointi oli siis lähempänä ihmiskäyttäjää kuin ohjelmallinen. Lisäksi ohjelmallinen toteutus ei olisi toiminut vanhojen versioiden kanssa, koska valintatyökaluun ei päästy käsiksi muuttamatta sovelluksen koodia. Skenaariossa käytettiin hyväksi Javan AWT Robot -toteutusta, jolla pystytään liikuttamaan hiirtä ja tekemään hiiren näppäinpainalluksia. RemoteSwingLibraryn laajennuskirjastoon toteutettiin hiiren liikutukselle avainsanat. Lisäksi tehtiin avainsana, joka ensin etsii pisteen näytöllä, jossa on karttaikkunan vasen yläkulma. Sen jälkeen avainsana suorittaa siitä pisteestä hiirellä raahausoperaation kartan oikeaan alakulmaan. Näin saadaan valittua kaikki näkyvät karttaobjektit hiirellä riippumatta karttaikkunan koosta.

Kartan tarkennus toteutettiin ohjelmallisesti jo osana leposkenaariota. Kartan tarkennuksessa oli ihmiskäyttäjän jäljittelyä tärkeämpää saada tarkennettua kartta tarkasti. Hiirellä on mahdollista tarkentaa karttaa, mutta pikselilleen saman näkymän saavuttaminen hiiren liikkeillä ei ole mahdollista. Tarkennusskenaariossa valittiin käyttää ohjelmallista tarkennusta, jotta skenaario ei vaihtele suoritusten välillä. Skenaariossa tarkennetaan ja loitonnetaan karttaa silmukassa sekunnin välein.

6.8 Skenaarioiden suorituksen automatisointi

Kun skenaariot oli automatisoitu Robot Frameworkilla, tarvittiin tapa automatisoida alkuvalmistelut ja Robot Framework skenaarion käynnistys. Alkuvalmisteluilla tarkoitetaan järjestelmän asennusta ja sovelluspalvelimen käynnistystä, jotka rajattiin pois Robot Frameworkin vastuualueesta. Tämä päätettiin jo automaattitestien suunnittelussa ja päätökseen johtaneita syitä on käsitelty seuraavissa kappaleissa.

Ensinnäkin sovelluspalvelimen hallinta on tällä hetkellä muutoksessa. Muutamaa itsenäistä palvelua ollaan liittämässä osaksi sovelluspalvelinta ja niiden tuomia muutoksia ei haluttu osaksi skenaarioiden hallintaa. Lisäksi versioiden asennusta ja hallintaa ollaan

siirtämässä Ansible-työkalun vastuulle, joka muuttaa asennuskäytäntöjä lähitulevaisuudessa merkittävästi. Ansible on järjestelmien asennuksen ja konfiguraation automatisointiin tarkoitettu työkalu [37].

Toisekseen sovelluspalvelinta ei haluta käynnistää jatkuvasti uudelleen. Sovelluspalvelimen tulisi toimia erilaisten skenaarioiden aikana täysin vastaavasti, vaikka sovelluspalvelin olisi ollut päällä pidempäänkin.

Myös kehityksen kannalta on helpompaa, jos sovelluspalvelimen hallinta ei ole osa Robot Frameworkin skenaarioita, vaan erikseen suoritettavina operaatioina. Asennus ja sovelluspalvelimen käynnistys ja sammutus toimivat eri tavalla kehitys- ja testausympäristöissä, joten testiskenaarioissa jouduttaisiin jatkuvasti huomioimaan molemmat ympäristöt. Käyttäjäsovellus toimii samoin molemmissa ympäristöissä ja sille tehtyjen skenaarioiden testaaminen kehitysympäristössä on mahdollista. Myös käynnissä olevaa sovelluspalvelinta voidaan hallita Robot Frameworkilla samoin molemmissa ympäristöissä, joten se ei tuota lisätöitä. Lisäksi sovelluspalvelimen käynnistys on hidas operaatio ja sen suorittaminen aina skenaariota ajettaessa olisi aikaa vievää ja turhauttavaa kehittäjille.

Suorituskykytesteistä muodostui muiden automaattitestien alaprojekti, kun suorituskykytestiskenaariot päätettiin toteuttaa Robot Frameworkilla. Suurin osa suorituksen automatisoinnista oli töille yhteistä. Kaikista Robot Framework -testiskenaariosta ja niiden ajamiseen tarvittavista tiedostoista päätettiin luoda käännöksessä yhteinen paketti. Paketti sisällytettiin sisäisten versioiden käännöksessä osaksi luotua järjestelmän asennuspakettia.

Myös jatkuvan integraation palvelimella (Jenkins) ajettavat tehtävät suunniteltiin niin, että niitä voidaan käyttää kaikkien Robot Framework -testiskenaarioiden suoritukseen. Yksi Jenkins-tehtävä suorittaa sovelluspalvelimen asennuksen ja konfiguroinnin. Toinen Jenkins-tehtävä suorittaa käyttäjäsovelluksen asennuksen, sovelluspalvelimen käynnistyksen ja itse testiskenaarion ajamisen. Testiskenaarion vastuulle jää käyttäjäsovelluksen käynnistys ja tilannetietojen tuottaminen skenaarion tarpeiden mukaan.

Ensin tarvittiin keino hallita testausympäristön koneita jatkuvan integraation palvelimelta. Suorituskykytestausympäristössä oli suoritettava komentoja Windows- ja Linux-käyttöjärjestelmissä. Projektissa molempiin oli vakiintuneet käytännöt, koska vastaavia tarpeita oli ollut ennenkin. Projektin Jenkins-palvelin pyörii Linux-käyttöjärjestelmällä, joten kaikissa palvelinkoneella suoritettavissa tehtävissä on mahdollista suorittaa Linux-komentorivikäskyjä [26]. Linuxin ssh-komentorivityökalulla voitiin muodostaa yhteys sovelluspalvelinkoneeseen ja antaa sille komentorivikäskyjä [38]. Käyttäjäsovelluskoneen hallinnan mahdollistamiseksi siihen asennettiin Jenkins-palveluagentti (slave agent). Palveluagentti käynnistetään käyttäjäsovelluskoneella, jonka jälkeen palveluagentti luo yhteyden Jenkins-palvelimeen. Jenkins-tehtävässä voidaan nyt määrittää

tehtävä suoritettavaksi suoraan palveluagenttikoneella [26]. Näin Jenkins-tehtävässä voidaan suorittaa Windows-komentorivikäskyjä käyttäjäsovelluskoneelle. Jenkins-palveluagentti asetettiin käynnistymään automaattisesti Windows-koneelle, jotta koneeseen saadaan yhteys heti sen käynnistyttyä. Tämä mahdollisti järjestelmän uudelleenkäynnistyksen Jenkinsin kautta.

Järjestelmän asennustehtävässä ajetaan ensin sovelluspalvelin alas, jos se on käynnissä. Tällä vain varmistetaan, että sovelluspalvelin ei ole jäänyt päälle. Yleisesti sovelluspalvelin on sammutettu edellisen ajon aikana. Seuraavaksi ajetaan järjestelmän asennukseen projektin sisäisesti toteutettu Bash-komentosarjasovellus [34]. Se lataa verkkolevyltä asennuspaketin, purkaa sen, konfiguroi asennuksen ympäristöön sopivaksi ja asettaa asennetun version aktiiviseksi. Aktiivinen versio löytyy aina palvelinkoneilta samasta polusta, joka mahdollistaa samojen komentorivikäskyjen käytön eri palvelinkoneilla ja järjestelmäversioilla. Seuraavaksi puretaan järjestelmän asennuspaketin mukana tullut testijärjestelmän asennuspaketti. Viimeisenä ajetaan testausjärjestelmän asennuspakettiin kuuluva Bash-komentosarjasovellus. Sovellus valmistelee sovelluspalvelimen suorituskykytestejä varten. Se asentaa Jrobotremoteserverin, sen riippuvuudet ja itse toteutetut avainsanakirjastot Wildfly-sovelluspalvelimelle. Sovellus muuttaa sovelluspalvelimen asetustiedostoja, jotta tilannetietogeneraattori ja läpimenoajan mittari ovat käytettävissä sovelluspalvelimen käynnistyessä. Lisäksi sovellus korvaa testitiedostoista tiettyjä paikanpitäjämerkkijonoja (placeholder) sovelluspalvelimen IP-osoitteella ja vastaavilla tiedoilla. Nyt sovelluspalvelin on valmis käynnistettäväksi ja sen toiminta on säädetty testiskenaarioille sopivaksi. Asennustehtävä on valmis.

Seuraavaksi toteutettiin testiskenaarioiden suoritusta hallitseva Jenkins-tehtävä. Kuten aiemmin mainittiin, se suoritetaan aina testiympäristön käyttäjäsovelluskoneella, jotta suoritusta voidaan hallita Jenkinsin palveluagentilla. Tehtävä korvaa ensimmäisenä vanhan käyttäjäsovelluksen uusimmalla versiolla, jonka se lataa sovelluspalvelinkoneelta. Tähän käytetään samaa projektissa toteutettua Windows-komentosarjaa (batch) kuin manuaalisessakin asennuksessa. Seuraavaksi poistetaan vanhat testausjärjestelmätiedostot käyttäjäsovelluskoneelta ja tilalle kopioidaan uudet sovelluspalvelimen kansiosta, johon testijärjestelmäpaketti purettiin. Nyt käyttäjäsovellus ja testiskenaariot ovat käynnistettävissä.

Käynnistyksen ensimmäinen vaihe on synkronoida kellot käyttäjäsovellus- ja sovelluspalvelinkoneiden välillä. Kellojen synkronointi on tärkeää läpimenoajan mittauksille [30]. Tämän jälkeen käynnistetään sovelluspalvelin. Sovelluspalvelimen käynnistyttyä ajetaan itse testiskenaario. Tehtävässä odotetaan skenaarion suoritus. Kun skenaario päättyy, tehtävä sammuttaa vielä sovelluspalvelimen. Nyt suoritustehtävä on valmis.

