Tampereen yliopisto

Vili Jaakko Saura

MIKROPALVELUPOHJAISEN JÄRJESTELMÄN AUTOMAATTITESTAUKSEN HALLINTA

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Hannu-Matti Järvinen
Markus Nenonen
Maaliskuu 2021

TIIVISTELMÄ

Vili Jaakko Saura: Mikropalvelupohjaisen järjestelmän automaattitestauksen hallinta
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-tutkinto
Maaliskuu 2021

Mikropalveluiden automaattitestausta voidaan suorittaa erilaisilla testausohjelmistoilla. Testausohjelmistoja käyttämällä mikropalveluille voidaan antaa oikeita käyttötilanteita mukailevia syötteitä ja viestejä, jotka mikropalvelun tulisi tulkita ja prosessoida oikein. Tämänkaltaisessa testauksessa mikropalvelun toiminnasta saadaan selkeämpi kuva kuin matalamman tason testauksessa ja erilaisilla testisyötteillä on mahdollista simuloida vaihtelevia tilanteita. Tällä tavalla suoritettava toiminnallisten ja ei-toiminnallisten ominaisuuksien testaus parantaa mikropalvelujärjestelmien laatua, kun virhetilanteet voidaan havaita ja korjata ennen palveluiden julkaisua.

Testattavan järjestelmän koon kasvaessa ja sen toiminnallisuuden monimutkaistuessa tarvitaan myös enemmän ja monimutkaisempia testausohjelmistoja. Testausohjelmistojen toiminnallisuuden pitää olla säädettävissä, testien tilan tulee olla seurattavissa ja mahdolliset virhetilanteet tulee kirjata, jotta niiden myöhempi analysointi ja korjaaminen olisi mahdollista. Laajempien testien suorittaminen ilman keskitettyä hallintajärjestelmää on työlästä ja aikaa vievää, jos eri testausohjelmistot pitää asentaa ja valmistella yksittäin. Tässä työssä suunnitellaan ja kehitetään hallintajärjestelmä, jonka avulla eri testiohjelmistoja voidaan käsitellä ja ohjata keskitetysti. Järjestelmä kattaa testidatan luomisen, testausohjelmistojen orkestroinnin sekä erinäisten testeihin liittyvien lokien kirjoittamisen ja yhdistämisen.

Työn tuloksena on toimiva testausjärjestelmä, jota on helppo käyttää ja jolla voidaan suorittaa erikokoisia ja erityyppisiä testejä. Järjestelmään voidaan kehittää uusia testausohjelmistoja Valmiiksi määriteltyjen rajapintojen sekä korkean tason toiminnallisuuden perusteella. Testaajat voivat suorittaa testaamisen kaikki vaiheet järjestelmä avulla, ja useiden erilaisten testitapausten samanaikainen suorittaminen ja hallinta on myös mahdollista. Testausjärjestelmän omat lokit kerätään keskitetysti, jotta niiden myöhempi tutkiminen olisi mahdollisimman yksinkertaista. Koska testausjärjestelmä on mikropalvelupohjainen, sitä voidaan laajentaa helposti esimerkiksi järjestelmän monitorointiin liittyvillä mikropalveluilla.

Avainsanat: Mikropalvelu, kontti, orkestraatio, automaattinen testaus

ABSTRACT

Vili Jaakko Saura: Controlling automatic tests for microservice based systems
Master's Thesis
Tampere University
Information technology
March 2021

The automatic testing of microservices can be done using different types of testing programs. These programs can send data and inputs that simulate the real use of the microservices, and the microservices should process this data correctly. This kind of testing can provide a clear picture of the status and functionality, especially compared to lower-level testing. Different types of test inputs and test data can be used to simulate different real-life scenarios. This kind of functional and non-functional testing improves the quality of microservice based systems, as different errors can be found and fixed before the microservice is deployed to real environment.

As the size and complexity of the system grows, so does the amount and complexity of the testing programs. The tester should be able to modify the outputs of the testing programs, monitor the state of the tests and potential errors should be logged clearly for later analysis. Managing larger tests without a dedicated control system can be challenging and time consuming, as the tester needs to install and configure the testing programs separately. In this thesis work we plan and develop a control system that can be used to manage and control different test programs. The control system can be used to generate the test data, orchestrate the different test programs, and collect the generated logs for later analysis.

The developed testing system is easy to use, and it can control tests of different sizes and types. The system can be expanded with new test programs based on the documented interfaces and high-level functionality. The tester can manage all phases of testing using this control system and they can control multiple different types of tests at the same time. The logs of this system are collected centrally, so they could be accessed easily during or after the tests. As the testing system itself is also based on microservices, new features can be added easily as a separate microservice.

Keywords: Microservice, container, orchestration, automated testing

ALKUSANAT

Yliopistotaipaleeni on ollut pitkä, mutta loppu häämöttää jo. Ajoittain työmäärä oli miltei musertava, mutta kaikesta huolimatta kurssit ovat paketissa ja diplomityökin on valmiina. Kiitän työn ohjaajaa ja valvojaa hyvästä palautteesta koko työn teon ajalta. Ilman heitä työ olisi ollut sekavampi ja vaikealukuisempi. He pitivät myös kirjoitusmotivaatiota yllä hyvän palautteen ja kommenttien avulla. Kiitokset myös kollegoilleni ja erityisesti Petrixin jäsenille. He auttoivat työn oikolukemisessa ja antoivat erinomaisia ideoita työn sisällöstä. Simulaattorien valmisteluun kuluneet tunnit ja kontrollerin käyttö tositilanteissa konkretisoivat työn tuomia hyötyjä. Kiitos kuuluu myös ystävilleni, jotka lukivat työ keskeneräisiä versioita ja auttoivat kirjoituskammon kanssa, kun tekstiä ei tuntunut syntyvän. Kiitän vielä perhettäni jatkuvasta tuesta, erityisesti korkeakoulu opintojeni aikana.

Tampereella, 21.3.2021

Vili Jaakko Saura

SISÄLLYSLUETTELO

KUVALUETTELO

LYHENTEET JA MERKINNÄT

E2E	engl. End to End, koko järjestelmän kattava testaustapa
JSON	engl. JavaScript Object Notation, tiedostoformaatti
REST	engl. Representational state transfer, verkkokutsuja käyttävä ohjelmistoarkkitehtuuri tyyli
VAJ	verkko analyysijärjestelmä

.

1. JOHDANTO

Mikropalvelupohjaisten järjestelmien yleistymisen myötä kasvaa myös tarve tällaisten järjestelmien testaamiselle. Järjestelmien koosta ja monimutkaisuudesta riippuen testejä voi olla mahdollista suorittaa manuaalisesti, mutta useimmissa tilanteissa tämänkaltainen testaaminen ei ole tarpeeksi kattavaa. Erityisesti ei-funktionaalisessa testauksessa manuaalinen testaus ei välttämättä edes ole varteenotettava vaihtoehto. Automatisoimalla testausta testejä voidaan suorittaa ilman testaajan valvontaa tai suoraa syötettä. Automatisoimalla yksinkertaiset testit niitä voidaan suorittaa nopeammin kuin manuaalisia, ja monimutkaisemmissa testeissä automatisointi voi pitää huolta testin valvonnasta ja virhetilanteiden keräämisestä.

Nokia kehittää mikropalveluista koostuvaa verkkodatan analyysijärjestelmää, johon viitataan tässä diplomityössä lyhenteellä VAJ (verkko analyysijärjestelmä). VAJ:n toimintaa ja suorituskykyä testataan erilaisilla mikropalveluilla, joita kutsutaan simulaattoreiksi. Simulaattorit simuloivat oikeita käyttötapauksia ja lähettävät sopivanmuotoista testidataa VAJ:n mikropalveluille. Simulaattorien toiminta on automaattista, mutta niiden valmistelu ja hallinta ei ole. Tämä manuaalinen testien valmistelu hidastaa testien suorittamista, erityisesti suuremmissa testeissä, joihin voi kuulua kymmeniä tai satoja simulaattoreita. Simulaattorien määrän kasvaessa niiden hallitseminen ja monitorointi muuttuu myös vaikeammaksi.

Tässä työssä tehdään konstruktiivinen tutkimus yllä mainitun ongelman ratkaisemiseksi. Tarkoituksena on luoda uusi järjestelmä, jonka avulla simulaattorien valmistelu ja hallinta olisi automatisoitua ja helppoa. Tämän hallintajärjestelmän suunnittelussa otetaan huomioon niin mikropalvelupohjaisten järjestelmien ominaisuuksia kuin VAJ:n testaamiseen liittyviä haasteita. Työn lopputuloksena on hallintajärjestelmän prototyyppi, joka käyttää hyödykseen mikropalveluiden hallintamenetelmiä.

Luvussa 2 käsitellään mikropalvelupohjaisia järjestelmiä ja näiden tärkeimpiä ominaisuuksia. Tämä luo teoreettista pohjaa hallintajärjestelmän suunnittelua ja toimintaa varten. Luvussa 3 käsitellään VAJ:n nykyinen testaustapa ja -ohjelmisto. VAJ:n testaamiseen liittyvät ongelmat ja puutteet käsitellään myös tarkemmin tässä luvussa, ja ne toi-

mivat luotavan hallintajärjestelmän oletusvaatimuksina. Hallintajärjestelmän vaatimukset, arkkitehtuuri ja käytettävät tekniikat käydään läpi luvussa 4. Järjestelmän eri osat ja näiden toiminta esitellään kyseisen luvun aliluvuissa.

Luku 5 käsittelee hallintajärjestelmän toimintaa ja komentoja esimerkkien avulla. Järjestelmän osien hallinta ja erilaisten lokien luominen käydään myös läpi. Kehitetyn toteutuksen toiminta ja onnistuminen arvioidaan luvussa 6. Toteutusta verrataan hallintajärjestelmälle annettuihin vaatimuksiin ja mikropalveluiden testaamisen teoreettiseen pohjaan.

Tutkimuksen aikana hallintajärjestelmälle ilmeni uusia toiminnallisuustarpeita ja tekniikoita. Nämä hallintajärjestelmälle suunnitellut jatkokehitysominaisuudet on kuvattu luvussa 7. Jatkokehitykseen kuuluu esimerkiksi orkestroinnin ja monitoroinnin parantaminen, sekä järjestelmän laajempi käyttöönotto erilaisissa mikropalveluiden testeissä.

2. MIKROPALVELUARKKITEHTUURI JA SEN OMINAISPIIRTEET

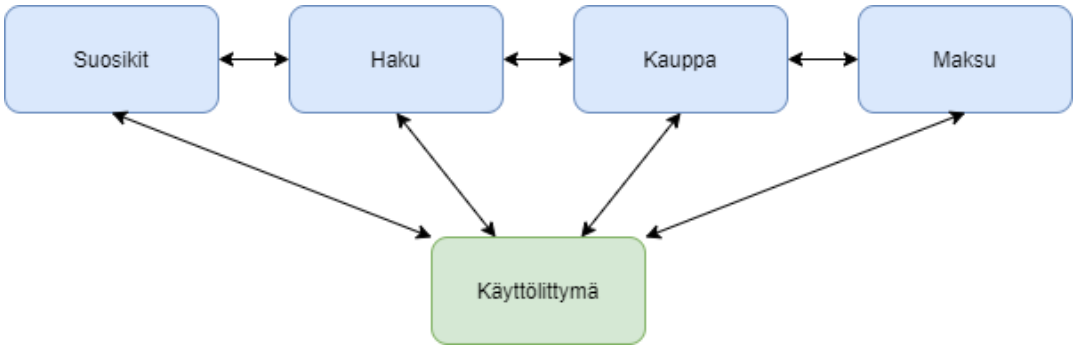
VAJ: koostuu mikropalveluista ja sen testauksessa käytetään myös mikropalveluita. Tämän takia on hyvä ymmärtää mikropalveluiden yleisimmät ominaisuudet ja käyttötavat. Koska kehitettävä järjestelmä keskittyy testaukseen, käymme myös läpi mikropalveluiden testauksen haasteita ja esittelemme kontit (engl. container), joiden sisällä mikropalveluita on mahdollista ajaa erilaisissa ympäristöissä.

Mikropalveluiden suosio on kasvanut viime vuosien aikana ja monet suuret yritykset ovat siirtyneet käyttämään mikropalveluita omissa ohjelmistoissaan [1-3]. Mikropalveluiden avulla ohjelmistojen toiminnallisuutta on mahdollista skaalata tarpeiden mukaan, uusien ominaisuuksien ja päivitysten kehittäminen on nopeaa ja koko järjestelmä kestää virhetilanteita paremmin. Tämä mahdollistaa nopean ohjelmistojen toimituksen sekä ketterämmän projektiympäristön, joka voi mukautua muuttuviin tarpeisiin ja tilanteisiin [1,4].

2.1 Johdatus mikropalveluihin

Mikropalvelut ovat pieniä, itsenäisesti toimivia ohjelmia, jotka erikoistuvat tiettyyn toiminnallisuuteen. Yhdistämällä useita mikropalveluita voidaan luoda monimutkaisia järjestelmiä, joiden yhteistoiminta perustuu mikropalveluiden väliseen kommunikointiin ja yhteistyöhön. Tällaisia järjestelmiä on mahdollista kehittää alusta alkaen mikropalvelupohjaiseksi, tai jo olemassa olevia monoliittijärjestelmiä voidaan muuttaa käyttämään mikropalveluja pilkkomalla toimintoja sopivan kokoisiksi mikropalveluiksi [2,4,5].

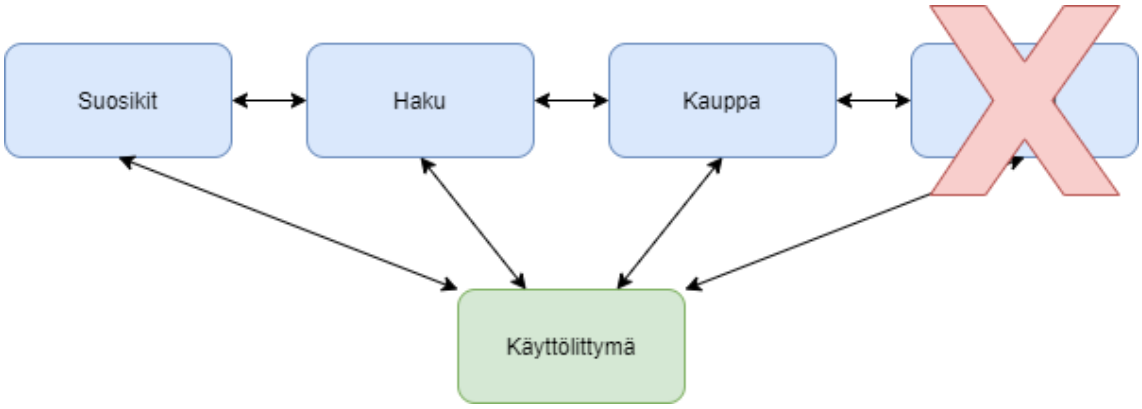
Vaikka mikropalvelut ovat nimensä mukaisesti pieniä ohjelmia, niiden kokoa ei ole virallisesti määritelty koodirivien tai työmäärän perusteella. Eri ohjelmointikielet voivat vaatia eri määrän koodirivejä saman toiminnallisuuden toteuttamiseksi, ja käytettävät kirjastot voidaan myös laskea osaksi ohjelman kokoa. Kokenut ohjelmoija kehittää palvelun nopeammin kuin kokematon, vaikka lopputuloksen toiminta olisi sama molemmissa tilanteissa. Nyrkkisääntönä mikropalvelu on sopivan kokoinen, jos se voidaan kirjoittaa uudelleen muutamassa viikossa [4]. Vastaavasti mikropalvelu ei saisi olla liian pieni. Liian pienet ja erikoistuneet mikropalvelut tekevät järjestelmästä monimutkaisemman ja hajautetumman. Erilaisten virhetilanteiden ja pullonkaulojen paikantaminen on näin ollen vaikeampaa ja järjestelmän ylläpitoon kuluu enemmän aikaa [1,5]



Kuva 1. *Yksinkertainen mikropalveluista koostuva järjestelmä*

Kuvassa 1 esitetään yksinkertainen mikropalveluista koostuva verkkokauppasovellus. Eri mikropalvelut on kuvattu suorakaiteina ja palveluiden välinen tiedonsiirto tapahtuu kuvan nuolien mukaisesti. Jokaisella palvelulla on selkeä toiminnallisuus ja vastuu, kuten tuotteiden haku, suosikkituotteiden tallentaminen tai tuotteiden maksaminen. Koska palveluiden toiminnallisuudet ovat suhteellisen yksinkertaisia ja erikoistuneita, niiden kehittäminen on helppoa ja voidaan asettaa yksittäisen tiimin vastuulle. Myös koko järjestelmän arkkitehtuuri pysyy yksinkertaisena, kun erityyppisten mikropalveluiden määrä ei ole tarvittua suurempi [1,4,5]

Mikropalveluiden pieni koko ja erikoistuminen johtavat palveluiden itsenäisyyteen. Yhden palvelun toiminta ei siis suoraan vaikuta toisen palvelun toimintaan tai ominaisuuksiin, esimerkiksi esimerkki kaupassamme tuotteiden haku ei vaikuta suosikkeihin tallennettuihin tuotteisiin. Mikropalvelun sisäisen tiedon tulee olla piilotettua muiden palveluiden näkökulmasta ja tiedon jakaminen palvelun ulkopuolelle tapahtuu palveluiden välisten verkkokutsujen (engl. network call) kautta. Itsenäisyyden ansiosta mikropalveluja voidaan poistaa tai lisätä ilman, että tämä vaikuttaisi muun järjestelmän toimintaan. Oletuksena uuden palvelun käyttöönoton ei pitäisi vaatia muokkauksia muiden mikropalveluiden toiminnallisuuteen. Itsenäisyyden takia yksittäisen mikropalvelun virheellinen toiminta tai rikkoutuminen ei myöskään estä koko järjestelmän toimintaa. Hajonnut palvelu voidaan eristää, korjata ja ottaa uudelleen käyttöön samalla, kun muu järjestelmä jatkaa toimintaansa [2,4,6].



Kuva 2. *Mikropalvelujärjestelmä viallisilla palveluilla*

Kuvassa 2 yhdessä järjestelmän palvelussa on ilmennyt vika ja se on poistettu. Verkkokauppasovellusta voidaan käyttää normaalisti, mutta poistetun mikropalvelun toiminnallisuus on hetkellisesti poissa käytöstä. Kun poistettujen elementtien vika on korjattu, ne voidaan asentaa takaisin ympäristöön, jolloin koko järjestelmä palaa kuvan 1 mukaiseen muotoon. Mikropalveluiden itsenäisyys saa mikropalveluista koostuvat järjestelmät kestämään yksittäisissä palveluissa tapahtuvia virhetilanteita [4].

Mikropalveluiden välinen kommunikointi tapahtuu erilaisten rajapintojen ja verkkokutsujen kautta, jolloin palvelu ei pääse suoraan käsiksi toisen palvelun tietoihin. Tämä auttaa itsenäisyyden saavuttamisessa ja helpottaa myös uusien palveluiden kehittämisessä tai vanhojen palveluiden päivittämisessä, kun kommunikaatioon tarkoitetut rajapinnat on määritelty valmiiksi. Koska palvelut liittyvät toisiinsa vain löyhästi näiden kutsujen ja rajapintojen kautta, palvelun sisäistä logiikkaa voidaan muuttaa radikaalisti ilman että se vaikuttaa koko järjestelmän toimintaan, kunhan nämä palveluiden väliset yhteydet pysyvät yhtenäisinä. Esimerkkijärjestelmämme Haku-palvelun sisäinen toiminta voidaan muuttaa hyvinkin erilaiseksi, kunhan sen rajapinnat vastaavat muiden mikropalveluiden käyttämiä rajapintoja. Ilman tätä mikropalveluiden välistä löyhää kytkentää (engl. loose decoupling) palvelut olisivat liian sidoksissa toisiinsa ja menettäisimme miltei kaikki mikropalvelupohjaisten järjestelmien tuomat edut, kuten skaalautuvuuden, kestävyyden ja itsenäisyyden [4,7].

Mikropalvelupohjaisia järjestelmiä on myös helppoa skaalata käytön ja tarpeen mukaan. Mikropalveluiden itsenäisyys ja erikoistuminen tarkoittavat, että tarvittaessa voimme lisätä vain järjestelmän tarvitsemia mikropalveluja. Jos järjestelmän tietyn osan käyttö kasvaa hetkellisesti, kyseisen osan palveluiden määrää voidaan nostaa vastaamaan tätä uutta tarvetta. Vastaavasti mikropalveluita voi skaalata alaspäin, kun kyseisille palveluille ei ole tarvetta. Jos esimerkkijärjestelmämme käyttäjien määrä kasvaa, voimme joutua

lisäämään kauppa-mikropalveluita vastaamaan tätä uutta tarvetta. Vastaavasti jos suosikki-mikropalveluiden käyttö laskee voimme vähentää niiden määrää. Palveluiden määrää skaalaamalla on mahdollista optimoida koko järjestelmän käyttämiä resursseja ja reagoida muuttuviin olosuhteisiin [2,4]. Skaalautuminen onkin yksi mikropalveluiden suurimmista eduista [1].

Löyhän kytkennän ansiosta mikropalveluiden toteutuksessa voidaan käyttää eri teknologioita [4]. Tämän teknologia-agnostisuuden ansiosta jokainen palvelu voidaan toteuttaa juuri kyseiseen toiminnallisuuteen parhaiten soveltuvalla tavalla, eikä eri ohjelmointikielten tai kirjastojen suhteen tarvitse tehdä kompromisseja. Tämä mahdollistaa myös uusien teknologioiden ja käyttötapojen nopean omaksumisen, kun näiden vaatimia muutoksia ei tarvitse tehdä kuin suhteellisen pieneen osaan järjestelmästä. Helppo käyttöönotto takaa myös nopean palautteen uusien toteutusten toimivuudesta. Teknologioita valitessa kannattaa kuitenkin painottaa kokoa ja keveyttä, jotta mikropalvelut pysyisivät pienikokoisina [1,2]. Esimerkkijärjestelmässämme eri palvelut voivat käyttää erilaisia tietokantoja riippuen palvelun toiminnasta ja tallennettavien tietojen käytöstä.

Mikropalveluiden käyttöön liittyy kuitenkin myös ongelmia. Mikropalveluiden toiminnallisuuden ja koon määrittely voi epäonnistua, jolloin mikropalvelut ovat liian suuria tai pieniä. Tämä heikentää mikropalveluiden tuomia etuja, kuten skaalautumista. Huonosti suunnitellut rajapinnat ja kommunikaatiotavat voivat hidastaa koko järjestelmän toimintaa, tai aiheuttaa myöhempiä ongelmia järjestelmän laajentuessa. Koska mikropalveluita voidaan kehittää tai päivittää nopeasti, niille ja koko järjestelmälle pitää suorittaa testejä uusien versioiden myötä. Koko järjestelmä on myös jakautuneempi verrattuna yksittäiseen, isoon ohjelmistoon, jolloin yksittäisten mikropalveluiden hallinta ja virhetilanteiden paikantaminen vaikeutuu. Suuri määrä mikropalveluja kuluttaa myös enemmän resursseja kuin vastaavan toiminnallisuuden kattava yksittäinen ohjelma, johtuen mikropalveluiden välisestä kommunikaatiosta.[1,4-6]

2.2 Mikropalveluiden testaus ja sen haasteet

Mikropalveluita ja mikropalvelujärjestelmien toimintaa testatessa tulee ottaa huomioon niiden ominaisuuksista seuraavia haasteita. Mikropalveluille on mahdollista suorittaa manuaalista testausta, mutta nopean kehityksen ja vaihtelevien mikropalveluyhdistelmien vuoksi automaattitestauksen käyttö on tehokkaampaa. Erityisesti matalan tason testejä voidaan vaatia satoja tai tuhansia, ja niiden ajaminen manuaalisesti ei aina ole mahdollista [4].

Automaatiotestausta on helppo suorittaa osana kehitysputkea (engl. deployment pipe), jossa uudelle mikropalvelun versiolle suoritetaan erikseen määritellyt automaattitestit. Yleensä tämä kehitysputki toteutetaan DevOps-tekniikkaa mukaillen, jonka avulla mikropalvelu ja sen testit voidaan ajaa halutussa ympäristössä. Tähän kehitysputkeen voidaan myös lisätä erilaisia lokitus- ja monitorointitapoja [8].

Erityisesti yksikkötestausta ja palvelutestausta voidaan suorittaa automaattitestauksen avulla. Mikropalveluiden yksikkötestaus ei juurikaan eroa muiden tyyppisten ohjelmien yksikkötestauksesta, koska testit keskittyvät palvelun sisäisen toiminnan testaamiseen. Tämänkaltainen testaus on helpointa suorittaa erilaisten tynkien (engl. stub) avulla [4]. Palvelutestauksessa (engl. service test) testataan yksittäisen palvelun toimivuutta, joka mikropalveluiden tapauksessa tarkoittaa yhtä mikropalvelua. Testattavan palvelun pitäisi olla eristettynä, jotta testien aikana löydetyt virheet ja ongelmatilanteet voidaan olettaa kuuluvan juuri testattavaan palveluun. Palvelutestaus tulisi suorittaa ennen mikropalvelun käyttöönottoa tai ennen isomman kokoluokan testausta [4,9]. Laajempien mikropalvelujärjestelmien hyväksymistestaus voi olla huomattavasti monimutkaisempaa kuin vastaavan toiminnallisuuden kattava monoliittijärjestelmä, johtuen mikropalveluiden välisistä yhteyksistä ja riippuvuuksista [6].

Mikropalvelut voivat myös vaatia tietoja tai yhteyksiä toisilta mikropalveluilta. Tätä varten voidaan luoda erilaisia tynkiä ja matkijoita (engl. mock) [10,11], jotka imitoivat oikeiden palveluiden toimintaa. Tyngät palauttavat aina samanlaisen vastauksen riippumatta palvelun tekemien kutsujen määrästä, kun taas matkija mukailee oikean palvelun toimintaa ja voi vastata eri tavoin tilanteen mukaan. Valinta tynkien, matkijoiden tai oikeiden elementtien käytön välillä riippuu testattavasta ominaisuudesta ja palvelusta. Esimerkiksi järjestelmän tilamuutoksia voidaan testata tynkien avulla, mutta tarkempi käytöksen testaus voi vaatia matkijan [10]. Matkijat käyttö on suositeltavaa erityisesti tilanteissa, joissa oikean elementin käyttö ei ole mahdollista, esimerkiksi tietokantojen tapauksessa, tai monimutkaista toiminnallisuutta testatessa [11].