Asennus- ja suoritustehtävät eivät ota kantaa suorituskykytestien tuloksiin. Ne merkataan onnistuneeksi, jos asennukset, käynnistykset ja sammutukset onnistuvat. Sen sijaan to-

teutettiin erillinen Jenkins-tehtävä, joka kerää yhteenvedon testien tuloksista. Testausjärjestelmää kehitetään ja testataan jatkuvasti. Jokaisesta testiajosta ei haluta kerätä tulosyhteenvetoa. Kun tuloksia keräävä Jenkins-tehtävä on erillään muista tehtävistä, tulosyhteenvedot voidaan suorittaa vain tiettyjen automatisoitujen testiajojen yhteydessä. Tulostehtävä voidaan suorittaa tarpeen mukaan myös manuaalisesti.

Yllä kuvatuilla tehtävillä testien ajaminen ja tulosten keräys onnistuvat nyt Jenkins-palvelimelta, mutta Jenkins-tehtäviä on kolme ja ne täytyy käynnistää manuaalisesti Jenkinsin web-sovelluksesta. Tehtävien suorituksen automatisointiin toteutettiin neljäs Jenkins-tehtävä, joka suorittaa edelliset kolme tehtävää järjestyksessä ja tietyin ehdoin. Ensin suoritetaan asennus, ja sen onnistuessa suoritetaan suoritustehtävä. Jos suoritustehtäväkin onnistuu, voidaan suorittaa tulostehtävä. Ennen tulostehtävää tarkastetaan, käynnistikö yöllisen version luonti vai käyttäjä tämän testikokonaisuuden. Tulostehtävä suoritetaan vain, jos testikokonaisuus oli käynnistetty yöllisen version luonnin yhteydessä.

Tehtäviin toteutettiin useita parametreja, joiden avulla tehtävien suoritusta voidaan muokata[26]. Asennus ja suoritustehtävät toimivat kaikkien Robot Framework -automaattitestien kanssa. Tulostehtäviä ei nähty järkeväksi yleistää, koska eri testeillä kerätään erilaisia tuloksia, eikä niitä haluta sotkea keskenään. Jenkinsissä on mahdollista kopioida [26] tulostehtävä ja muokata siitä testeihin sopiva. Jos tulostehtäviä tulee jatkossa useita, saatetaan niille toteuttaa yhteinen päätehtävä, joka osaa päätellä, mitkä tulokset skenaarioista on mahdollista kerätä.

Ajettava testiskenaario parametroitiin, jotta eri automaattitestejä on mahdollista ajaa samalla tehtävällä. Ymmärtääkseen miten toteutettu parametri toimii, täytyy käyttäjän ensin ymmärtää, miten testejä ajetaan Robot Frameworkilla. Robot Framework -testit suoritetaan komentorivikäskyllä "`robot`". Käskylle voidaan antaa argumenttina useita asetuksia [39]. Käskyn loppuun määritellään kansiot tai tiedostot, joista testitapaukset löytyvät.

Vain yksi robot-asetus haluttiin käyttöön kaikissa testeissä. Robot luo suorituksen aikana useita lokitiedostoja ja nämä haluttiin siistiin hakemistorakenteeseen. Asetuksella ”*--outputdir <hakemisto>*” kaikki yhden suoritustehtävän aikana luodut tiedostot voitiin sijoittaa aikaleimalla nimettyyn kansioon [39]. Samaan kansioon on mahdollista siirtää testiskenaarioissa ajoaikaisesti muitakin järjestelmän tuottamia lokitiedostoja, jolloin kaikki yhden suorituksen aikana tuotetut tiedostot saadaan säilöttyä siististi samaan kansioon.

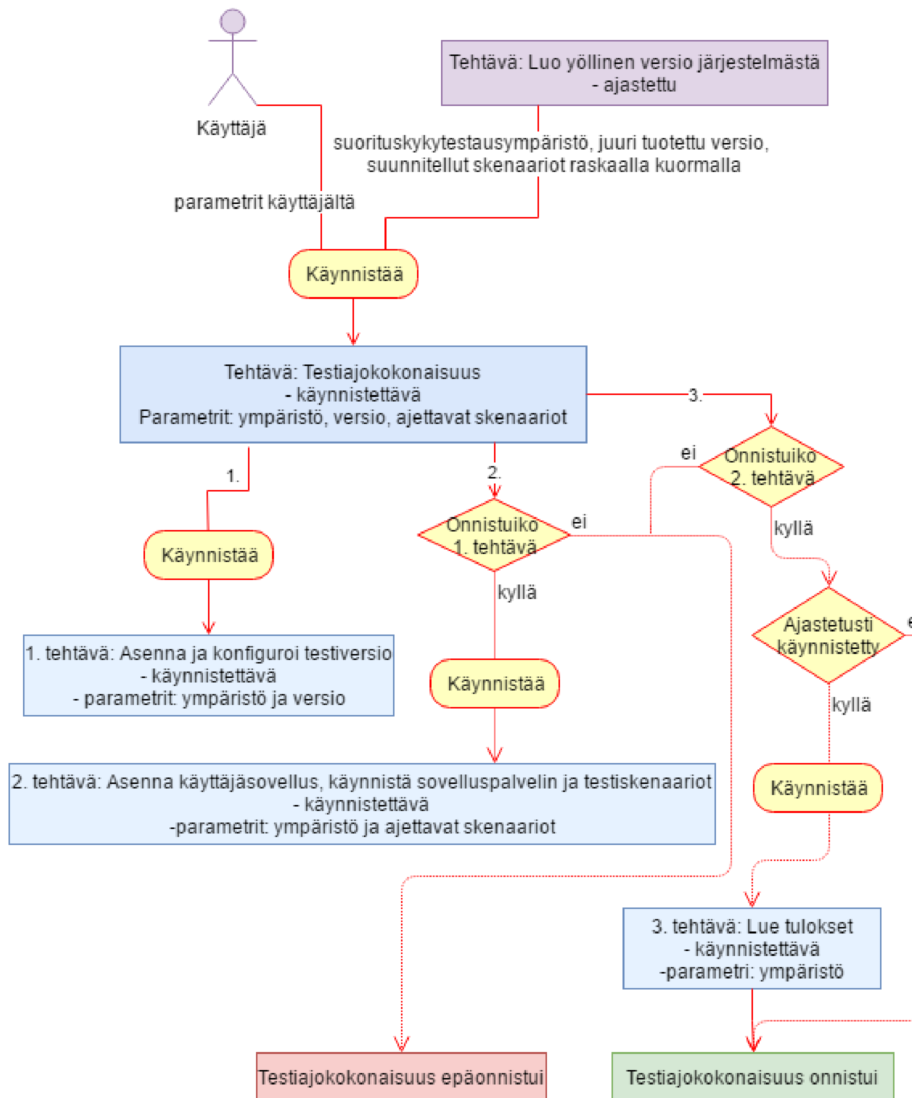
Jenkinsin testien suoritustehtävän parametrissa ei haluttu rajoittaa, miten robot-sovellusta voidaan ajaa, joten parametri annetaan sellaisenaan robot-komentorivikäskylle. Parametriin voi siis testitapausten lisäksi antaa suoritukselle tarpeelliset robot-käskyn asetukset. Suorituskykytesteissä erityisasetuksille ei ole ollut tarvetta ja parametriin on listattu vain ajettavat skenaariot, mutta jatkokehityksessä ja muissa automaattitesteissä asetusparametreista on varmasti hyötyä.

Jotta kaikkia automaattitestejä ei tarvitse ajaa suorituskykytestausympäristössä, myös tehtävän suoritusympäristöstä tehtiin parametroitava. Jenkins ei lähtökohtaisesti anna parametroida tehtävien suoritusympäristöä, vaan tehtävä suoritetaan joko missä tahansa käytössä olevassa ympäristössä tai vaihtoehtoisesti tehtävää luodessa tai muokattaessa voidaan asettaa staattinen etiketti (label), jota käytetään ympäristön tunnistamiseen [26]. Käytännössä jokaiselle suoritusympäristölle voi asettaa eri etiketin, jolloin muokkaamalla tehtävää olisi mahdollista valita ympäristö ennen ajoa. Tämä ei olisi ollut kätevää ja erityisenä vaarana oli, että automaattisesti käynnistyvät testit päätyvät ajoon väärässä ympäristössä, jos joku manuaalisesti muokkaa tehtävän asetuksia ja unohtaa palauttaa ne. Onneksi Jenkinsiin löytyy satoja laajennoksia [26] ja tämänkin ongelman ratkaisuun löytyi omansa. Jenkins-palvelimelle asennettiin Node and Label parameter -laajennos [40], jolla voidaan määrittää tehtävälle ajokohtaisesti ympäristö, jossa se halutaan suorittaa. Oletusarvona tehtävien ympäristönä toimii suorituskykytestausympäristö, mutta tehtäviä on testattu myös virtuaalisessa ympäristössä, johon on asennettu Robot Framework -testityökalu. Erityisesti automaattitestien, jotka eivät testaa suorituskykyä, suoritus saatetaan siirtää virtuaaliseen ympäristöön.

Ensimmäinen versio asennustehtävästä asensi aina uusimman version. Sisäisesti toteutetussa asennussovelluksessa tämä mahdollisuus oli suoraan mukana, joten se oli helpoin toteuttaa. Tarvitsi vain kutsua asennussovellusta tietyllä vivulla ja ilman parametreja. Asennussovellukselle on kuitenkin mahdollista antaa parametrina järjestelmäpaketti, joka halutaan asentaa. Niinpä paketista voitiin tehdä parametri. Parametrin avulla asennustehtävälle voitiin antaa vanhan version pakettinimi. Tämä mahdollisti vanhojen versioiden asennuksen.