Koko mikropalvelun toimivuutta testataan päästä päähän -testauksessa (engl. End-to-End. Lyh. E2E). E2E-testit vastaavat usein palvelun oikeita käyttötilanteita ja yksittäisten komponenttien oletetaan läpäisseen yksikkö- ja palvelutestit. E2E-testaus kattaa kokonaisen järjestelmän ja osa haasteista tulee vastaan jo ennen testauksen aloittamista. Ongelmia tuottaa esimerkiksi testattavien mikropalveluiden versioiden valinta. Eri käyttötapauksissa ja eri asiakkailla voi myös olla käytössä erilainen mikropalveluiden yhdistelmä, jolloin jokainen eri yhdistelmä tulisi testata. Toisaalta tällä tavalla saatetaan testata sama toiminnallisuus useaan kertaan, mikä kasvattaa testien suorittamiseen kuluvaa aikaa. Kaikkien yhdistelmien testaaminen on aikaa vievää ja testit kattaisivat osittain

samoja asioita. Tämänkaltaisia ongelmia varten on mahdollista käyttää erilaisia testausmenetelmiä, kuten regressiotestausta tai vihreä/sininen käyttöönottoa (engl. blue/green deployment) [4,12]

Mikropalveluille voidaan suorittaa myös sopimustestausta (engl. contract testing). Sopimustestauksessa mikropalveluiden välille määritellään rajapintojen lisäksi yhteinen sopimus, jossa sovitaan palveluiden välinen toiminta korkealla tasolla. Jos sopimuksessa määritellyt ehdot eivät täyty testauksen aikana, mikropalvelu on automaattisesti viallinen. Sopimustestaus asettuu E2E– ja hyväksymistestauksen välimaastoon [13,14].

Kaikille ohjelmistoille voidaan myös suorittaa ei-toiminnallista testaamista (engl. non-functional testing) joka kattaa esimerkiksi palvelun viiveen, käyttäjäkapasiteetin mittaamisen tai järjestelmän turvallisuuden. Mikropalvelujärjestelmä voi läpäistä E2E-testit mutta jos sen toiminta ei ole halutulla tasolla, se ei ole valmis julkaistavaksi. Nämä ei-toiminnalliset vaatimukset vaihtelevat mikropalvelusta ja projektista riippuen: joissain tilanteissa pidempi viive on hyväksyttävää ja toisissa taas resurssien käyttö on tarkasti rajoitettu. On tärkeää testata juuri oman palvelun vaatimuksia ja suorittaa tätä testausta tarpeeksi aikaisessa vaiheessa projektia, jotta testien tuloksia voidaan käyttää hyväksi mikropalvelun kehityksessä ja potentiaaliset ongelmat voidaan korjata [4,9].

Ei-toiminnalliseen testaukseen kuuluu mm. suorituskykytestaus (engl. performance test), kuormatestaus (engl. load test) ja stressitestaus (engl. stress test). Tällaisilla testeillä on mahdollista saada kattavampaa kuvaa järjestelmän toiminnasta erilaisten kuormien ja epätavallisten käyttötilanteiden kanssa. Verrattuna toiminnalliseen testaukseen ei-toiminnallinen testaus vaatii enemmän aikaa ja valmistelua, esimerkiksi testidatan ja tarkkailujärjestelmien valmistelussa. Vaikka ei-toiminnallisten ominaisuuksien kattava testaus on suositeltavaa ja usein pakollista, erinäisistä syistä se ei aina ole mahdollista halutulla tasolla. Esimerkiksi aika- ja budjettirajoitteet tai tekniset ongelmat vaikuttavat ei-toiminnallisen testauksen suorittamiseen [15,16].

Korkean tason testit, kuten E2E- ja ei-toiminnallinen testaus vaativat enemmän valmistelua kuin matalamman tason automaattitestit. Tämä valmistelu kattaa testiympäristöjen sekä testi- että tarkkailutyökalujen valmistelun. Testien suorittaminen oikeanlaisissa ympäristöissä on erityisen tärkeää: jos testiympäristö ei vastaa resursseiltaan ja toiminnaltaan palvelun oikeaa käyttöympäristöä testien tulokset eivät näytä järjestelmän oikeaa toimintaa [15]. Korkean tason testeissä voidaan käyttää erikoistuneita testaustyökaluja, joilla voidaan hallita saapuvan kuorman määrää tai aiheuttaa erilaisia virhetilanteita järjestelmässä [9,17]. Testien tulokset voivat vaatia lisäanalyysia tai valvontaa, mikä voi vaatia erilaisia monitorointityökaluja. Tarkkailemalla järjestelmän toimintaa koko testin

ajalta on mahdollista löytää pullonkauloja tai virhetilanteita, joita muuten ei olisi löydetty [16].

Mikropalvelujärjestelmien monitorointi on tärkeää myös testien ulkopuolella. Koska mikropalvelut ovat oletuksena hajautettuja niissä tapahtuvien virhetilanteiden alkuperän paikantaminen voi olla vaikeaa. Monen palvelun samanaikainen toiminta voi aiheuttaa rinnakkaisuuteen liittyviä ongelmia esimerkiksi tietokantoja lukiessa tai niihin kirjoitettaessa. Ympäristössä tapahtuva virhe, kuten verkkoyhteyden menettäminen, voi myös aiheuttaa järjestelmästä riippumattoman virhetilanteen. Mikropalvelupohjaiset järjestelmät vaativatkin miltei poikkeuksetta lokitusjärjestelmän, jonka avulla virhetilanteiden ja ohjelman kulun seuraaminen on mahdollista [1,3,4]. Lokien rakenteen tulee olla selvästi suunniteltu, jotta se kuvaisi tarkasti järjestelmän toimintaa. Esimerkiksi samaan tapahtumaan liittyvät verkkokutsut ja mikropalveluiden toimet voidaan tallettaa saman ID:n alle [4]. Näitä lokeja ja muita järjestelmään liittyviä tietoja on mahdollista tarkkailla erilaisilla monitorointiohjelmilla, joidenka avulla tiedot voidaan visualisoida eri mittarien ja aikavälien suhteen.

Kuva 3. *Grafana [18] ohjelmalla luotu tietojen visualisaatio*

Eri tietoja voidaan visualisoida esimerkiksi kuvan 3 mukaisella Grafana-ohjelmalla luodulla paneelilla. Tällä tavalla käyttäjä voi tarkkailla järjestelmän useita osa-alueita keskitetysti ja mahdolliset ongelmat, kuten yllättävät resurssinkulutukseen liittyvät piikit erottuvat helposti. Tietoja kannattaa analysoida piikkien lisäksi kokonaisuuden suhteen, sillä joissain tilanteissa yksittäiset piikit ovat suvaittavissa eivätkä ne vaikuta järjestelmän toimintaan pitkällä aikavälillä [4].

2.3 Mikropalveluiden erilaiset ympäristöt

Mikropalveluita voidaan toteuttaa erilaisilla teknologioilla ja mikropalvelujärjestelmien toiminta perustuu palveluiden väliseen kommunikointiin. Eri teknologioiden käytöstä voi kuitenkin seurata tilanteita, joissa yksittäinen palvelin tai ympäristö ei voi tukea kaikkia järjestelmän mikropalveluita. Mikropalveluita testattaessa ja kehitettäessä voidaan myös tarvita muiden palveluiden asentamista, mikä vie aikaa kehitykseltä ja testaamiselta, jos sopivia ympäristöjä ei ole tarjolla. Itsenäisiä mikropalveluita olisi myös hyvä pystyä siirtämään yhtenä, kaiken tarpeellisen kattavana pakettina. Näitä puutteita on mahdollista korjata käyttämällä virtualisaatiota ja kontteja (engl. container) [19-21].

Virtualisaatio pohjautuu isäntäpalvelimen jakamiseen pienempiin osiin. Eri osia voi ajatella pienempinä palvelimina, joilla kaikilla voi olla omat käyttöjärjestelmänsä ja prosessinsa. Tällä tavalla on mahdollista luoda sopivia mikropalveluympäristöjä ja käyttää olemassa olevavia palvelinresursseja tehokkaasti. Yhdellä fyysisellä palvelimella voidaan ajaa useita pienikokoisia mikropalveluita, jotka muodostavat yhtenäisen järjestelmän [19].

Kuva 4. *Kuvaus virtuaalikoneesta [4] mukaillen*

Kuvassa 4 on palvelin, joka on jaettu kolmeksi virtuaalikoneeksi. Jokaisella virtuaalikone on oma käyttöjärjestelmänsä, niiden sisällä on myös oma käyttöjärjestelmän ydin (engl. kernel) ja erillinen virtuaalikoneiden valvoja (engl. virtual machine monitor, hypervisor) pitää huolen resurssien jakamisesta virtuaalikoneiden välillä. Sekä virtuaalikoneet että mikropalvelut voivat toimia itsenäisesti ja ulkopuolinen pääsy niiden sisältöön on rajattu [4,19]

Virtuaalikoneiden haittapuolena on kuitenkin palvelimen jakamisesta aiheutuvat resurssikustannukset. Jokainen virtuaalikone vaatii oman ytimensä ja käyttöjärjestelmänsä isäntäpalvelimen ytimen ja käyttöjärjestelmän lisäksi. Mitä pienempiin osiin isäntäpalvelin pilkotaan, sitä enemmän resursseja kuluu pelkästään virtuaalikoneiden ajamiseen riippumatta niiden sisällöstä [4]. Virtuaalikoneet eivät myöskään poista täysin mikropalveluiden asentamiseen liittyvää aikaongelmaa, sillä tarvittavat virtuaalikoneet täytyy myös määritellä ja asentaa erikseen.

Virtuaalikoneiden resurssiongelma voidaan ratkaista konttien (engl. container) avulla. Kontteja voi ajatella kevyinä virtuaalikoneina: ne eivät asenna kokonaan uusia käyttöjärjestelmiä, vaan jakavat isäntäpalvelimen prosessitilan kontin tarpeen mukaan. Koska kontit ovat virtuaalikoneita kevyempiä, niiden asentaminen ja käyttöönotto on nopeampaa ja helpompaa erilaisissa testi- ja asiakasympäristöissä. Tämän ansiosta konttipohjaista järjestelmää on myös nopea skaalata tarpeiden mukaan [4,20-22].

Kuva 5. *Kuvaus konteista [4] mukaillen*

Kuvassa 5 kolme palvelimelle asennettua konttia. Vertaamalla kuvia 4 ja 5 näemme yksittäisen kontin vievän vähemmän resursseja kuin virtuaalikone, kun erillistä käyttöjärjestelmän ydintä ei tarvita. Kontit eivät myöskään vaadi valvojaa, mikä vapauttaa myös isäntäjärjestelmän resursseja [4]. Konttien välinen erottelu perustuu usein Linux-ominaisuuksien käyttöön, kuten nimiavaruuden jakamiseen [20,22]. Yksittäisen kontin tulisi sisältää ohjelmiston lisäksi kaikki sen tarvitsemat riippuvuudet, kuten kirjastot. Kontin asentamisen lisäksi ajoympäristöön ei siis tarvitse tehdä muita muutoksia ja monimutkainenkin mikropalvelujärjestelmä on mahdollista asentaa pelkkien konttien avulla [20-22]. Kontteja voi ajatella mustina laatikkoina (engl. black box), jolloin käyttäjien ei tarvitse tuntea kuin kontin rajapinnat sen sisältämää ohjelmistoa käytettäessä. Kontteja voidaan käyttää ja hallita erilaisilla tähän tarkoitukseen suunnitelluilla ohjelmilla, kuten Dockerilla [22,23]. Konttien määrän kasvaessa niiden hallinta käsin muuttuu kuitenkin vaikeammaksi, ja tätä hallintaa varten on kehitetty erilaisia orkestraatio tekniikoita. Orkestraation

avulla on mahdollista hallita konttien elinkaarta, valmistelua ja toimintaa. Sopivan orkestraatiomenetelmän käyttö helpottaa suurien konttipohjaisten järjestelmien käyttöä [20,21]

Sekä virtualisaatio että kontit tuovat monia etuja mikropalveluiden kehittämiselle ja käytölle. Mikropalveluiden pakkaaminen ja asentaminen helpottuu kaiken tarvittavan sisältävän kontin avulla. Konttien elinkaarta on helppo valvoa ja hallita, jolloin uusia tai päivitettyjä mikropalveluita on helppo lisätä ja testata [21]. Kontit ovatkin erityisen suosittuja pilvipalveluissa ja DevOps-tekniikoissa ja niiden käyttö voi nopeuttaa sekä mikropalveluiden kehitystä että käyttöönottoa [1,8,20].

3. VAJ:N TESTAUS

Tässä luvussa esitellään VAJ:n rakenne ja sen ei-funktionaalisien ominaisuuksien testausmenetelmät. Testaus perustuu matkijoita [11] muistuttavien mikropalveluiden käyttöön erilaisia käyttötilanteita mukailevissa ympäristöissä. Testaukseen liittyy kuitenkin erilaisia haasteita ja ongelmia, jotka vaikuttavat testien suorittamiseen ja testitulosten laatuun. Haasteina on mm. testauksen valmisteluun kuluva aika, testauksessa käytettyjen ohjelmistojen erilaisuus ja sopivan testidatan hankkiminen. Luku perustuu useiden ei-funktionaalisten testien aikana todettuihin havaintoihin ja käytännön kokemuksiin. Diplomityötä varten tutustuttiin myös eri testauspalveluiden dokumentaatioon ja toimintaan käytännössä.

3.1 Kuvaus VAJ:sta

VAJ on mikropalveluista koostuva verkkodata-analyysijärjestelmä. Vaikka tuote koostuu monista mikropalveluista, tässä työssä keskitytään vain ulkoista verkkodataa vastaanottaviin palveluihin ja niiden ei-funktionaalisten ominaisuuksien testaamiseen. Näitä ulkoisesta lähteestä saapuvaa dataa vastaanottavia palveluita kutsutaan työssä vastaanottajiksi. Vastaanottajia on monen tyyppisiä erilaisia datatyyppejä varten ja VAJ:ssä voi olla useampia samaa dataa käsitteleviä vastaanottajia riippuen vastaanotettavan datan määrästä. VAJ:tä kehitetään jatkuvasti vaatimusten mukaan ja siihen lisätään uusia vastaanottajatyyppejä uusia datatyyppejä varten. Jo olemassa oleville vastaanottajille tehdään myös päivityksiä ja virhekorjauksia.