Koska testausjärjestelmäpaketti oli osa järjestelmän asennuspakettia, toteutettiin tehtävään myös varmuuskopiointitoiminto, joka tallensi uusimman testausjärjestelmäpaketin eri kansioon sovelluspalvelinkoneella. Nyt asennustehtävä pystyi käyttämään tätä pakettia, jos asennettava versio oli niin vanha, ettei siinä ollut testausjärjestelmäpakettia mukana.

Lisäksi projektissa on käytössä Jenkins-tehtävä, jolla voidaan luoda versio tietystä haarasta versionhallinnassa. Suorituskykytestausjärjestelmää kehittäessä muutoksia oli usein tarve testata suorituskykytestausympäristössä. Versiot olivat usein karkeita, eikä niitä haluttu liittää versionhallinnan päähaaraan. Helpoin tapa oli luoda kehityshaarasta versio ja suorittaa testikokonaisuuden ajava tehtävä käyttäen luotua versiota parametrina. Tätä helpotettiin entisestään luomalla yksinkertainen tehtävä, joka kutsui ensin kehityshaaraversion luontiin käytettyä tehtävää ja putkitti versiopaketin nimen suoraan testikokonaisuustehtävälle.



Kuva 6: Jatkuvan integraation palvelimen tehtävähierarkia

Kuvassa 6 on havainnollistettu tehtävähierarkiaa. Jenkinsin web-käyttöliittymästä käyttäjä voi manuaalisesti käynnistää testiajokokonaisuuden tai minkä tahansa sen alitehtävistä. Näin käyttäjällä on mahdollisuus esimerkiksi asentaa versio kerran ja toistaa sen jälkeen suoritustehtävää useasti. Yöllisen version tuottamista varten oli luotu tehtävä jo aiemmin. Tämän työn osana tehtävään lisättiin ehdollinen loppukäsky: jos tehtävä on onnistunut ja versio on saatu tuotettua, käynnistetään testiajokokonaisuus tietyin parametrein.

Yöllisen version luonnin jälkeisessä automaattisessa suorituksessa testiajokokonaisuuden ympäristöksi asetetaan aliluvussa 6.5 kuvattu suorituskykytestausympäristö. Skenaarioiksi määriteltiin aliluvun 6.2 operaatiot, joita jokaista toistetaan 30 minuuttia. Niihin kaikkiin on asetettu raskas kuorma ja jokaisen välillä käynnistetään käyttäjäsovellus, kuormitusnauhoite ja kuormitusgeneraattori uudestaan. Versio on tietenkin se, jonka yöllisen version luonti tehtävä tuotti.

7. SUORITUSKYKYMITTAUSTEN AUTOMATISOINNIN TESTAUS JA TYÖN ARVIOINTI

Tässä luvussa arvioidaan toteutettua järjestelmää. Ensin arvioidaan testausympäristön luotettavuutta sekä kerrotaan kehityksessä ja toimivuuden todentamisessa havaituista ongelmista. Käyttöliittymäviivemittarin testaukseen on kiinnitetty erityistä huomiota, koska mittari ei ole ollut käytössä aiemmin. Seuraavaksi peilataan työn onnistumista asetettuihin vaatimuksiin ja luvun lopulla käsitellään työn aikana heränneet jatkokehitysideat.

7.1 Testausympäristön luotettavuuden tutkiminen

Jotta työn voisi katsoa olevan onnistunut, on testausympäristön oltava luotettava. Niinpä tässä työssä testausympäristöä ja mittareita testattiin suunnitelmallisesti. Vaikka tiedettiin näin ison kokonaisuuden testaamisessa törmättävän todennäköisesti joihinkin ongelmiin, otettiin lähtöoletukseksi, että järjestelmä toimii moitteetta. Oletusta lähdettiin todentamaan ajamalla kymmeniä testiskenaarioajoja ja odottaen vertailukelpoisia tuloksia ajojen välillä. Kuten ennalta arvattiin, testauksen aikana havaittiin useita ongelmia. Kun ongelmiin törmättiin, etsittiin niihin ratkaisu, jonka jälkeen toistettiin lisää testiskenaarioajoja. Tätä prosessia toistettiin, kunnes testausjärjestelmän koettiin olevan luotettava. Tärkeimmät havainnot testiskenaarioajoista on kerätty seuraaviin alilukuihin niiden esiintymisjärjestyksessä.

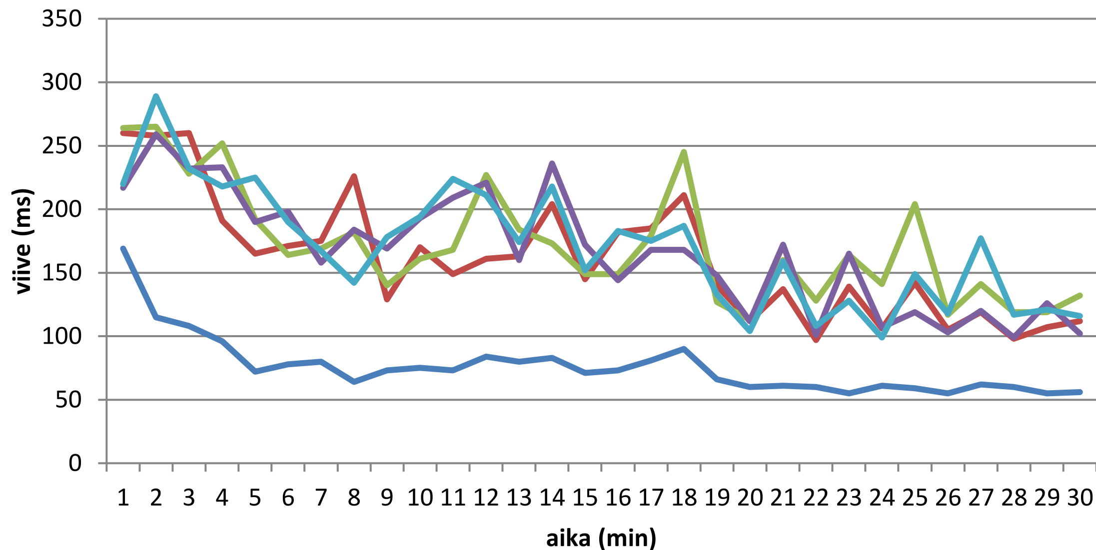
7.1.1 Windowsin etätyöpöytäyhteys vääristää mittaustuloksia

Jo kehitysvaiheessa ilmeni ongelmia Windows-etätyöpöytäyhteyden kanssa, jota käytettiin käyttäjäsovellusympäristön hallintaan. Normaalisti Windows-ympäristö on aina kirjautuneena, jotta testit voidaan ajaa. Etäyhteyden sulkeminen lukitsee työpöytänäkymän, joka puolestaan aiheuttaa Java-prosessiin kuormitustilan, jossa 50 prosenttia prosessoriajasta kuluu järjestelmäkutsuihin. Normaalissa ajossa järjestelmäkutsujen osuus on alle prosentin prosessoriajasta. Jos etäyhteys on suljettu ennen Java-prosessin käynnistystä, kuormitustilaa ei tapahdu, mutta käyttöliittymäobjektien valintaskenaariota ei pystytä suorittamaan ja sen tulokset vastaavat leposkenaariota.

Ongelma kierrettiin sulkemalla etätyöpöytäyhteys käyttäen komentorivikäskyä "$tscon\ \%sessionname\%\ /dest{:}\ console$", joka ei lukitse työpöytää. Etätyöpöytäyhteyden käyttö aiheutti silti epävarmuutta ja testitulosten puhtauden varmistamiseksi luotiin uusi Jenkins-tehtävä, joka käynnistää käyttäjäsovelluskoneen uudelleen ennen testejä. Nyt etäyhteys olisi otettava testin aikana, jotta ongelma toistuisi automaattisissa testiajoissa. Käyttäjäsovelluskoneen uudelleenkäynnistystehtävä sijoitettiin aliluvussa 6.8 kuvatun testiajokokonaisuustehtävän alkuun.

7.1.2 Ensimmäiset testit käyttöliittymäviiveen mittarille

Käyttöliittymäviiveen mittari toteutettiin ja otettiin käyttöön tämän työn aikana. Läpimenoajan mittari oli ollut käytössä vuosia ja siihen katsottiin voitavan luottaa. Samoin TeamQuestin mittauksiin luotettiin, koska työkalu on alalla laajasti käytössä [41]. Käyttöliittymäviiveen mittarin toiminta haluttiin varmentaa ja se olikin testauksen pääpainopiste.



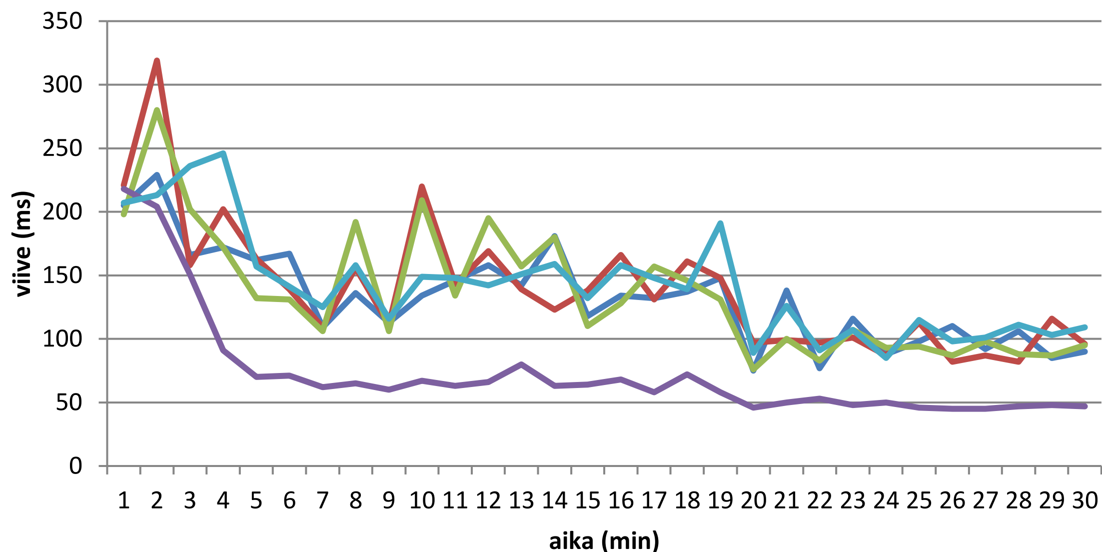
Kuva 7: Viiden 30 minuutin testiajon 90. prosenttipisteen viiveen mittaustulokset valintaoperaatiolle.

Käyttöliittymäviiveen mittarin toiminnan varmistamiseksi suoritettiin ensin samaa testitapausta useita kertoja ja seurattiin tuloksia. Testitapaukseksi valittiin karttaobjektien valinta, koska sen tiedettiin olevan käyttötuntumaltaan hitain suunnitelluista operaatioista. Kuvassa 7 on esitetty 90. prosenttipisteen mittaustulokset, kun samaa skenaariota toistettiin viisi kertaa 30 minuutin ajan. Sovelluspalvelinta ei uudelleenkäynnistetty skenaarioiden välillä. Suorituksen ensimmäinen mittauspiste on jätetty pois kuvasta, koska käyttäjäsovelluksen käynnistyksessä tapahtuva lataus aiheuttaa tuloksen vääristymisen suurempaan kertaluokkaan.

Kuvasta 7 voidaan huomata, että yksi (sininen käyrä) viidestä suorituksesta on ollut viiveeltään selvästi pienempi kuin neljä muuta. Kaikissa suorituksissa viive myös laskee huomattavasti suorituksen aikana. Testejä suoritettiin eri skenaarioilla ja sama ilmiö toistui arviolta kerran viidessä testiajossa. Oli selvää, että testausjärjestelmä ei toiminut toivotun vakaasti.

7.1.3 Käyttäjäsovelluksen dynaamisen piirto-optimoinnin vaikutus tuloksiin

Omituista oli, että raskas kuorma ei ollut välttämättä hitain vaan pieni viive tuntui osuvan yhtä usein kaikkien kuormitustasojen testeihin. Käyttäjäsovelluksessa on ominaisuus, jonka avulla se yrittää sopeutua automaattisesti kuormitustilanteisiin: tiettyjen piirto-operaatioiden suoritusväliä kasvatetaan, mikä vapauttaa prosessoriaikaa kriittisemmille operaatioille. Kun kuormitus laskee, sovellus korjaa piirto-operaatioiden suoritusvälin takaisin normaalitasolle. Tämän ominaisuuden uskottiin olevan syyllinen tulosten heittelyyn. Ominaisuus otettiin pois käytöstä, jotta nähtiin sen vaikutus tuloksiin.



Kuva 8: Viiden 30 minuutin testiajon 90. prosenttipisteen viiveen -kuvaajat valintaoperaatiolle, kun sovelluksen optimointiominaisuus oli poistettu käytöstä.

Ominaisuuden poistaminen käytöstä vaikutti yksittäisen testin aikana tapahtuneeseen värähtelyyn. Yksittäisen testiskenaarion mittapisteiden välinen erotus oli siis pienempi, koska optimointia ei tehty. Kuviin 7 ja 8 valittiin 90. prosenttipisteen kuvaajat, koska niissä värähtely oli hyvin havaittavissa. Kuvassa 8 on esitetty viisi testiajoa, jotka on toistettu muuten täysin samoilla parametreilla kuin kuvan 7 testiajot, mutta optimointiominaisuus on poistettu käytöstä. Kuvista nähdään, kuinka tuloksissa tapahtuva sahausliike on pienentynyt. Sen sijaan edelleen on selvästi näkyvissä, kuinka viive oli pienempi yhdessä viidestä skenaariosta.

7.1.4 Pidemmät skenaarioajot auttavat löytämään mittaustuloksia vääristävän tekijän

Tässä vaiheessa päätettiin ajaa pidempiä skenaarioita. Haluttiin nähdä, tasaantuuko suoritus ajan mittaan ja voidaanko pidempien testiajojen tuloksista havaita jotain, jota ei

nähdä puolen tunnin ajojen aikana. Samalla saatiin suoritettua kestotesti sekä itse järjestelmälle että testausjärjestelmälle. Jokaista operaatiota päätettiin toistaa jokaisella rasitustasolla viisi tuntia.

Koska skenaariot oli suunniteltu yksinkertaisiksi, niiden kestoa oli helppo muokata. Toistettiin vain operaatioita pidempään. Generaattori toistaa tilannetietoja loputtomiin, joten siihen ei tarvinnut tehdä muutoksia. Sen sijaan ongelmaksi muodostui nauhoite, jolla korvattiin yksi tilannetietogeneraattoreista. Nauhoitteen pituus oli vain noin puoli tuntia. Koska nauhoite käynnistettiin skenaarioiden osana, ongelma oli helppo kiertää Robot Frameworkin ominaisuuksilla. Toteutettiin Robot Framework -skenaarioille yhteinen silmukka-avainsana, jolle operaatio voitiin antaa parametrina. Silmukassa käynnistettiin nauhoite ja toistettiin tämän jälkeen parametrina saatua operaatiota puoli tuntia. Tämän jälkeen sama toistettiin. Skenaarioiden pituudeksi valittiin 5 tuntia ja skenaariot toistettiin jokaisella rasitustasolla. Kaikissa käytettiin samaa nauhoitetta.

Kun pitkien skenaarioiden vaatimat muutokset oli tehty, voitiin ne suorittaa. Suoritus kesti yhteensä 60 tuntia. Testattava järjestelmä ja testausjärjestelmä kestivät molemmat hyvin. Testauksen aikana järjestelmässä oli tapahtunut muutamaan kertaan kaksi poikkeusta, jotka eivät kuitenkaan aiheuttaneet järjestelmään pysyvää vikatilaa. Poikkeuksista kirjattiin vikaraportit, jotka on sittemmin jo korjattu. Myös pitkäaikaisissa testeissä oli selvästi näkyvissä, kuinka käyttäjäsovellus oli toiminut välillä hitaammin ja välillä nopeammin. Testeissä odotettiin näkyvän, kuinka joku rasitustaso tai operaatio on hidastumisen taustalla. Sen sijaan havaittiin hidastumisen ilmenevän ja poistuvan aina nauhoitteen uudelleenkäynnistyksen yhteydessä.

Jatkoselvityksessä ilmeni, että nauhoite ei toimi odotetusti. Generaattori, jonka nauhoitteen oli tarkoitus korvata, oli ollut jo hetken sivussa suorituskykytesteistä ja sen tuottama tieto oli korvattu toisella generaattorilla. Kuten aliluvussa 6.1 selitettiin, generaattorien tuottamat tilannetiedot eivät suoraan vastanneet toisiaan. Testattavan järjestelmän tietojen käsittelyyn oli tullut vika, jonka vuoksi kaikki nauhoitteen sisältämä tieto ei aina välittynyt käyttäjäsovellukseen. Tästä syystä käyttäjäsovelluksessa oli välillä enemmän ja välillä vähemmän tilannetietoa esitettävänä, joka taas johti testitulosten vääristymiseen. Tämä oli hyvä esimerkki siitä, miksi kattavat automatisoidut testit ovat niin tärkeitä. Kun asiaa testataan harvoin, voi hyvinkin merkittävä ongelma jäädä huomaamatta pitkäksi aikaa.

Tutkinnassa havaittiin toinenkin ongelma. Uusien skenaarioiden suunnittelussa oli haluttu käyttää tiettyä päivitysväliä, mutta generaattorissa oli virhe, jonka seurauksena päivityksiä oli moninkertaisesti enemmän kuin oli suunniteltu. Nauhoite nauhoitetaan uudelleen, kun saadaan korjatuksi vika, jonka takia nauhoitteeseen tuli liikaa päivityksiä.

Järjestelmän ongelmaa ei pystytty korjaamaan tämän työn aikana, joten nauhoite oli pakko hyllyttää käyttöliittymäviiveen mittarin testauksessa. Nauhoite päätettiin olla korvaamatta muulla kuormalla, koska nauhoite on tarkoitus ottaa osaksi skenaarioita mahdollisimman pian. Kuorma vaikuttaa käyttöliittymäviiveen määrään, mutta käyttöliittymäviive on olemassa kuorman määrästä huolimatta. Testeillä varmistettiin kuorman olevan riittävä ilman nauhoitetta, jotta erot skenaarioiden välillä ovat selkeästi nähtävissä tuloksissa. Myöhemmin esiteltävässä kuvassa 9 nähdään, kuinka skenaariot ovat selvästi erilaisia pienemmälläkin kuormalla.

Seuraavaksi ajettiin taas muutamia kymmeniä 30 minuutin testejä samalla skenaariolla. Nyt tulokset olivat yhtenevät. Käyttöliittymäviiveen mittarin testaus vaikutti mahdolliselta. Aiemmin mainittu käyttäjäsovelluksen piirtovälinoptimoija jätettiin pois käytöstä käyttöliittymäviiveen mittarin testauksen ajaksi, koska muuttujia haluttiin rajoittaa, jotta voitiin todentaa mittarin toimivuus. Kun suorituskykytestaus otetaan käyttöön osana jatkuvaa integraatiota, optimoija on päällä ja käytetään aina raskasta kuormitustasoa. Näissä testeissä tehtyjen havaintojen perusteella osataan selittää optimoijan aiheuttama suurempi vaihtelu mittaustuloksissa.