Saapuva data on peräisin erilaisista verkkoelementeistä ja datan lähetystapa sekä muoto vaihtelevat elementtien välillä. Data voi saapua vastaanottajiin tietyin väliajoin tai jatkuvana tapahtumavirtana. Verkkoelementin ja vastaanottajan välinen yhteys perustuu asiakas/palvelinyhteyteen (engl. client/server [24]) jossa palvelin tarjoaa asiakkaalle asiakkaan vaatimia palveluja. Tämän yhteyden roolit voivat kuitenkin vaihdella verkkoelementin mukaan. Verkkoelementtien tuottaman datan muoto on ennalta tiedossa, mutta siihen voi tulla ajoittain muutoksia, esimerkiksi jos dataan lisätään uusia tietokenttiä tai salausmenetelmiä.

Järjestelmän testausympäristö ja testattavat vastaanottajat vaihtelevat asiakkaiden käyttötapausten ja suoritettavien testityyppien mukaisesti. Testiympäristöjen saatavuus ja käytettävyys vaikuttavat myös siihen, missä ympäristössä testit suoritetaan ja mitä re-

sursseja testillä on käytössä. Testattavat mikropalvelut asennetaan sopivaan testausympäristöön, jonka jälkeen testauksessa käytettävät simulaattorit voidaan asentaa ja konfiguroida. VAJ lähettää tietoja tilastaan ja yksittäisen vastaanottajien tilanteesta erilliseen monitorointiohjelmaan. Tällä tavalla vastaanottajien toimintaa voidaan tarkkailla helposti testien ja oikean käytön aikana.

Asiakkaan tavallisessa käytössä tuotteen vastaanottajat vastaanottavat verkkodataa, käsittelevät datan ja lähettävät sen järjestelmän muille mikropalveluille. Oikeassa käyttötilanteessa vastaanottajalle saapuva data on voitu koota useasta eri lähteestä ja saapuvan datan määrä on tuhansia tapahtumia (engl. event) sekunnissa. Tämä datamäärä voi nousta ajoittain moninkertaiseksi ”kiiretuntina” (engl. busy hour), jolloin dataa tuotetaan tavallista enemmän. Vastaanottajien pitääkin siis pystyä käsittelemään näitä datapiikkejä sekä tarvittaessa palautumaan niiden aiheuttamista ylikuormitustilanteista. Kiiretuntien datamäärät eivät saisi aiheuttaa suuria pullonkauloja tai viiveitä järjestelmän toiminnassa, tai järjestelmän tulisi vähintäänkin pystyä ilmoittamaan ja reagoimaan näihin datan määrästä seuraaviin ongelmiin. Saapuva data voi myös olla virheellistä, mikä ei saisi myöskään aiheuttaa palautumatonta virhetilannetta elementissä tai järjestelmässä.

3.2 Testausohjelmisto

Vastaanottajien testausta varten on kehitetty testauspalveluita, joita kutsutaan simulaattoreiksi. Simulaattorit on suunniteltu simuloimaan erilaisten verkkoelementtien toimintaa ja ne muistuttavat toiminnaltaan matkijoita [11]: simulaattoreilla voi simuloida erilaisia käyttötilanteita ja datamääriä, mutta ne ovat huomattavasti yksinkertaisempia kuin oikeat verkkoelementit. Simulaattorien ja vastaanottajien välinen yhteys muodostetaan samalla tavalla kuin oikeiden verkkoelementtien kanssa, ja simulaattorien lähettämä data vastaa verkkoelementtien tuottamaa dataa. Simulaattorien lähettämää data voidaan muokata testaajien toimesta ja joissain tapauksissa datan lähetysnopeutta on myös mahdollista muokata.

Simulaattorit on suunniteltu vastaanottajien pitkäaikaiseen testaukseen, esimerkiksi suorituskykytestaukseen [4,15]. Näissä tilanteissa simulaattorit voivat olla käynnissä useita päivä tai viikkoja, lähettäen ennalta määriteltyä datakuormaa. Simulaattoreita voidaan myös käyttää lyhyemmissäkin testeissä, kuten osana DevOps automaattitestausta [8].

3.2.1 Simulaattorien toiminta

Jokaista erityyppistä verkkoelementtiä kohden on kehitetty vastaava simulaattori. Yleensä verkkoelementtiä vastaavan vastaanottajan kehittämä tiimi on myös kehittänyt

tarvittavan simulaattorin. Joissain tapauksissa simulaattori on kehitetty huomattavasti aikaisemmin ja vastaanottajan kehittäjätiimi toimii vain simulaattorin ylläpitäjänä. Kaikille verkkoelementeille ei myöskään ole toimivia simulaattoreita ja osa simulaattoreissa on vanhentuneita. Erityyppisille simulaattoreille voidaan johtaa karkea rajapinta verkkoelementtien toiminnan perusteella, kuten käytettävien yhteyksien ja datan muodon suhteen. Simulaattorien tarkempi toiminta on kuitenkin kehittäjätiimin vastuulla ja vaihtelee simulaattorin mukaan.

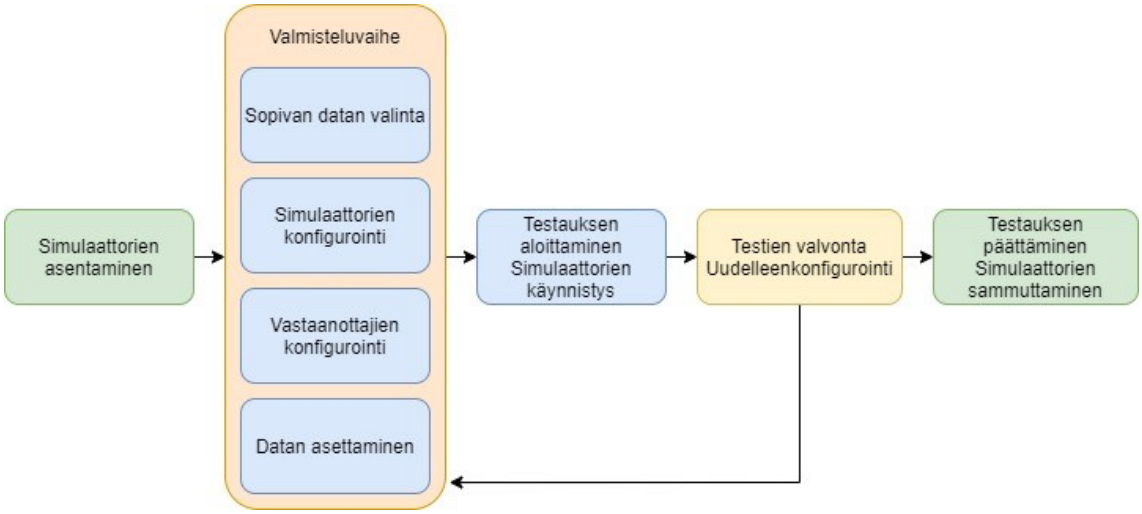
Simulaattorien käyttämä data perustuu määriteltyihin standardeihin ja muotoihin, joita testattava vastaanottaja pystyy käsittelemään. Datan pitää vastata tyypiltään ja määrältään oikean käyttötilanteen dataa, jotta testaus takaisi vastaanottajan oikeanlaisen toiminnan ja suorituskyvyn. Joissain tapauksissa datalle voidaan asettaa tarkempia vaatimuksia, kuten uniikkien verkkoelementin käyttäjien määrä. Data on voitu saada asiakkaalta, kerätty aikaisemmin oikeilta verkkoelementeiltä tai se on voitu generoida erikseen tyhjästä tai olemassa olevaa dataa mukaillen.

Korkealla tasolla eri simulaattorit toimivat samalla tavalla. Simulaattoreille määritellään testissä käytetty data, johon simulaattorit tekevät tarvittaessa muutoksia. Esimerkiksi datassa esiintyvät aikaleimat päivitetään vastaamaan niiden lähetyshetkeä. Simulaattori voi luoda yhteyden vastaanottajaan, tai vastaanottaja voi ottaa yhteyden simulaattoriin riippuen vastaanottajan ja simulaattorin pohjana olleen verkkoelementin tyypistä. Molempiin palveluihin tulee myös määritellä oikeat portti- ja IP-osoitetiedot, jotta yhteys saadaan muodostettua. Yhteyden muodostamisen jälkeen simulaattorit aloittavat datan lähettämisen. Data voi saapua vastaanottajaan tasaisesti tai tietyin väliajoin, ja simulaattorit lähettävät dataa toistuvasti, kunnes ne suljetaan. Simulaattorien ja vastaanottajien toiminta, sekä testiympäristön resurssit voivat rajoittaa datan lähetysnopeutta.

Simulaattorien toiminnassa ja valmistelussa on kuitenkin myös eroja. Simulaattori voi olla ajettavana konttina, jolloin simulaattorin ajo ei vaadi muuta kuin kyseisen kontin asentamisen. Jos simulaattori ei ole ajettavana konttina, sen käyttö voi vaatia useampia erillisiä ohjelmia ja ylimääräistä määrittelyä, esimerkiksi käyttäjätietojen ja kansiorakenteiden kanssa. Simulaattoreiden ohjaus, kuten datan asettaminen voi tapahtua ulkoisen rajapinnan kautta, tai se voi vaatia muutoksia kooditasolla. Simulaattori voi myös vaatia erillisiä komentoja yhteyden luomista ja datan lähettämistä varten. Oletuksena yksi simulaattori yhdistetään yhteen vastaanottajan, mutta yksi vastaanottaja voidaan yhdistää useisiin simulaattoreihin tai yksi simulaattori voidaan yhdistää useaan erityyppiseen vastaanottajaan.

3.2.2 Simulaattorien käyttö testauksessa

Simulaattoreiden pääasiallinen käyttö on pitkäaikaisessa, koko järjestelmän kattavassa testauksessa. Tällöin simulaattoreiden kuorma määritellään testiä vastaavan asiakastilanteen mukaan ja vastaanottajien sekä VAJ:n muiden osien toimintaa tarkkaillaan koko testin ajan. Testissä on normaalisti useita erilaisia vastaanottajatyyppejä ja jokaista vastaanottajaa kohden käytetään vastaavaa simulaattoria. Joissain tilanteissa testi voi vaatia tavallista korkeampaa kuormaa tai tarkoituksena on testata kuorman jakoa VAJ:n sisällä. Tällaisissa tilanteissa simulaattorien ja vastaanottajien määrää voidaan nostaa, erityisesti jos yksittäinen simulaattori ei pysty tuottamaan haluttua testikuormaa.



Kuva 6. *Simulaattorien käyttö testauksessa*

Kuvassa 6 on kuvattu simulaattorien valmistelua ja käyttöä testauksessa. Valmistelu tapahtuu manuaalisesti simulaattori kerrallaan ja valmisteluvaiheet ovat samat simulaattorista tai testistä riippumatta. Kun testiympäristö ja testeissä käytettävät simulaattorit ovat tiedossa, kyseiset simulaattorit asennetaan omaan ympäristöönsä, josta ne voivat lähettää dataa vastaanottajapalveluille. Simulaattoreiden asennusympäristö on VAJ:n ulkopuolella, etteivät simulaattorit veisi VAJ:n resursseja ja datan saapuminen simuloisi paremmin oikean verkkoelementin toimintaa. Simulaattorit on kuitenkin mahdollista asentaa myös samaan ympäristöön VAJ:n kanssa, mutta tällöin täytyy pitää mielessä näiden aiheuttamat muutokset järjestelmän toiminnassa niin resurssien kulutuksen kuin mikropalveluiden määränkin suhteen. Ei-funktionaalisten ominaisuuksien testauksessa simulaattoreita ei asenneta samaan ympäristöön VAJ:n kanssa juuri tästä syystä.

Asentamisen jälkeen simulaattorit tulee konfiguroida oikein ennen testien aloittamista. Kuvan 6 valmisteluvaihe kuvaa yleisimmät konfigurointivaiheet, mutta nämä voivat vaihdella simulaattorin mukaan. Simulaattorille asetetaan testisuunnitelmassa määritelty da-

takuorma, joka on voitu generoida testiä varten tai se valitaan jo olemassa olevista dataseteistä. Jos simulaattorin lähetysnopeutta tai tapaa on mahdollista muokata, myös se suoritetaan valmisteluvaiheessa. Simulaattori voi esimerkiksi vaatia oman komennon, jotta dataa lähetettäisiin toistuvasti. Simulaattorille, ja joissain tapauksissa VAJ:n sisällä olevalle vastaanottajalle, määritellään myös palveluiden välistä yhteyttä varten tarvittavat tiedot.

Konfiguroinnin jälkeen simulaattorit ovat valmiita testien suorittamiseen ja ne käynnistetään. Testin aikana simulaattorien, testattavien vastaanottajien ja testausympäristön tilaa valvotaan ja tarvittaessa simulaattoreihin tehdään muutoksia. Esimerkiksi jos datan lähetysnopeus ei ole halutulla tasolla, vastaanottajassa voi olla jonkinlainen ongelma tai simulaattorin ja vastaanottajan välisessä yhteydessä on ongelmia. Näissä tilanteissa simulaattorin uudelleenkonfigurointi riittää ongelmien korjaamiseen, mutta ajoittain koko simulaattori täytyy asentaa uudestaan ja valmisteluvaihe toistetaan. Kun testit on suoritettu loppuun, simulaattorit suljetaan seuraavaa testikierrosta odottaen. Jos mahdollista, simulaattoreita ei poisteta kokonaan, jotta niiden käyttöönotto on nopeampaa samassa ympäristössä suoritettavia testejä varten.