7.1.5 Kolmen kuukauden kehityksen tuottamat erot käyttöliittymän viiveeseen

Käyttöliittymäviivettä oli ajateltu verrattavan vanhemman version ja uusimman version välillä. Vanhaksi versioksi valittiin manuaalisissa suorituskykytesteissä käytetty versio tammikuulta 2017. Uusin versio oli siis huhtikuulta 2017, joten versioiden välissä oli noin kolme kuukautta kehitystyötä. Skenaariot oli toteutettu siten, että ne olivat ajettavissa myös vanhemmalla versiolla. Käyttäjäsovelluksen käynnistykseen oli tehty muutos, jonka huomioiminen yhden vanhan version testauksen tähden koettiin liian työlääksi. Vanha versio vaati käyttäjäsyötteen käyttöliittymäelementtiin ja uudessa oli toteutettu mahdollisuus antaa syöte komentoriviargumenttina sovellusta käynnistettäessä. Skenaariot toimivat uudella tavalla. Vanhasta versiosta tuotettiin uusi käännös, johon lisättiin yhden rivin muutos, jossa käyttäjäsyöte korvattiin koodivakiolla. Tämä mahdollisti testiskenaarion ajamisen vanhalla versiolla. Tämä nähtiin pienimmäksi vaivaksi, koska ei ollut tarkoitus testata useita vanhoja versioita, vaan verrata suorituskykymuutosta muutaman kuukauden aikana.

Versioiden välillä oli toteutettu uusia ominaisuuksia ja muutamia suorituskykyparannuksia. Toiveena oli, että suorituskykyparannukset näkyisivät myös testeissä. Näin ei kuitenkaan käynyt. Automaattisten suorituskykytestien tulokset olivat versioiden välillä lähes identtiset. Pelättiin käyttöliittymäviiveen mittarin olevan liian epätarkka huomatakseen eroja, jotka ihminen huomaa. Tässä vaiheessa manuaaliset suorituskykytestit aina suorittavaa testaajaa pyydettiin ajamaan suorituskykytestit ja kiinnittämään suurta huomiota

siihen, onko käyttöliittymä hänen mielestään hidastunut tai nopeutunut. Testaajalle ei kerrottu automaattitestien tuloksia, koska ei haluttu vaikuttaa hänen objektiivisuuteensa. Testaaja kirjasi raporttiinsa: *”Mielestäni operoinnin suhteen ei ollut havaittavissa selkeää muutosta edelliseen versioon.”*. Manuaalisen testauksen tulokset olivat siis linjassa käyttöliittymäviiveen mittarin kanssa.

7.1.6 Käyttöliittymän tahallinen hidastaminen

Koska vanhan ja uuden version välillä ei ollut merkittävää eroa käyttöliittymäviiveessä, piti löytää toinen tapa tuottaa järjestelmään selvästi havaittava viive. Koska edelleen uskottiin tapahtumajonon olevan avain viiveen mittaamiseen, keksittiin lisätä mittarin herätetapahtumiin viivettä. Herätetapahtumia tuotetaan 20 millisekunnin välein ja siinä missä normaalisti herätetapahtuman suoritus kesti häviävän pienen ajan, nyt niihin lisättiin 15 millisekunnin säikeen nukutus. Nukutuksen kesto löydettiin kokeilemalla. Tällä nukutuksen kestolla saatiin käyttäjäsovellus toimimaan sen verran hitaammin, että ihminen pystyi huomaamaan eron selvästi, mutta käyttöliittymästä ei tullut täysin käyttökelvoton, kuten esimerkiksi 18 millisekunnin nukutuksella.

Suorituskykytestaaja ja kaksi kehittäjää testasivat hidastettua ja normaalia käyttäjäsovellusta. Jokainen heistä osasi valita hitaamman version. Toinen kehittäjistä kommentoi hitauden tuntuvan ”*aidolta*” eli vastaavalta kuin hidastuminen, jota käyttäjäsovelluksessa on koettu kuormitustilanteessa. Tämän pohjalta myös suorituskykytestaajalta kysyttiin aitoudesta. Hänkin koki tuntuman olevan vastaava, mutta huomasi tahallisesti hidastetun sovelluksen olevan tasaisemmin hidas kuin normaali sovellus kuormitustilanteessa. Hidastumisen tasaisuudesta ei katsottu olevan haittaa viiveenmittarin testauksessa. Käytännössä mittari kuitenkin laskee tuloksia minuutin ajan kerrallaan, joten tuloksissa epätasainenkin hidastuminen tasoittuu, kun siitä lasketaan minuutin keskiarvo.

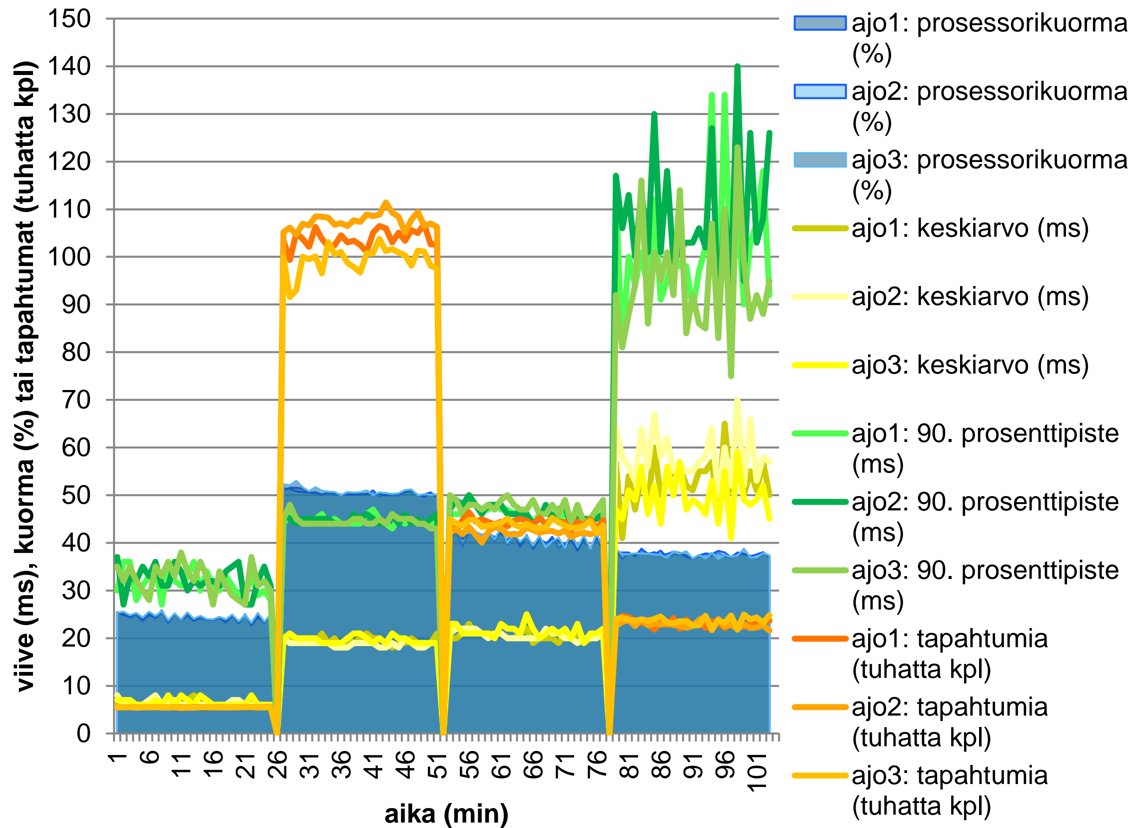
7.1.7 Käyttöliittymäviiveen mittarin tulosten vertailu normaalilla ja hidastetulla versiolla

Nyt hidastetulla ja normaalilla käyttäjäsovelluksella ajettiin suunnitellut neljä skenaariota. Ajoissa käytettiin raskasta kuormaa, josta oli jätetty nauhoite pois. Käyttäjäsovelluksen piirto-optimointi oli kytketty pois päältä.

Toistettavuuden vertaamiseksi testiajot suoritettiin kolme kertaa. Kuvassa 9 on esitetty kolme ajoa normaalilla käyttäjäsovelluksella. Ensimmäisenä vasemmalla on leposkenaario, sitten järjestyksessä näppäimistösyöte, kartan tarkennus ja karttaobjektien valinta.

Kuviin 9, 10 ja 11 on valittu 25 minuutin osuus skenaarioiden suorituksen keskeltä, jotta saatiin poistettua käynnistyksessä ja sammutuksessa tapahtuva tulosten vääristyminen. Skenaarioiden väliin on asetettu mittaustulos nolla, jotta skenaariot erottuvat selvästi. Ku-

viin on otettu vertailuarvoksi prosessorikuorma prosenttiyksikköinä, joka on esitetty sinisen sävyillä. Kuvissa on esitetty keltaisilla sävyillä viiveen minuutin keskiarvo millisekunteina. Vihreän sävyillä on esitetty viiveen 90. prosenttipisteen arvo millisekunteina. Oransseilla sävyillä on esitetty tapahtumien kokonaismäärä minuutissa. Tapahtumien määrä on näytetty tuhansina tapahtumina, jotta tapahtumat mahtuivat samalle y-akselille.



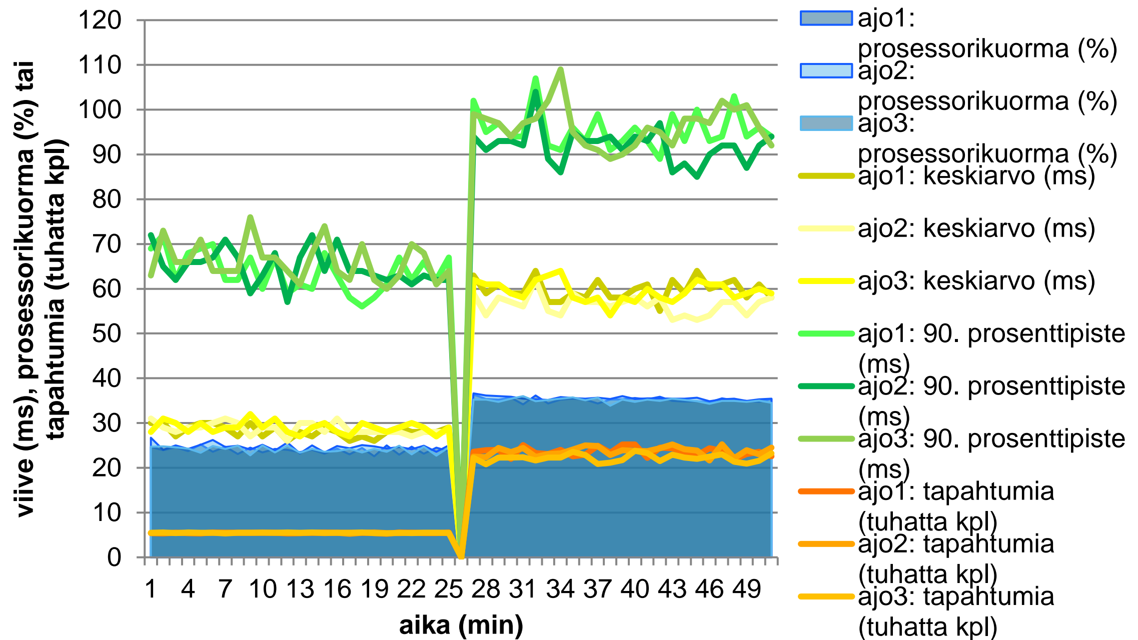
Kuva 9: Kolme testiajoa normaalilla versiolla. Jokaisessa ajossa suoritettu järjestyksessä lepo-, näppäimistösyöte-, tarkennus- ja valintaskenaariot.

Kuvassa 9 nähdään, kuinka toisiaan lähellä olevat värisävyt seuraavat toisiaan. Käyrät eivät ole suoria vaan mittapisteiden välillä tapahtuu selvää hajontaa, mutta tulokset operaatioiden välillä ovat selvästi erotettavissa toisistaan. Vaikka näppäimistösyötteen ja kartan tarkennuksen viiveen keskiarvot ovat melko samanlaiset, pystytään ne erottamaan helposti vertaamalla tapahtumamääriä.

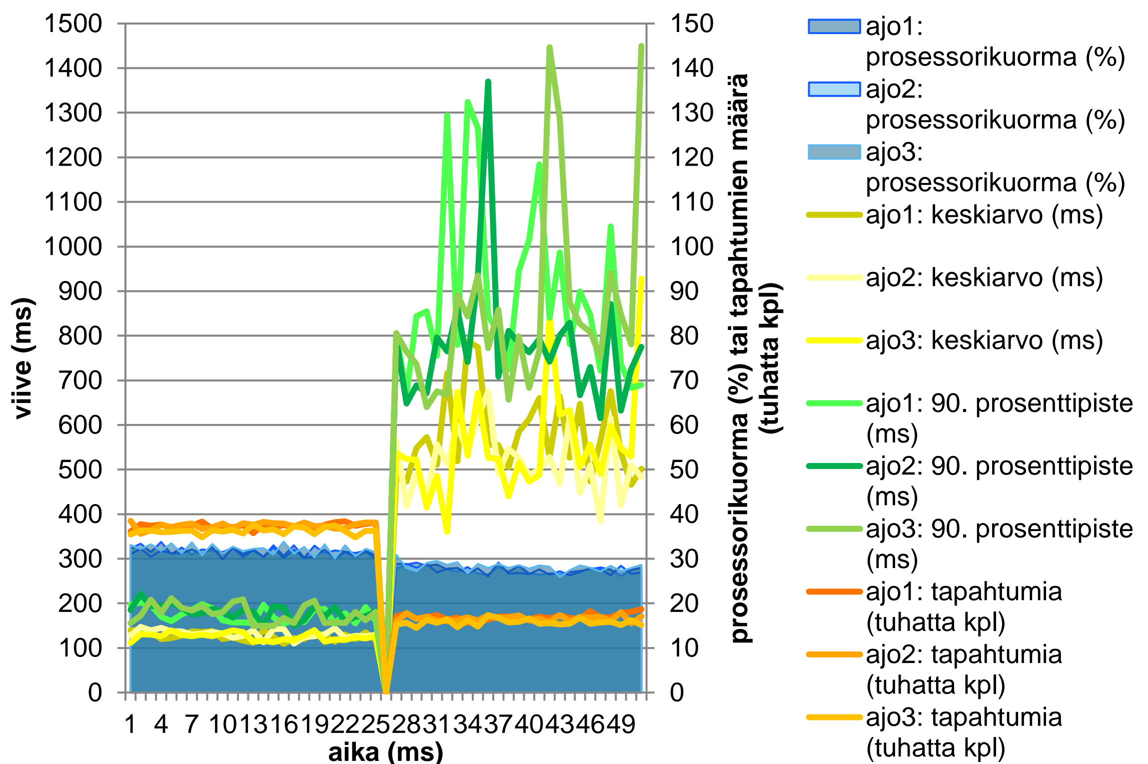
Kiinnostavaa kuvassa 9 on myös se, että prosessorikuorma ja viive eivät näytä olevan yhteydessä. Mitattu viive on selvästi isoin valintaskenaariossa, mutta prosessorikuorma on leposkenaarion jälkeen pienin. Tapahtumien määrä ei myöskään korreloi viiveen kanssa.

Hitaalla sovelluksella mitattu viive kasvoi niin merkittävästi, että selkeyden vuoksi skenaariot oli jaettava kahteen eri kuvaan. Kuva 10 esittää lepo- ja näppäimistöskenaarioiden mittaustulokset hidastetulla käyttäjäsovelluksella. Kuva 11 taas esittää tarkennus- ja valintaskenaarioiden mittaustulokset. Kuvasta 11 on tärkeää huomata, että kuvaan on lisätty

toinen akseli. Vasemmalla akselilla on esitetty viiveen (vihreät ja keltaiset kuvaajat) yksiköt ja oikealla akselilla prosessorikuorman (siniset) ja tapahtumamäärän (oranssit) yksiköt. Viive valintaskenaariossa oli niin suuri, että vasen akseli jouduttiin kasvattamaan kymmenkertaiseksi verrattuna kuviin 9 ja 10.



Kuva 10: Lepo- ja näppäimistösyöteskenaariot hidastetulla käyttäjäsovellusversiolla



Kuva 11: Tarkennus- ja valintaskenaariot hidastetulla käyttäjäsovellusversiolla

Ihmisen testatessa hidastettua sovellusta käyttöliittymä oli vielä käyttökelpoinen, mutta hidastuminen oli ilmeistä. Vertaamalla normaalin (kuva 9) ja hidastetun (kuvat 10 ja 11) käyttäjäsovelluksen mittaustuloksia, voidaan nähdä, että hidastuminen on selvästi havaittavissa kaikissa neljässä skenaariossa. Mitattu viive on kasvanut jopa kymmenkertaiseksi hidastetussa sovelluksessa.

Käyttöliittymäviiveen mittari ja ihmisen kokema hidastuminen kulkevat tuloksissa selvästi käsi kädessä. Ihmisen tekemissä testeissä aktiivisista operaatioista sulavimmaksi koettiin näppäimistösyötteet ja selvästi hitaimmaksi karttaobjektien valinta. Sama nähdään sekä normaalin että hidastetun käyttäjäsovelluksen mittaustuloksissa.

Hidastetussa sovelluksen prosessorikuorma ei ole noussut, vaikka viive on kasvanut. Tämä johtuu tavasta hidastaa sovellusta. Säikeen lepuutus ei käytä prosessoriaikaa, vaikka se viivästää operaatioiden suoritusta. Mutta jo normaalista sovelluksesta voitiin huomata, että prosessorikuorma ja käyttöliittymän vasteaika eivät ole suoraan riippuvaisia toisistaan. Jos prosessori kuormitettaisiin äärirajoille, on todennäköistä, että myös vasteaika nousee, mutta vasteaika voi nousta, vaikka prosessorikuorma laskee. Näiden havaintojen pohjalta voidaan luottavaisesti sanoa, että käyttöliittymäviivemittari todellakin mittaa käyttöliittymän hidastumista, kuten käyttäjä sen kokee. Mitatut tulokset vastasivat suoraan käyttökokemusta operaatioiden välillä ja normaalin ja hidastetun käyttäjäsovelluksen välillä.

7.1.8 Muun suorituskykytestausjärjestelmän toiminta

Käyttöliittymäviive oli ainut mittari, jonka mittaustulosten oikeellisuus oli tarpeen todentaa. Muille mittareille riitti, että varmistetaan tulosten tallentuvan oikein, koska mittarit olivat olleet käytössä jo aiemmin. Läpimenoajan, Oracle-tietokantatietojen ja TeamQuestin resurssien käyttötietojen keräystä tarkkailtiin käyttöliittymäviiveen mittarin testauksen aikana. Tulokset kerättiin luotettavasti tietokantaan läpi testauksen ja niitä pystyttiin tarkkailemaan TeamQuest View'sta ongelmitta. Tulokset olivat yhteneviä suorituskertojen välillä. Tuloksia ei haluttu tarkastella tarkemmin osana tätä diplomityötä, koska tulokset liittyvät enemmän johtamisjärjestelmän testaukseen kuin työn aiheena olevien automaattisten suorituskykytestien toimivuuden todentamiseen.

Kun läpimenoajan mittarilta saadut tulokset kerätään TeamQuest-mittausagentilla minuutin välein tietokantaan, saadaan samalla kierrettyä aliluvussa 4.3 esitelty ongelma, jossa käyttäjälle esitettyihin tuloksiin päätyi merkityksetön mediaanien mediaani. Nyt jokaiselta minuutilta esitetään aito mediaani.

Kaikki aliluvussa 7.1 suoritetut testit suoritettiin täysin toteutettujen Jenkins-tehtävien avulla. Jenkins-tehtävissä on laskuri, joka laskee suorituskertoja. Laskuri ei erittele onnistuneita ja epäonnistuneita suorituksia, mutta siitä voi saada kuvan testauksen laajuudesta. Testauksen aikana testien suoritustehtävää käynnistettiin lähes 400 kertaa. Laajin

suoritustehtävä sisälsi 20 eri skenaariota ja pisin kesti yli 60 tuntia. Asennustehtävää ja testiajokokonaisuustehtävää suoritettiin molempia melkein 200 kertaa. Lukujen pohjalta voidaan siis katsoa Jenkins-tehtävien tulleen hyvin testatuiksi.