Koska simulaattorien hallinta tapahtuu manuaalisesti, simulaattorien elinkaari ja sen valvonta aiheuttaa muutamia ongelmia. Testien loputtua datan lähetys pitää lopettaa simulaattori kerrallaan manuaalisesti. Jos testeissä käytetään useita simulaattoreita, yksittäinen simulaattori voi jäädä vahingossa päälle testien loputtua, jolloin se kuluttaa ylimääräisiä resursseja ja aiheuttaa odottamatonta kuormaa VAJ:lle. Pahimmassa tapauksessa ylimääräinen simulaattori voi pilata testin, jos ylimääräinen datakuorma aiheuttaa virhetilanteita tai resurssien puutteita. Vastaavasti jos simulaattori lopettaa toimintansa kesken testin, tästä ei tule automaatista ilmoitusta eikä simulaattori pysty jatkamaan datan lähettämistä. Testikuorman laskeminen huomataan järjestelmän tarkkailuohjelmiston kautta, mutta tämä voi tapahtua paljon simulaattorin sammumisen jälkeen ja sen uudelleenkäynnistämiseen kuluu aikaa.

Simulaattorit luovat myös lokeja testien aikana, esimerkiksi lähetetyn datan määrästä tai testattavan vastaanottajapalvelun sijainnista verkon sisällä. Myös virheet kirjataan erilliseen lokiin. Näitä lokeja ei kuitenkaan kerätä automaattisesti testien aikana, ja jos simulaattori ajetaan konttina, lokit katoavat kontin sammumisen jälkeen. Testattavista vastaanottajapalveluista kerätään kattavasti metriikkaa, joten simulaattorien tarkkailu on epäsuorasti mahdollista, mutta hyvin karkeaa. Vastaanottaja voi esimerkiksi näyttää datan saapumisen keskeytyneen, jolloin syy voi olla simulaattorissa, vastaanottajassa, näiden välisissä yhteyksissä tai testausympäristössä.

3.3 Testauksen ja simulaattorien ongelmat

Simulaattorit toimivat hyvin palvelutestauksessa. Koska datan määrällä ei ole suurta merkitystä testin tulosten suhteen, testaaja voi vain valita jonkin valmiina olevan datasetin testiinsä. Yhtä simulaattori käytettäessä sen manuaalinen asentaminen ja konfigurointi on suhteellisen nopeaa, ja usein testit tapahtuvat samassa ympäristössä, jolloin simulaattorina ei tarvitse asentaa uudestaan testien välillä

Isommissa E2E- ja ei-toiminnallisissa testeissä simulaattorien käyttöön liittyy kuitenkin ongelmia. Testauksessa käytettävät ympäristöt vaihtelevat ja testit kattavat useita eri vastaanottajia. Tämän seurauksena uusia testejä varten simulaattorit tulee konfiguroida tai jopa asentaa uudestaan ennen kuin testaus voidaan aloittaa. Jos testausympäristö on tavallisesta poikkeava, tämä voi viedä huomattavasti enemmän aikaa kuin tuttuun ympäristöön asentaminen. Suuren skaalan testeissä voidaan myös tarvita kymmeniä eri simulaattoreita, joista jokainen tulee asentaa ja konfiguroida erikseen. Kaikki tämä hidastaa testien aloittamista ja vaatii valmistelua testausta suorittavalta tiimiltä, kun pelkästään simulaattoreiden valmistelu vie pahimmassa tapauksessa useita työpäiviä. Testaukseen käytettävä aika on usein rajallista [15] ja tämä testausohjelmiston asentamisesta johtuva viivästys voi tarkoittaa lyhyempää testiä.

Simulaattoreiden käyttötavat vaihtelevat huomattavasti ja simulaattorien dokumentaatio on hyvin hajautettua. Ennen tätä diplomityötä suoritetussa simulaattorien arvioinnissa huomattiin, että kaikilla simulaattoreilla ei edes ole selkeitä käyttöohjeita ja ne luotiin vasta arvioinnin yhteydessä. Tämä puute on vaikeuttanut simulaattorien uusien käyttäjien työtä, erityisesti virhetilanteiden ilmetessä. Simulaattorit eivät myöskään jaa tai lähetä sisäisiä virhelokejaan mihinkään ulkoiseen palveluun, mikä vaikeuttaa simulaattoreiden terveyden seurantaa ja virhetilanteiden paikantamista [1,4]. Yhtenäisten rajapintojen ja komentojen avulla simulaattorien käyttö olisi helpompi oppia ja niiden konfigurointi olisi nopeampaa, ja keräämällä simulaattoreiden lokit yhteiseen paikkaan simulaattoreiden statusta ja virhetilanteita olisi helpompi valvoa [4,9].

Testeissä käytettävä dataan liittyy myös ongelmia. Ajantasaisen ja realistisen testidatan saaminen asiakkaalta tai oikeilta verkkoelementeiltä keräämällä ei aina ole mahdollista. Parhaimmassakin tapauksessa datan hankinta vie aikaa ja rahaa, ja lopputuloksena voi olla liian pieni tai suuri määrä dataa, joka soveltuu näin ollen vain tietyntyyppisiin testeihin. Liian suuret datamäärät kuormittavat järjestelmää liikaa ja vievät resursseja muilta simulaattoreilta, kun taas liian pienellä datamäärällä ei saada oikeaa kuvaa järjestelmän ja vastaanottajien toiminnasta suuren kuorman alla.

Jo olemassa oleva data voi olla vanhentunutta, salattua tai peräisin tuntemattomista lähteistä, jolloin sen sisällöstä tai uniikkien kenttien määrästä ei ole tarkkaa tietoa. Datasettien koossa ei myöskään ole tarpeeksi vaihtelua, jotta niitä olisi mahdollista käyttää kaikissa testeissä. Esimerkiksi yksi simulaattori sisältää valmiina kolmen kokoisia datasettejä, joista pienin on liian pieni suoraan suorituskykytestaukseen, mutta sitä suuremmat ovat liian suuria ja kuormittavat VAJ:tä liikaa. Joissain testitilanteissa datan tulisi sisältää juuri oikea määrä uniikkeja datakenttiä, esimerkiksi lähetetyn datan pitää sisältää 20 000 eri käyttäjän dataa. Koska data on kerätty erilaisista lähteistä eri tavoin, eri simulaattorien lähettämä data ei ole yhtenäistä esimerkiksi sijaintitietojen suhteen, eikä näin ollen vastaa oikeiden verkkoelementtien lähettämää dataa. Näitten datapuutteitten takia testauksessa on jouduttu tekemään useita kompromisseja, jotta niistä olisi mahdollista saada jonkinlaisia vertailukelpoisia tuloksia.

Asiakkaiden käyttämiin oikeisiin verkkoelementteihin tulee myös muutoksia useita kertoja vuodessa. Nämä muutokset voivat liittyä joko verkkoelementin toimintaan tai sen tuottamaan dataan. Joka tapauksessa tällaisen muutoksen tapahtuessa sekä vastaanottajaan että simulaattoriin täytyy tehdä vastaavia muutoksia. Erityisesti datassa tapahtuvat muutokset vaativat uuden datasetin hankkimista ja mahdollisesti simulaattorin toiminnan päivittämistä, jotta uusia vastaanottajia voidaan testata oikein.

Näin ollen vaikka suurempien testien suorittaminen nykyisillä simulaattoreilla ja järjestelmillä on mahdollista, saadut tulokset eivät aina ole käyttökelpoisia, tai niissä on kompromisseista seuraavia puutteita. Jotta laajempien, tarkempien ja keskenään vertailukelpoisten testien suorittaminen olisi mahdollista, tässä luvussa käsiteltyihin ongelmiin ja esteisiin pitää kehittää ratkaisuja.

4. HALLINTAJÄRJESTELMÄN SUUNNITTELU

Simulaattorien manuaaliseen asentamiseen ja konfigurointiin liittyvät ongelmat vaikeuttavat pitkien testien suorittamista. Väärin asennettu tai määritelty simulaattori ei luonnollisesti toimi halutulla tavalla ja voi aiheuttaa virhetilanteita jo ennen oikeiden testien aloittamista. Testien skaalan kasvaessa tarvitaan useita simulaattoreita ja sopivan testikuorman löytäminen tai luominen vaikeutuu. Useiden simulaattoreiden hallinta ja valvonta on myös vaikea suorittaa ilman erillisiä hallintajärjestelmiä.

Tämän diplomityön tarkoituksena on kehittää hallintajärjestelmä, jonka avulla simulaattorien asentaminen ja konfigurointi saataisiin automatisoitua ja jonka avulla suuren simulaattorimäärä hallinta ja valvonta olisi mahdollista. Lopputuloksen tulisi olla samanaikaisesti toimiva hallintajärjestelmä, jota voi käyttää sellaisenaan, että teoreettinen pohja kehittyneemmälle ja laajemmalle hallintajärjestelmälle.

Tässä luvussa määritellään hallintajärjestelmän vaatimukset ja ominaisuudet, sekä sen kehittämisessä käytettyjä tekniikoita. Luvussa määritellään myös yhteinen toiminnallisuus uusia simulaattoreita varten sekä kuvaillaan testidatan luomisessa käytettävän datageneraattorin toiminta.

4.1 Hallintajärjestelmän tavoitteet

Aikaisemmissa luvuissa kuvailtujen mikropalveluiden ominaisuuksien ja simulaattorien käyttöön liittyvien ongelmien perusteella hallintajärjestelmälle voidaan asettaa erilaisia vaatimuksia ja tavoitteita. VAJ:n yhteydessä käytettävät tekniikat ja ohjelmistot vaikuttava myös hallintajärjestelmän suunnitteluun, koska jo käytössä olevien toiminnallisuuksien uudelleenkäyttö olisi suotavaa.

Hallintajärjestelmän toiminnalliset tavoitteet voidaan jakaa kolmeen eri osa-alueeseen, jotka voidaan jakaa vielä pienempiin osiin:

1. Simulaattorit
    a. Simulaattorien asentaminen
    b. Simulaattorien konfigurointi
    c. Simulaattorien hallinta
2. Monitorointi
    a. Lokien luominen
    b. Lokien kerääminen ja yhdistäminen
    c. Järjestelmän tilan tarkkailu

3. Testidata
    a. Testidatan luominen
    b. Simuloitavan verkon topologian luominen.

Hallintajärjestelmän kautta pitää pystyä asentamaan ja konfiguroimaan testin vaatimat simulaattorit automaattisesti. Konfigurointi kattaa tässä tilanteessa sekä testidatan asettamisen että muiden tarvittavien tietojen määrittelyn. Simulaattoreiden käynnistämisen sekä sammuttamisen pitää myös tapahtua keskitetysti hallintajärjestelmän kautta. Jotta simulaattorien hallinta olisi mahdollista toteuttaa, simulaattoreille pitää suunnitella yhteinen rajapinta. Tämä rajapinta käydään läpi aliluvussa 4.4. Hallintajärjestelmä sisältää myös käyttöliittymän simulaattorien ohjaamista varten, jolloin käyttäjän ei tarvitse tuntea simulaattorien tarkempaa toimintaa.

Monitorointi kattaa simulaattoreiden muodostaman testausjärjestelmän tietojen tallentamisen sekä keräämisen. Sekä hallintajärjestelmän että simulaattorien pitää luoda lokeja, joiden tietojen tulee olla yhdistettävissä. Järjestelmän tilaa pitää myös valvoa, jotta mahdolliset virhetilanteet tai simulaattoriongelmat huomattaisiin nopeasti. Tämänkaltaisella monitoroinnilla virheiden sijainti on helpompi löytää riippumatta käytettyjen simulaattorien määrästä ja simulaattoreiden virhetilanteet voidaan korjata nopeammin [4].

Sopivan testidatan luomista varten on kehitetty erillinen generaattori-mikropalvelu. Hallintajärjestelmän tulisi käyttää tätä generaattoria halutunlaisen testidatan luomiseen. Generaattorin avulla on myös mahdollista luoda simulaattorien käyttämiä konfiguraatiotietoja sekä simulaattorien verkkotopologia. Generaattorin toiminta käydään läpi korkealla tasolla aliluvussa 4.5.

Testausta suoritetaan monissa erilaisissa ympäristöissä, joten hallintajärjestelmän ja sen osien tulee myös toimia eri ympäristöissä ilman että järjestelmään tai ympäristöön tarvitsee tehdä muutoksia. Helppo ja nopea asennus on pakollista, jotta testauksen valmisteluun kuluva aika saataisiin minimoitua. Hallintajärjestelmän käyttäjän pitäisi pystyä suorittamaan sivulla 16 olevan kuvan 6 mukaiset vaiheet helposti ja nopeasti. Kaiken simulaattorien valmisteluun ja käyttöön liittyvää toimintaa pitää olla mahdollista ohjata hallintajärjestelmän kautta.