7.2 Työn arviointi vaatimusten valossa

Työssä onnistuttiin automatisoimaan koko ketju yöllisen version luonnista aina suorituskykymittaustulosten automaattiseen keräykseen asti. Mittaustulosten esittämistä kehittäjille halutaan vielä parantaa, josta lisää luvussa 7.3. Kokonaisuutena tämä tärkein vaatimus täytettiin mainiosti.

Testien toistettavuudessa saavutettiin tyydyttävä taso. Oli valitettavaa, että tilannetietonauhoitteen kanssa ilmeni ongelmia, joita ei ehditty diplomityön aikana korjata. Ongelmat olivat itse testattavassa järjestelmässä ja sen generaattoreissa. Testijärjestelmän toteutuksen tarkoitus on havaita juuri tällaisia ongelmia, joten tässä mielessä saavutettiin hyödyllisiä tuloksia. Ilman nauhoitetta kerätyt tulokset osoittavat toistettavuuden olevan testausjärjestelmän ja mittarien osalta hyvällä mallilla.

Kehityksen kestävyys huomioitiin operaatioiden ja kuormituksen suunnittelussa. Koska testausjärjestelmä on juuri otettu käyttöön, ei ole vielä todistettu sen olevan kehityksen kestävä. Operaatioiden valinnassa ja toteutuksessa tämä oli tärkein lähtökohta. Valittiin todella yksinkertaiset operaatiot, joiden ei uskota muuttuvan, ja ne toteutettiin siten, että kartan tarkennusta lukuun ottamatta ne olisi helppo siirtää kokonaan toiseen vastaavaan järjestelmään. Näistä syistä on toivottavaa, että ne kestävät kehityksen ja samoja operaatioita voidaan käyttää koko järjestelmän elinkaaren ajan.

Lisätavoitteena oli pystyä ajamaan testit myös vanhemmilla versioilla. Tiedettiin järjestelmän käynnistyksen muuttuneen viime vuoden lopulla niin radikaalisti, että sen vanhempia versioita ei olisi mahdollista testata ilman suurta ylimääräistä työpanosta. Tammikuun versiolla testiskenaariot pystyttiin ajamaan hyvin pienellä muutoksella ajettavaan versioon. Tähän oltiin tyytyväisiä, koska sillä pystyttiin jo vertaamaan viimeisen kolmen kuukauden aikana tapahtuneita muutoksia.

Laajennettavuus otti ison roolin testausjärjestelmän kehityksessä. Jälkikäteen arvioituna erityisesti testiajoympäristön vaihdon helpottamiseen käytettiin liikaakin aikaa. Vaatimus toteutuu selvästi, koska skenaarioihin toteutettiin yhteinen pohja, jonka päälle voi rakentaa muita skenaarioita helposti. Kuormitustason voi määrittää yhdellä parametrilla. Jopa kolmannen osapuolen Robot Framework RemoteSwingLibrary-laajennosta parannettiin, jotta se olisi helpommin laajennettavissa. Lisäksi kaikki muuttujat ja polut on toteutettu siten, että ne toimivat missä tahansa ympäristössä. Kehitys- ja testausympäristöön on toteutettu omat täysin automatisoidut asennusohjelmansa, jotka hoitavat testausjärjestelmän pystytyksen. Kehittäjä tai testaaja joutuu itse asentamaan vain Pythonin ja Robot Frameworkin ympäristöönsä. Kaikki muu hoituu automatisoidusti.

Käyttöliittymän viiveenmittariin kasvoi testauksen aikana vahva luotto. Testauksen alku oli kivistä, kun ei tiedetty johtuvatko ongelmat mittarista vai järjestelmästä. Kun ongelmien syyt löydettiin ja viiveenmittaria testattiin lisää, saatiin hyvää näyttöä, että mittarin esittämät tulokset ovat suoraan verrannollisia käyttäjäkokemuksen kanssa. Tahallisesti hidastettua järjestelmää testasi ihminen ja automatisoitu testausjärjestelmä, ja molemmissa hitaampi järjestelmä erottui selvästi. Samalla havaittiin, että resurssien käyttöasteesta, kuten prosessoriajasta, hidastumista ei voinut päätellä.

7.3 Jatkokehitys

Työtä tehdessä saatiin monta kehitysideaa, jotka rajattiin pois nyt toteutettavasta kokonaisuudesta. Alle on kerätty asioita, joita toteutetaan tulevaisuudessa tarpeiden mukaisessa järjestyksessä.

Tilannetietonauhoite suunniteltiin osaksi suorituskykytestausta, mutta se jouduttiin jättämään pois työn aikana tehdystä testauksesta, koska sekä nauhoitteessa että testattavassa järjestelmässä ilmeni ongelmia. Nauhoite nauhoitetaan uudestaan ja liitetään takaisin osaksi suorituskykytestien kuormitusta heti, kun järjestelmässä ja generaattorissa havaitut ongelmat on saatu korjattua.

Tulosten julkaisua parannetaan jatkossa varmasti. Kehittäjille on tärkeää esittää yksinkertaisesti suorituskyvyn kehitys. Tämä toteutetaan määrittelemällä tietyt raja-arvot, joita käyttöliittymäviiveen ja läpimenoajan keskiarvot eivät saa ylittää. Raja-arvojen määritystä varten halutaan kerätä ensin tuloksia kymmenistä versioista, jotta voidaan nähdä yleinen vaihtelu tulosten välillä. Määrityksessä hyödynnetään aliluvussa 2.2 käsiteltyä tietoa ihmisen viiveensietokyvystä. Jos raja-arvot ylitetään, merkataan tuloksia keräävä Jenkins-tehtävä joko epävakaaksi tai korkeamman raja-arvon ylittyessä epäonnistuneeksi. Tehtävä määritellään ilmoittamaan ylityksestä sähköpostilla kehittäjille, jotka olivat liittäneet tuotoksiaan versionhallinnan päähaaraan edellisen kelvollisen version ja suorituskykyä heikentäneen version välillä.

Myös skenaarioita kehitetään jatkossa lisää. Monta järjestelmän ominaisuutta jää nyt suunnitelluissa suorituskykytesteissä testaamatta, koska keskityttiin ominaisuuksiin, joita voidaan testata järjestelmän kehittyessä. Automaattitestien kehitys on otettu osaksi projektin jatkuvaa kehitystä. Ne suoritetaan työssä suunnitellun automaation avulla. Lisäksi tietyille ominaisuuksille kehitetään omat suorituskykyskenaariot. Nämä skenaariot eivät ole yhtä muuttumattomia kuin tässä työssä suunnitellut, vaan niitä on tarkoitus sopeuttaa kehitykseen. Myöskään niiden tuloksia ei ole tarkoitus tarkastella yhtä pitkällä aikavälillä, vaan ymmärretään kehityksen vaikuttavan selvästi tuloksiin.

Käyttäjän liikkeiden autenttiseen jäljittelyyn nauhoitettujen käyttöliittymäskenaarioiden avulla heräsi työn aikana kiinnostusta, mutta ne rajattiin työn ulkopuolelle. Nauhoitettuja

käyttäjän liikkeitä voidaan hyödyntää edellisessä kappaleessa kuvattuun tietyn ominaisuuden testaukseen. Nauhoitteen toiston liittäminen testausjärjestelmään on yksinkertaista, koska nauhoitteen toisto voidaan käynnistää suoraan Robot Frameworkilla.

Kaksi mittaria, jotka nousivat esiin työn aikana, mutteivät mahtuneet työn piiriin ovat DDS:n ja Java-virtuaalikoneen muistinkäytön kerääminen TeamQuestin tietokantaan. Molempien toteutus on jo selvillä. DDS:n muistinkäyttöä voidaan tarkkailla OpenSplice DDS -tuotteen mukana tulevalla mmstat-työkalulla. Java-virtuaalikoneen muistinkäyttöä voidaan taas tarkkailla jstat-työkalulla, joka tulee virtuaalikoneen asennuspaketin mukana. Molemmat ovat komentorivityökaluja, joten niistä pitäisi voida kerätä suoraan tietoja TeamQuestiin.

Projektin käytössä on JProfiler-työkalu, joka voidaan yhdistää käynnissä olevaan Java-sovellukseen ja kerätä metodikohtaista tietoa sovelluksen suorituskyvystä [42]. Tulevaisuudessa olisi kiinnostavaa tutkia, kuinka JProfilerin käyttö voitaisiin automatisoida siten, että havaittaessa testatun version olevan hitaampi käynnistetään JProfiler ja yhdistetään se käyttäjäsovellukseen tai sovelluspalvelimeen ja toistetaan testiskenaario. JProfilerin keräämät tulokset voidaan tuottaa verkkosivumuotoon ja liittää muiden tulosten mukaan. Näin kehittäjille saataisiin automaattisesti tieto siitä, mitkä metodit järjestelmässä veivät eniten prosessoriaikaa hidastuneessa järjestelmässä.

8. YHTEENVETO

Diplomityössä kuvataan, kuinka asiakas-palvelin-mallin sovelluksen suorituskykytestaus automatisoitiin ja liitettiin osaksi jatkuvaa integraatiota. Työn tärkein vaatimus oli koko testiketjun automatisointi ja jokaöisten kehitysversioiden suorituskykytestaus. Lisäksi kerätyistä suorituskykytestauksen tuloksista piti pystyä tulkitsemaan suorituskyvyn kehitys järjestelmässä. Tulosten piti vastata toisiaan samalla versiolla ja muuttua suorituskyvyn mukana. Osa käytetyistä mittareista toteutti tämän vaatimuksen ennestään. Käyttöliittymäviiveen mittari toteutettiin työn aikana ja testattiin automatisoiduilla suorituskykytesteillä. Järjestelmän tuli olla myös helposti jatkokehitettävissä.