Vaatimuksilla on erilaisia prioriteetteja ja hallintajärjestelmän eri osia kehitetään samaan aikaan. Tärkeimpänä on toteuttaa simulaattorien hallinta, jotta uusia ja vanhoja simulaattoreita voidaan testata. Tällöin hallintajärjestelmä voidaan ottaa käyttöön ei-funktionaalisia testejä suoritettaessa. Myös datan generointi ja kyseisen datan asettaminen simulaattoreille on tärkeää hallintajärjestelmän käyttöönotolle. Vaikka monitorointi on tärkeä

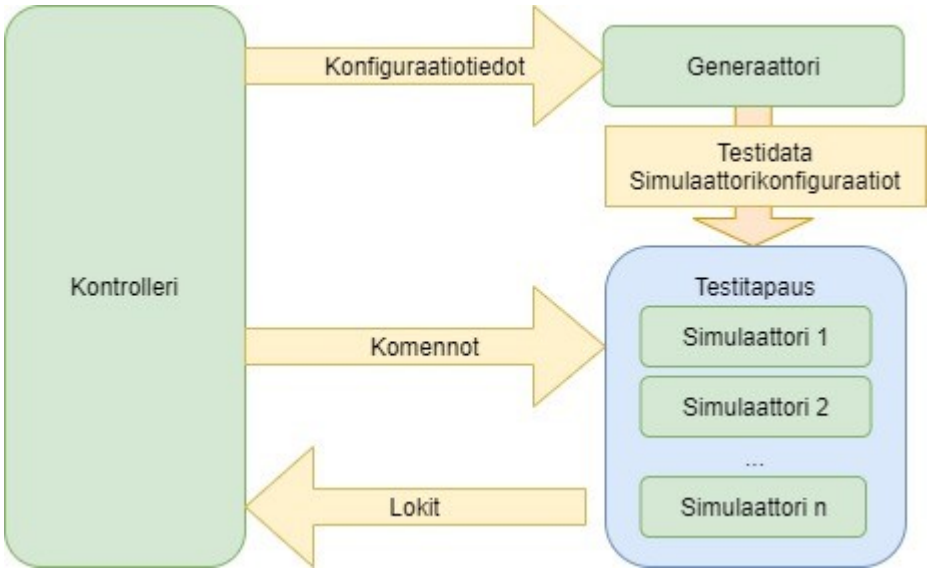
osa hallintajärjestelmää, nämä ominaisuudet voidaan toteuttaa vasta, kun simulaattorien hallinta toimii halutulla tasolla.

Hallintajärjestelmälle on myös suunniteltu mahdollisia lisäominaisuuksia, kuten monitoroinnin laajentamista ja useiden palvelinten hyötykäyttöä. Hallintajärjestelmän toteutusta esitellään myös testauksesta vastaavalle tiimille, jonka antama palaute otetaan huomioon nykyisissä ja tulevissa vaatimuksissa. Kaikki järjestelmän vaatimukset, ominaisuudet ja mahdolliset käyttöohjeet dokumentoidaan sisäisesti.

4.2 Arkkitehtuuri

Hallintajärjestelmä toteutetaan mikropalveluina. Koska järjestelmää tullaan käyttämään erilaisissa testeissä, joissa käytettyjen simulaattorien tyyppi ja määrä voivat vaihdella paljon, hallintajärjestelmä voi käyttää hyväksi mikropalveluiden helppoa skaalaamista. Simulaattorien ajoittainen päivittäminen ei myöskään vaadi muutoksia koko järjestelmään, vaan vain kyseisen mikropalvelun vaihtaminen riittää. Mikropalveluiden kautta järjestelmän osat voidaan toteuttaa helposti eri tekniikoilla ja erityisesti simulaattoreiden toteutuksessa voi olla verkkoelementeistä johdettavia tekniikkavaatimuksia, kuten lähetettävän datan salaustapa [2,4,6].

Järjestelmän keskuksena toimii muita mikropalveluita ohjaava kontrolleri. Kontrolleri toimii testaajan käyttöliittymänä järjestelmän muiden osien, simulaattorien ja generaattorin, ohjaamiseen. Järjestelmän tilan valvonta ja lokitietojen kerääminen on myös kontrollerin vastuulla. Hallintajärjestelmän valmistumista varten kontrollerista pitää toteuttaa prototyyppi, ja tämän suunnittelu ja kehittäminen on diplomityön keskiössä.



Kuva 7. *Järjestelmän toiminta korkealla tasolla*

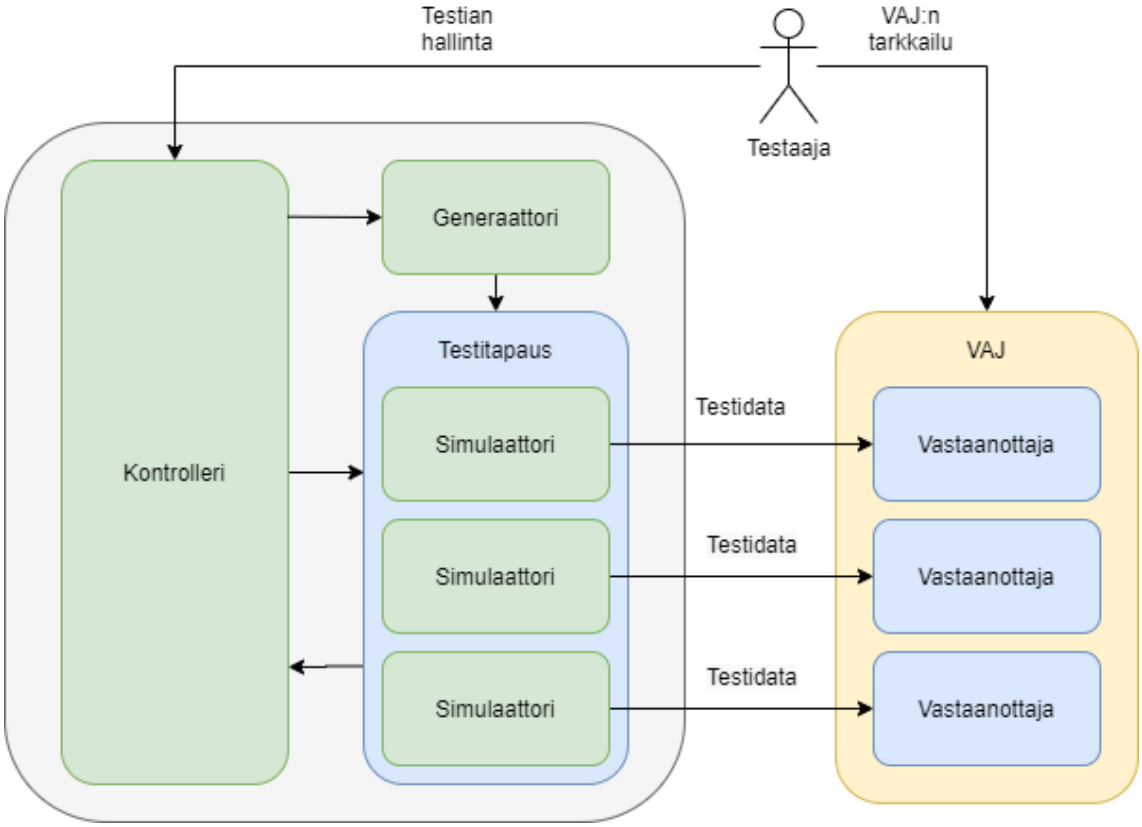
Kuvassa 7 on esitelty testausjärjestelmän toiminta korkealla tasolla. Kontrolleri vastaa tarvittavien konfiguraatiotietojen antamisesta oikeille konteille, konttien tilan valvonnasta sekä simulaattorien luomien lokien keräämisestä. Kontrollointi tapahtuu erilaisten testitapausten kautta: Testitapaus sisältää joukon simulaattoreita, simulaattoreiden konfiguraatiotiedot sekä simulaattorien käyttämän datan. Näiden tietojen avulla kontrolleri voi käynnistää testitapaukseen kuuluvat simulaattorit ja asettaa generoidun datan oikealle simulaattorille. Tällä tavalla samaan testitapaukseen kuuluvat simulaattorit voidaan käynnistää yhdellä komennolla. Koska simulaattorikontit on sidottu testitapaukseen, lokien aggregointi ja järjestely on helppoa kyseisen testitapauksen tunnuksen kautta. Testitapaukset luodaan generaattorin avulla ja kontrolleri voi ajaa useita testitapauksia samanaikaisesti.

Kuvan nuolet näyttävät käskyjen ja tietojen siirtosuuntaa järjestelmän palveluiden välillä. Kontrolleri käynnistää generaattorikontit erillisellä komennolla, jossa generaattorille annetaan halutut konfiguraatiotiedot sekä testitapauksessa käytettyjen simulaattoreiden topologia. Näiden tietojen perusteella generaattori luo yksittäisen testitapauksen ja tallentaa sen järjestelmän palvelimelle. Tämän jälkeen kontrolleri voi käynnistää testitapaukseen kuuluvat simulaattorit ja asettaa niille oikeat tiedot. Kun testit on suoritettu loppuun, kontrollerilla voidaan sulkea kyseinen testitapaus, jolloin simulaattorien keräämät lokit kerätään ja yhdistetään kontrollerin luomiin lokeihin.

Hallintajärjestelmän ajoympäristö vaihtelee ja käytössä olevista resursseista ei ole takeita. Ympäristö on kuitenkin Linux-pohjainen palvelin, jolle on mahdollista ottaa etäyhteys. Koska kontrolleri toimii järjestelmän käyttöliittymänä ja yhteys ajoympäristöön tapahtuu komentorivin kautta, kontrollerille ei tarvitse suunnitella graafista käyttöliittymää

kehityksen alkuvaiheessa. Hallintajärjestelmän käyttämät mikropalvelut ja mahdolliset tallennustilat eivät oletuksena kuulu ajoympäristöön, joten näiden luominen tai noutaminen kuuluu myös hallintajärjestelmän vastuulle.

Testidatan koko vaihtelee testien ja simulaattorien välillä, mutta simulaattorien määrän kasvaessa pienenkin datan määrä kasvaa moninkertaiseksi. Testidatan ja konfiguraatiotietojen asettamisen tulisi olla mahdollisimman kevyt ja yksinkertainen operaatio sekä ajan- että resurssienkäytön optimoimiseksi. Järjestelmän sisäisen kommunikaation pitäisikin olla mahdollisimman kevyttä, jotta isäntäpalvelimen resurssit saataisiin ohjattua simulaattoreiden ajamiseen. Simulaattorit tarvitsevat myös jonkin paikan, minne ne voivat tallentaa lokitietoja. Tämän paikan elinkaaren pitää olla vähintään vastaavan simulaattorin elinkaaren pituinen, jotta lokitiedot eivät katoa simulaattorikontin sulkemisen jälkeen. Myös kontrollerin luomien ja keräämien lokien tiedoille tarvitaan tallennuspaikka ja kansiorakenne.



Kuva 8. *Järjestelmä yhdessä VAJ:n kanssa*

Kuvassa 8 näytetään hallintajärjestelmän ja VAJ:n suhde vastaanottajien testauksen aikana. Todellisuudessa VAJ koostuu useammasta mikropalvelusta, mutta testausjärjestelmän kannalta pelkästään vastaanottajilla on merkitystä. Sekä testausjärjestelmä että VAJ ovat eri ympäristöissä, joidenka välille on mahdollista luoda yhteys vastaanottajien kanssa kommunikointia varten.

4.3 Käytettävät tekniikat

Hallintajärjestelmä yhdistää monia mikropalvelutestauksen osa-alueita, kuten testauksessa käytettyjen simulaattorien hallintaa sekä niiden tuottamien lokien keräämistä ja yhdistämistä. Kaiken halutun toiminnallisuuden toteuttaminen vaatii useiden eri tekniikoiden yhdistämistä. Tässä aliluvussa käsitellään valittuja tekniikoita ja perusteluja niiden valitsemiselle.

Kontrollerin toteutukselle ei ole suoria teknisiä vaatimuksia, mutta mitä vähemmän se kuluttaa resursseja sitä enemmän niitä riittää järjestelmän muiden osien käyttöön. Koska diplomityön tavoitteena on kehittää järjestelmän prototyyppi ja saada se testauskäyttöön mahdollisimman nopeasti, kontrollerin toteutuskieleksi on valittu Python 3.8. Kieli on entuudestaan tuttu, sillä on yksinkertainen syntaksi, laajoja lisäkirjastoja ja sitä on mahdollista käyttää erilaisilla ympäristöillä ja käyttöjärjestelmillä [25,26]. Kontrollerille ei luoda erillistä käyttöliittymää vaan sen käyttö tapahtuu isäntäpalvelimen komentorivin avulla.

Nykyisten simulaattoreiden hallinta on vaikeaa vaihtelevien käyttötapojen ja toiminnallisuuksien takia. Kontrollerin pitää pystyä hallitsemaan simulaattoreita yhteisellä logiikalla ja toiminnalla. Tämä vaatii yhtenäisen rajapinnan ja toiminnallisuuden määrittelyä, joka käsitellään tarkemmin aliluvussa 4.4. Simulaattorien hallinnan ja asentamisen helpottamiseksi niitä tullaan käyttämään aliluvussa 2.3 kuvailtuina kontteina.

Konttiteknologiaksi valittiin Docker [22], joka tarjoaa monia valmiita ominaisuuksia konttien hallintaan ja tilanvalvontaan liittyen. Docker-kontit perustuvat kuviin (engl. image) jotka sisältävät kontissa ajettavan mikropalvelun tiedot. Kuvia voidaan säilyttää erillisessä varastossa (engl. repository) josta ne voidaan noutaa Docker-komennolla tarpeen vaatiessa. Noutamisen jälkeen kuva tallennetaan isäntäpalvelimelle, josta kyseinen kontti voidaan käynnistää.

Kontteja on mahdollista kustomoida käynnistämisen yhteydessä. Kontille voidaan määritellä sisäiset ja ulkoiset portit, metadataa erillisen tunnuksen (engl. label) avulla ja jopa yhdistää isäntäpalvelimella olevia kansioita kontin sisälle, jolloin kontti voi käsitellä kansion sisältämiä tiedostoja ja kansioihin tallennetut tiedot säilyvät kontin sammuttamisenkin jälkeen. Näiden ominaisuuksien avulla simulaattorikonttien konfigurointi voidaan suorittaa suurimmaksi osaksi kooditasolla, kun konttia käynnistetään. Erillisiä verkkokutsuja datan asettamiseksi ei siis tarvita [22,23,27,28].