Suorituskykytestaus eristettiin omaan ympäristöönsä, jossa käyttäjäsovellus, sovelluspalvelin ja tietokanta saivat kukin omat fyysiset koneensa. Suorituskykytestausympäristöä hallittiin jatkuvan integraation palvelimen tehtävillä, jotka suorittivat komentorivikäskyjä sovelluspalvelin- ja käyttäjäsovelluskoneilla. Robot Framework-testityökalulle toteutettiin suoritettavat skenaariot, joiden avulla hallittiin käyttäjäsovellusta ja tietojen tuottamista sovelluspalvelimelle.

Aikataulu oli niin tiukka, että testattavassa järjestelmässä havaitun virheen aiheuttama viivästys johti tulosten keräysvaiheen siirtymiseen diplomityön ulkopuolelle. Tulosten keräysvaiheen tarkoitus oli löytää suorituskykymittausten tuloksille tietyt raja-arvot, joiden ylittyessä voidaan varoittaa kehittäjiä järjestelmän hidastumisesta. Tämä on järjestelmäkohtaista, joten yleishyödyllisyyden kannalta tulosten keräys ei ole merkityksellistä. Yleisesti raja-arvojen valintaan voidaan hyödyntää diplomityön teoriaosuudessa käsiteltyä ihmisen viiveensietokykyä.

Työlle asetetut vaatimukset saavutettiin kiitettävästi. Koko suorituskykytestausprosessi saatiin automatisoitua ja käyttöliittymäviiveelle toteutettiin mittari, jonka mittausten oikeellisuudesta saatiin vahva näyttö.

LÄHTEET

[1] A.B. Bondi, Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice, Addison-Wesley Professional, Boston, Massachusetts, USA, 2014, 448 p.

[2] C.U. Smith, L.G. Williams, Performance Solutions: A Practical Guide To Creating Responsive, Scalable Software, Addison-Wesley, Boston, Massachusetts, USA, 2001, 544 p.

[3] L. Vanhala, VR:n lippujärjestelmä meni nurin. Se on vain yksi esimerkki isoista it-mokista. Suomen Kuvalehti, Vol. 2012, Iss. 3, 2012, s. 30-37. Saatavissa (viitattu 29.4.2017): https://suomenkuvalehti.fi/jutut/kotimaa/nain-vr-sotki-lippujarjestelmansa-miksi-it-projektit-epaonnistuvat/.

[4] K. Systä, OHJ-02300 Johdatus ohjelmistotuotantoon, Tampereen teknillisen yliopiston ohjelmistotekniikan laitos, Tampere, luentokalvosarja, 2013, 41 s. Saatavissa (viitattu 29.4.2017): http://www.cs.tut.fi/~systa/JOTU2013/01JohdantoWeb.pdf.

[5] S. Oaks, Java Performance: The Definitive Guide, O'Reilly, Sebastopol, California, USA, 2014, 426 p.

[6] A. Adamoli, D. Zaparanuks, M. Jovic, M. Hauswirth, Automated GUI performance testing, Software Quality Journal, Vol. 19, Iss. 4, 2011, pp. 801-839.

[7] E. Macedo Rodrigues, d.O. Moreira, L. Teodoro Costa, M. Bernardino, A.F. Zorzo, S.S. do Rocio, R. Saad, An empirical comparison of model-based and capture and replay approaches for performance testing, Empirical Software Engineering, Vol. 20, Iss. 6, 2015, pp. 1831-1860.

[8] R.D. Craig, S.P. Jaskiel, Systematic software testing, Artech House, Boston, Massachusetts, USA, 2002, 536 p.

[9] A. Martens, H. Koziolek, L. Prechelt, R. Reussner, From monolithic to component-based performance evaluation of software architectures, Empirical Software Engineering, Vol. 16, Iss. 5, 2011, pp. 587-622.

[10] D. Westermann, R. Krebs, J. Happe, Efficient Experiment Selection in Automated Software Performance Evaluations, in: N. Thomas (ed.), Computer Performance Engineering: 8th European Performance Engineering Workshop, EPEW 2011, Borrowdale, UK, October 12-13, 2011. Proceedings, Springer Berlin Heidelberg, Berlin, Germany, 2011, pp. 325-339.

[11] A. Sabetta, H. Koziolek, Performance Metrics in Software Design Models, in: I. Eusgeld, F.C. Freiling, R. Reussner (ed.), Dependability Metrics: Advanced Lectures, Springer Berlin Heidelberg, Berlin, Germany, 2008, pp. 219-225.

[12] J. Nielsen, Usability engineering, Morgan Kaufmann, San Francisco, California, USA, 1993, 362 p.

[13] R. Miller, Response Time in Man-computer Conversational Transactions, Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I, ACM, San Francisco, California, USA, pp. 267-277.

[14] B. Shneiderman, Designing the user interface: strategies for effective human-computer interaction, 3rd ed. Addison-Wesley, Reading, Massachusetts, USA, 1998, 639 p.

[15] S.C. Seow, Designing and Engineering Time: The Psychology of Time Perception in Software, Addison-Wesley Professional, Upper Saddle River, New Jersey, USA, 2008, 224 p.

[16] R.A. Doherty, P. Sorenson, Keeping Users in the Flow: Mapping System Responsiveness with User Experience, Procedia Manufacturing, Vol. 3, 2015, pp. 4384-4391.

[17] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, The Java Virtual Machine Specification, Java SE 8 Edition, Oracle America, Inc., Redwood City, California, USA, 2015, 600 p.

[18] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition, Oracle America, Inc., Redwood City, California, USA, 2015, 768 p.

[19] M.P. Bates, PIC Microcontrollers: An Introduction to Microelectronics, 3rd ed. Newnes, Oxford, UK, 2011, 456 p.

[20] M.L. Scott, Programming Language Pragmatics, 2nd ed. Elsevier Science, Saint Louis, Missouri, USA, 2005, 915 p.

[21] Johtamisjärjestelmät - Insta Defsec Oy, Insta Defsec Oy, verkkosivu. Saatavissa (viitattu 29.4.2017): https://www.instadefsec.fi/fi/ratkaisut/puolustusratkaisut/johtamisjarjestelmat.html.

[22] J. Gosling, H. McGilton, The Java Language Environment (White paper), Sun Microsystems, Inc, Mountain View, California, USA, 1995, 84 p.

[23] OSGi Core Release 5, OSGi Alliance, San Ramon, California, USA, 2012, 396p. Saatavissa (viitattu 30.4.2017): https://osgi.org/download/r5/osgi.core-5.0.0.pdf.

[24] Data Distribution Service (DDS), 1.4th ed., Object Management Group, Inc., Needham, Massachusetts, USA, 2015, 168 p. Saatavissa (viitattu 30.4.2017) http://www.omg.org/spec/DDS/1.4/PDF/.

[25] C. Kemper, Foundation Version Control for Web Developers, Apress, Berkeley, California, USA, 2012, 406 p.

[26] J. McAllister, Mastering Jenkins, Packt Publishing, Birmingham, UK, 2015, 334 p.

[27] F. Klassen, Tcpreplay Overview, AppNeta, verkkosivu. Saatavissa (viitattu 30.4.2017): http://tcpreplay.appneta.com/wiki/overview.html.

[28] S. Bisht, Robot Framework Test Automation, Packt Publishing, Birmingham, UK, 2013, 98 p.

[29] TeamQuest View User Guide, TeamQuest Corporation, Clear Lake, Iowa, USA, 2012, 344 p. Saatavissa (viitattu 30.4.2017): https://public.support.unisys.com/c71/docs/teamquest/17.200/viewuserguide.pdf.

[30] M. Välimäki, Tiedon läpimenoajan mittaaminen hajautetussa johtamisjärjestelmässä, Tampereen teknillinen yliopisto, Tietotekniikan laitos, Tampere, 2012, 42 s.

[31] S. Tatham, PuTTY User Manual, verkkosivu. Saatavissa (viitattu 30.4.2017): https://the.earth.li/~sgtatham/putty/0.69/htmldoc/.

[32] DB-Engines Ranking - popularity ranking of database systems, DB-Engines, verkkosivu. Saatavissa (viitattu 30.4.2017): https://db-engines.com/en/ranking.

[33] K. Ormbrek, Jrobotremoteserver - Userguide, Github, Inc, verkkosivu. Saatavissa (viitattu 30.4.2017): https://github.com/ombre42/jrobotremoteserver/wiki/User-Guide.

[34] C.F.A. Johnson, J. Varma, Pro Bash Programming: Scripting the GNU/Linux Shell, 2nd ed. Apress, Berkeley, California, USA, 2015, 237 p.

[35] J. Friesen, Beginning Java 7, Appress, Berkeley, California, USA, 2011, 920 p.

[36] J. Härkönen Extending SwingLibrary, Github, Inc, verkkosivu. Saatavissa (viitattu 30.4.2017): https://github.com/robotframework/SwingLibrary/wiki/Extending-SwingLibrary.

[37] M. Heap, Ansible: From Beginner to Pro, Apress, Berkeley, California, USA, 2016, 170 p.

[38] S. van Vugt, Beginning the Linux Command Line, 2nd ed. Apress, Berkeley, California, USA, 2015, 399 p.

[39] Robot Framework - User Guide, Robot Framework Foundation, verkkosivu. Saatavissa (viitattu 30.4.2017): http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html.

[40] Jenkins Node and Label parameter, the Jenkins project, verkkosivu. Saatavissa (viitattu 30.4.2017): https://plugins.jenkins.io/nodelabelparameter.

[41] TeamQuest - Our Customers, TeamQuest Corporation, verkkosivu. Saatavissa (viitattu 30.4.2017): http://www.teamquest.com/en/about-us/our-customers/.

[42] O. Oransa, Java EE 7 Performance Tuning and Optimization, 1st ed. Packt Publishing, Birmingham, UK, 2014, 555 p.