Dockeria ja sen komentoja voidaan käyttää suoraan kooditasolla Pythonille suunnitellun Docker SDK-kirjaston avulla [29]. Kirjasto sisältää Docker-komentoja vastaavia funktioita ja kontteihin liittyviä tietoja on helppo käsitellä ja tarkastella erilaisten tietorakenteiden

avulla. Esimerkiksi konteille asetetut nimet ja tunnukset on helppo noutaa kontin käynnistyksen yhteydessä. Kirjastoa käyttämällä kontteja on mahdollista käynnistää, tarkastella ja sammuttaa suoraan kooditasolla ja kontrollerin ei tarvitse tuottaa erillisiä komentorivi käskyjä.

Hallintajärjestelmän sisäinen kommunikointi tapahtuu REST-verkkokutsujen avulla. REST on kevyt ja se sisältää valmiita metodeja, joita voidaan käyttää hyväksi eri komennoissa. Simulaattoreille määritellään rajapinta, jota kontrolleri voi käyttää simulaattorien hallintaa varten. Tähän rajapintaan voidaan tarvittaessa lisätä komentoja, jos jo olemassa olevissa komennoissa havaitaan puutteita [30].

Sekä kontrollerin että simulaattorien luovat lokeja JSON-muodossa [31]. Lokien kirjoittamisessa hyödynnetään valmista Log4J-moduulia [32,33] joka on myös vastaanottajien käytössä. Tämä auttaa lokien yhtenäistämisellä ja erilaisten virhetilanteiden löytäminen VAJ:n ja hallintajärjestelmän välillä on helpompaa. Tarvittaessa lokeille kehitetään oma JSON-rakenne, jotta kaikki monitoroitavat tiedot saadaan katettua. Kontrolleri käyttää Pythonin JSON-kirjastoa lokien lukemista, kirjaamista ja kokoamista varten.

Koko testausjärjestelmän tulee toimia erilaisilla Unix-pohjaisilla järjestelmillä, kuten Linux tai CentOS. Docker-konttien ansiosta osa järjestelmästä on teknologia-agnostinen ja kontrolleri kehitetään Linux Ubuntu -virtuaalikoneella ja sitä testataan myös Linux-pohjaisella palvelimella, joka muistuttaa oikeaa VAJ:n testiympäristöjä.

4.4 Simulaattorien rajapinta ja hallinta

Nykyisten simulaattorien käyttötavat ja rajapinnat ovat liian erilaisia, että niitä olisi helppo ohjata hallintajärjestelmän avulla. Simulaattoreille pitää määritellä yhteinen rajapinta ja korkean tason toiminta, jotta hallintajärjestelmän ja erityisesti kontrollerin toiminnasta voidaan tehdä yksinkertaista. Rajapintojen määrittelyn ansiosta hallintajärjestelmällä voidaan käyttää tulevaisuudessa kehitettäviä simulaattoreita ja näiden simulaattoreiden kehittäminen on yksinkertaisempaa, kun osa toiminnasta voidaan uudelleen käyttää jo olemassa olevista simulaattoreista. Yhteinen rajapinta ja toiminnallisuus tarkoittaa myös muutoksia vanhoihin simulaattoreihin. Tämä simulaattorien toiminnan yhtenäistäminen yksinkertaistaa simulaattorien käyttöä myös automatisoidun järjestelmän ulkopuolella, kun simulaattorien käyttötapa on sama simulaattorista riippumatta. Kaikkien simulaattoreiden tulee täyttää tässä aliluvussa mainitut ominaisuudet.

Simulaattorit tulee toteuttaa Docker-kontteina aikaisemmassa aliluvussa 4.3 kuvailluista syistä. Tällä tavalla simulaattorien toiminnallisuus voidaan toteuttaa halutuilla tekniikoilla, mutta luodun kuvan koon tulisi olla mahdollisimman pieni, jotta sen ajaminen ei veisi

ylimääräisiä resursseja. Docker-kuvat muodostuvat kerroksista (engl. layer) jotka sisältävät tiedostoja ja metatietoja kontin luomista varten. Pitämällä näiden kerrosten koon ja määrän mahdollisimman pienenä myös kontin koko pysyy sopivana. Esimerkiksi ohjelmointikieli voi vaikuttaa kerrosten kokoon [22].

Simulaattoreille pitää olla mahdollista antaa testissä käytettävä testidata sekä konfiguraatiotiedot. Tiedot annetaan simulaattorikontin käynnistymisen yhteydessä, jonka jälkeen simulaattorin tulee lukea nämä tiedot. Tietojen muoto ja erityisesti konfiguraatiotietojen sisältö on simulaattorin toteuttavan tiimin päätettävissä, mutta kaiken vaihtuvan tiedon, kuten vastaanottajan IP-osoitteen, tulisi olla annettavissa tällä tavoin. Näin simulaattoreille ei tarvitse tehdä kooditason muutoksia eri ympäristöissä tai testeissä. Simulaattorikonttien monitorointia varten simulaattorien tulee myös luoda erilaisia lokeja ja näiden lokien tulee olla kontrollerin kerättävissä. Jos testitapaukseen kuuluu useita simulaattoreita lokien siirtely voi kuitenkin viedä liika resursseja ja aiheuttaa viivettä hallintajärjestelmän toiminnassa.

Tietojen antaminen ja lokien kerääminen on mahdollista toteuttaa REST-kutsuilla, mutta jo simulaattoreiden määrän ja lokien koon kasvaessa jatkuvat verkkokutsut voivat hidastaa koko hallintajärjestelmän toimintaa. Dockerissa on olemassa volume-ominaisuus [22,23] jolla kontin isäntäpalvelimella oleva kansio voidaan linkittää kontin sisällä olevaan kansioon. Tällöin kontti voi käyttää kyseisen kansion tietoja omassa toiminnassaan ja kontin sulkemisen jälkeen kansioon tehdyt muutokset säilyvät isäntäpalvelimella. Käyttämällä tätä ominaisuutta simulaattoreille voidaan linkittää testidatan sisältävät kansio, josta simulaattori voi lukea tarvittavat tiedot ja tallentaa luomansa lokit saman kansion alle. Tällä tavalla lokit säilyvät simulaattorikontin elinkaaren yli ja yllättävät sammumiset eivät tuhoa jo luotuja lokeja.

Simulaattoreissa tulee olla myös REST-rajapinta, jonka avulla simulaattoria voidaan komentaa esimerkiksi toistamaan testauksessa käytetyn datasetin lähettäminen. Rajapinnan komennoissa käytettävänä protokollana on HTTP, rajapinnan polkuna rajapinnan versio (esim. v1 ensimmäiselle versiolle) ja oletusporttina 8080. Taulukko 1 näyttää toteutettavat komennot sekä niiden parametrit selityksineen.

Taulukko 1. *Simulaattorien REST-komennot parametreineen*

Komento	Parametri	Tyyppi	Oletusarvo	Kuvaus
POST /start	repeat	boolean	false	Toistetaanko annettu datasetti
	startTime	String	nykyinen aika	ISO aika (muodossa ”yyyy-MM-dd’T’HH:mm:ss.SSS’Z”")
	logFrequency	int	1000	Tietuelokin kirjoittamisen intervalli millisekunteina
POST /stop	-	-	-	-

Esimerkiksi komento

```
POST iposoite:8080/v1/start {repeat: true}
```

saisi kyseisen simulaattorin aloittamaan datan lähettämisen toisto-ominaisuuden kanssa. Saman simulaattorin lähettäminen voidaan lopettaa komennolla

```
POST iposoite:8080/v1/stop
```

Rajapinnan ollessa valmis komentoja voidaan tarvittaessa määritellä lisää, jos simulaattorien tarkemmalle hallinnalle on tarvetta.

Vanhat simulaattorit luovat jo valmiiksi erilaisia lokeja, mutta niitä ei aina tallenneta ja ne katoavat kontin sulkemisen jälkeen. Jotta testien myöhempi analysointi olisi mahdollista, lokien tulee säilyä myös simulaattorikontin elinkaaren ulkopuolelle. Tässä analysoinnissa on myös tarpeen tietää simulaattorin lähettämän datan määrä, jota vanhat simulaattorit eivät lokita erikseen. Uusien simulaattorien tuleekin käyttää aikaisemmin mainittua linkitettyä kansiota lokien tallennuspaikkana, ja luoda sinne kolmentyyppisiä lokeja. Nämä lokit ovat: virheloki (engl. errorLog), tilaloki (engl. allLog) ja tietueloki (engl. recordLog). Virhelokiin kirjataan kaikki simulaattorin virhetilanteet, tilalokiin mahdolliset tilamuutokset ja yhteydet, ja tietuelokiin simulaattorin lähettämien tietueiden määrä ja intervalli. Virhe- ja tilaloki käyttävät Log4J-lokitustapaa [32], joka on yhtenäinen testattavien vastaanottajien lokitustavan kanssa.

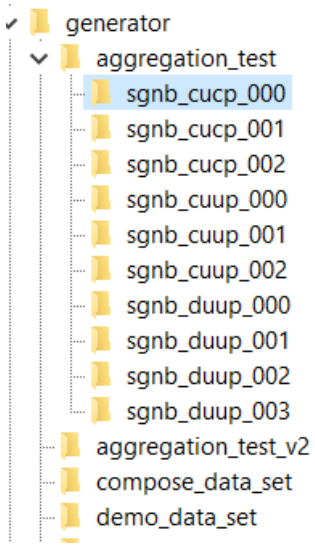
4.5 Datageneraattori

Kontrollerin ja simulaattorien lisäksi järjestelmään kuuluu testidatan generaattori. Generaattorin vastuulla on simulaattorien käyttämän datan ja konfigurointitiedostojen luonti

annetun verkkotopologian perusteella. Generaattori luo myös testitapauksen käynnistämisessä käytettävän topologiatiedoston, joka sisältää käytettävät simulaattorit, simulaattorikonttien tiedot ja konteille annettavan datan.

Generaattori koostuu kahdesta kokonaisuudesta: generaattorin yleinen osa ja elementtikohtaiset osat. Yleinen osa on koko generaattorin käyttöliittymä, joka hallitsee elementtikohtaisia generaattoreita sekä kokoaa niiden tuottaman datan. Elementtikohtaiset generaattorit generoivat kyseisen verkkoelementin käyttämän datan ja konfiguraatiotiedot. Elementtikohtaisten generaattoreiden toteuttaminen on kyseisen verkkoelementin vastaanottajan kehittäneen tiimin vastuulla, jolloin he voivat tarkemmin määritellä simulaattoriin sopivan datan ja muut tarpeelliset tiedot. Koko generaattori ajetaan omana Docker-konttinaan, joka sulkee itsensä testitapauksen luonnin jälkeen.

Testidata generoidaan kontrollerin antamien konfiguraatiotiedostojen perusteella. Näissä tiedostoissa määritellään testiin kuuluvat simulaattorit, testidatan muoto ja määrä, sekä mahdolliset lisäominaisuudet, kuten koko testitapauksen kesto. Näiden tietojen perusteella generaattorin yleinen osa ohjaa oikeat elementtikohtaiset generaattorit generoimaan datan, joka tallennetaan isäntäpalvelimelle erillisiin kansioihin. Kun kontrolleri käynnistää testitapauksen simulaattorit, nämä kansiot linkitetään konttien sisältämiin kansioihin volume-ominaisuuden [34] avulla, jolloin simulaattorit voivat lukea tarvittavan tiedon sieltä.



Kuva 9. *Generaattorin luomien testitapausten kansiorakenne*

Generaattori luo kuvan 9 mukaisen kansiorakenteen. Jokaiselle testitapaukselle luodaan oma kansio, jossa on simulaattorien konfigurointitiedot sisältävän imageMap.json tiedoston sekä simulaattorikohtaisen testidatan sisältävän kansion. Samaa kansiota käytetään simulaattorin luomien lokien tallennuspaikkana. Esimerkiksi kuvan 9 aggregation_test

niminen testitapaus koostuu kymmenestä erillisestä simulaattorista, joista jokaisella on oma simulaattorikontille asetettava kansio, kuten sgnb_cucp_000.

5. AUTOMATISOIDUN TESTAUSJÄRJESTELMÄN TOIMINTA

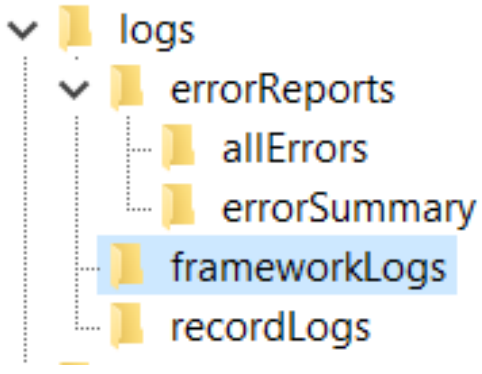
Tässä luvussa käsittelemme tarkemmin testausjärjestelmän toimintaa ja toteutusta. Käymme läpi järjestelmän eri komennot sekä niiden toiminnan. Kontrollerin toteutus ja käyttö käydään läpi tarkasti, kun taas generaattorista ja simulaattoreista käymme läpi niiden ulkoiset rajapinnat ja kuinka kontrollerin toiminta vaikuttavat niihin.

Järjestelmä kehitettiin Linux-virtuaalikoneella ja sen toiminta on varmistettu erillisellä palvelimella. Järjestelmää on testattu oikeilla testitapauksilla, generaattorilla ja simulaattoreilla. Hallintajärjestelmän kehitystä ja toimintaa esiteltiin ei-funktionaalisista testeistä vastaavalle tiimille ja arkkitehdille. Heiltä saatu palaute on otettu huomioon kehityksessä, jotta kaikki tarpeelliset toiminnallisuudet ja tiedot saadaan katettua.

5.1 Kontrolleri

Kontrolleri voidaan ajaa Pythonia, Dockeria ja Docker SDK -kirjastoa tukevassa ympäristössä. Kontrolleri on toteutettu Python-ohjelmana, jonka käynnistämisen jälkeen sillä voidaan suorittaa erilaisia käskyjä testitapausten luomista ja hallintaa varten. Kontrolleri suorittaa myös taustatoimintoja järjestelmän terveyden ja tilan valvontaa varten.

Kontrolleri vaatii toimiakseen tietynlaisen kansiorakenteen erilaisten lokien ja testitapausten tallentamista varten. Testitapausten kansiorakenne on kuvattu aliluvussa 4.5. Kontrollerin keräämät ja luomat lokit asetetaan omaan logs-kansioon ja sen alikansioihin.



Kuva 10. *Kontrollerin lokikansioiden rakenne*

Kuvassa 10 on kontrollerin käyttämän lokikansion rakenne. ErrorReports-kansio sisältää kaksi alikansiota erityyppisiä virhelokeja varten. AllErrors-kansioon tallennetaan suu-

rempi virheloki, johon on yhdistetty kaikkien simulaattoreiden ja kontrollerin virhelokit karsimattomina. ErrorSummary-kansioon tallennetaan tiivistetty versio tästä lokista, joka sisältää aggregoidun tiedon virheiden määrästä ja vakavuudesta niin koko järjestelmän tasolla kuin myös testitapauskohtaisesti. Molemmat lokityypit tallennetaan lokin luomishetken mukaisesti, muodossa YYYY-DD-MM;HH:mm:SS.

FrameworkLogs-kansio sisältää kontrollerin itsensä luomat lokit. Nämä lokit ovat virheloki, yleisloki sekä testitapausloki. Virhelokiin errors.json kerätään kontrollerissa tapahtuvat virheet ja se yhdistetään myöhemmässä vaiheessa simulaattorien luomiin virhelokeihin. Yleisloki all.json sisältää kontrollerin ja järjestelmän yleiset tapahtumat, kuten käytetyt komennot. Testitapausloki testcases.json on tärkein loki, sillä se pitää kirjaa aktiivisista testitapauksista ja niihin kuuluvista simulaattorikonteista ja niiden tiedoista. Kun testitapaus käynnistetään, kyseinen tapaus ja siihen kuuluvien simulaattorien tiedot tallennetaan ”Active tests”-avaimen alle. Kun testitapaus lopetetaan, nämä tiedot siirretään saman lokin ”Closed tests”-avaimen alle. Tällä tavalla voimme pitää kirjaa ajossa olevista tapauksista ja verrata sitä järjestelmän todelliseen tilaan.

Kontrollerin ei tarvitse olla jatkuvassa ajossa, vaan käyttäjä voi halutessaan sulkea sen vaikuttamatta testitapausten toimintaan. Kontrolleri ei kuitenkaan tällöin tee terveystarkastuksia eikä kerää lokeja. Kontrollerin ei myöskään aina tarvitse luoda uutta testitapausta vaan sen avulla voidaan käynnistää aikaisemmin luotuja testitapauksia.

5.2 Käyttöliittymä ja käskyt

Kontrolleria ohjataan komentorivipohjaisella käyttöliittymällä. Käyttöliittymän avulla käyttäjä voi luoda uusia testitapauksia, käynnistää luotuja tapauksia, tutkia käynnissä olevia tapauksia sekä sammuttaa haluamansa tapauksen. Käyttöliittymän kautta voidaan myös tutkia järjestelmän terveyttä ja lukea järjestelmän tuottamia lokeja. Käyttöliittymä aukeaa heti, kun käyttäjä on käynnistänyt kontrolleriohjelman.

```
root@vili-VirtualBox:/home/vili# python3 frameworkController.py
1. Start test case
2. Show running tests
3. Stop test case
4. Generate test data set
5. Run health check
6. Read framework logs

Choose (1...6, X=return):
```

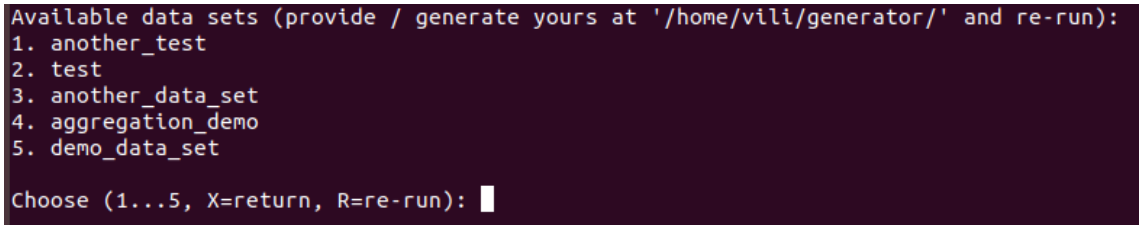
Kuva 11. *Kontrollerin käyttöliittymä käynnistämisen jälkeen*

Kuvassa 11 on kontrollerin käyttöliittymä. Käyttäjä valitsee haluamansa käskyä vastaavan numeron perusteella. Käskyn mukaan käyttäjältä voidaan pyytää lisäkäskyjä, kuten

käynnistettävän testitapauksen valitsemista. Kontrolleri näyttää tilansa käskyjen jälkeen ja kontrolleri voidaan sammuttaa X-komennolla. Kaikki kontrolleriin ja sen toimintaan liittyvät tulosteet näytetään komentorivillä.

5.2.1 Testitapauksen käynnistäminen

Kontrollerin ensimmäinen käsky on testitapauksen käynnistäminen, eli Start test case. Ennen kuin testitapaus voidaan käynnistää, kontrolleri noutaa luodut testitapaukset ja pyytää käyttäjää valitsemaan niistä haluamansa. Jos testitapauksia ei ole, palautetaan tyhjä lista ja käyttäjää kehotetaan luomaan testitapaus generaattorin avulla aliluvussa 5.2.3 kuvatun tavan mukaisesti. Luodut testitapaukset säilyvät isäntäpalvelimella, ellei niitä poisteta manuaalisesti, ja käyttäjä voikin ajaa aikaisemmin luomiaan testitapauksia.



Kuva 12. *Käynnistettävän testitapauksen valinta*

Kuva 12 näyttää, kuinka eri testitapaukset listataan kontrollerin käyttöliittymässä. Kun käyttäjä on valinnut käynnistettävän testitapauksen, kontrolleri avaa testitapauskohtaisen imageMap.json nimisen tiedoston, joka pitää sisällään testitapauksen topologian, konteissa käytettävät kuvat sekä kontin konfigurointiin ja käynnistykseen liittyvät tiedot. Tämä tiedosto on tallennettu testitapaus-kansioon yhdessä testissä käytettävän datan kanssa.

```
{
"images": [
    {
          "image":simulaattori 1:n repository,
          "containerDir": "/usr/src/simulator/files",
          "hostDir": "./sgnb_duup_003"
          },
          {
          "image":simulaatottori 2:n repository
          "hostDir": "./npc001",
          "containerDir": "/usr/src/simulator/files",
          "hostPort": 2222,
          "containerPort": 22
    }
    ]
}
```

Ohjelma 1. *Verkkotopologia imageMap.json tiedostosta*

Ohjelma 1 näyttää yksinkertaisen testitapauksen imageMap-tiedoston. Tässä esimerkissä testitapaukseen kuuluu kaksi erityyppistä simulaattoria. Molemmilla simulaattoreilla on konttikuvan osoite sekä kansiopolut, joita käytetään Docker Volumen asettamisessa: hostDir on kontrollerin ajoympäristön kansio, joka sisältää testeissä käytettävän datan ja containerDir kontin sisäinen kansio, josta simulaattori noutaa datan lähetystä varten. Tämä kansio voi myös sisältää lisäkonfiguraatioita, jos generaattori on luonut sellaisia. Toisella simulaattorilla on myös erikseen määritelty ulkoinen ja sisäinen portti, joiden kautta kyseinen simulaattori ottaa yhteyden vastaanottajaan. Näiden tietojen avulla kontrolleri voi käynnistää simulaattorikontit oikeilla konfiguraatioilla. Konttien käynnistys tapahtuu yksi kerrallaan imageMapissa olevassa järjestyksessä. Tiedoston lukeminen ja konttien käynnistys ei tuota käyttäjälle tulosteita.

```
   container = client.containers.run(simulator["image"], detach = True, labels
= {'testGroup': folderName}, ports = portMapping, volumes = {hostdir:{'bind':
simulator["containerDir"], 'mode': 'rw'}})

              newLog.append({
                  'simulatorID': container.id,
                  'simulatorName': container.name,
                     "deploymentTime":   time.strftime('%Y-%m-%d   %H:%M:%S',
      time.localtime(time.time())),
                  'elementType': simulator["hostDir"].split("/")[-1]
                  })
```

Ohjelma 2. *Simulaattorikontin käynnistäminen ja tietojen lokitus*

Simulaattorikonttien käynnistys tapahtuu Ohjelma 2:n mukaisesti Pythonin Docker-kirjaston [29] avulla. Ensimmäisellä rivillä oleva client.containers.run -komento toimii docker run -komennon tavoin ja käynnistää kontin annetuilla parametreilla. Tässä tapauksessa kontin tiedot on noudettu Ohjelma 1:n tiedoista. Kontti ajetaan irrotetussa muodossa (engl. detached mode), jolloin se pyörii järjestelmän taustalla eikä tuota tulosteita komentoriville. Kontille asetetaan myös tunnus (engl. label) testGroup-avaimen alle. Tämä tunnus on käynnistetyn testitapauksen nimi ja sen avulla on helppoa noutaa kaikki kyseiseen testitapaukseen kuuluvat simulaattorit, ja erotella samalla tavalla konfiguroituja kontteja eri testitapausten välillä.

Kontin käynnistämisen jälkeen osa sen tiedoista tallennetaan Ohjelma 2:n jälkimmäisen osan mukaisesti. Docker-kirjaston avulla voimme noutaa käynnistetyn kontin tunnisteen ja nimen container.id ja container.name avulla. Näitä tietoja käyttäen voimme verrata käynnistettyjä kontteja ajossa oleviin kontteihin ja selvittää, onko yksi tai useampi konteista sammunut virheellisesti. Lokitamme myös kontin käynnistysajan sekä kontin ele-

menttityypin, joka vastaa Ohjelma 1:ssä määriteltyä isäntäkansiota. Kaikki tiedot tallennetaan JSON-listaan, johon lisätään vielä testitapauksen nimi ennen sen tallentamista recordLogs.json-lokiin. Kun kaikki testitapauksen kontit on käynnistetty ja niiden tiedot tallennettu, kontrolleri palaa päänäkymään odottamaan käyttäjän seuraavaa komentoa.

5.2.2 Testitapausten listaus ja lopettaminen

Käyttäjä pystyy listaamaan käynnissä olevat testitapaukset Show running tests -komennolla sivun 32 kuvan 11 mukaisesti. Tällä tavalla käyttäjä voi selvittää, mitkä testitapaukset ovat jo ajossa ja minkä nimiset kontit kuuluvat mihinkin testitapaukseen.

```
Currently running test cases:

Data set: aggregation_demo
Containers:
---modest_chandrasekhar4
---interesting_ellis
---tender_lamarr
---sleepy_cannon
---charming_dewdney
---recursing_franklin
---objective_wilson
---epic_torvalds
---kind_feistel
---upbeat_buck

Data set: another_data_set
Containers:
---focused_bhabha
---focused_greider
---modest_villani
---clever_moore
---hungry_dewdney
---admiring_mendeleev
---naughty_wilbur
---sharp_mestorf
---ecstatic_proskuriakova1
---wizardly_lamport
```

Kuva 13. *Ajossa olevien testitapausten listaus*

Kuvassa 13 käyttäjä on listannut ajossa olevat testitapaukset. Kuvasta näemme, että järjestelmä ajaa tällä hetkellä kahta eri testitapausta: aggregation_demo ja another_data_set, joihin molempiin kuuluu kymmenen simulaattorikonttia. Testitapausten ja konttien tarkemmat tiedot löytyvät kontrollerin lokeista.

Kun testitapausta ei enää tarvita, käyttäjä voi lopettaa yksittäisen tapauksen Stop test case -komennolla. Käyttäjän pitää vielä valita lopetettava testitapaus uudella syötteellä, samalla tavalla kuin testitapausta käynnistettäessä.

Kuva 14. *Suljettavan testitapauksen valinta ja sulkeminen*

Kuvan 14 alussa näemme ajossa olevien testitapausten nimet. Tapausten nimet haetaan kontrollerin lokeista ”Active tests” -avaimen alta. Valitsimme näistä another_data_set **-** tapauksen ja kontrolleri listaa kyseiseen tapaukseen kuuluvat simulaattorikontit ja tuhoaa ne yksi kerrallaan. Ainoastaan kyseiseen tapaukseen kuuluvat kontit tuhotaan konttien ”testGroup”-tunnuksen perusteella, muihin testitapauksiin ei kosketa. Kun testitapaus suljetaan, kontrollerin lokit päivitetään ja testitapaus siirretään lokeissa ”Closed tests”-avaimen alle. Samalla kirjataan simulaattorikonttien sulkemisaika. Kontrolleri myös kerää simulaattorien luomat virhe- ja tietuelokit. Tietuelokit yhdistetään tässä vaiheessa ja virhelokit tallennetaan väliaikaisesti virheraportin luomiseen asti. Virheraportin luominen käydään tarkemmin läpi aliluvussa 5.2.4.

5.2.3 Generaattorin ohjaaminen

Generaattorin toiminta on käyty läpi yleisellä tasolla jo aliluvussa 4.5. Kontrollerin avulla on mahdollista käynnistää generaattorikontti ja asettaa sille tarvittavat konfigurointitiedot

samalla tavalla kuin simulaattorikonttien tapauksessa. Generaattorikontti tuottaa testidatan ja luo oikean kansiorakenteen, jonka jälkeen se sulkee itsensä. Kun testitapaus ja sen data on luotu, se voidaan käynnistää aliluvun 5.2.1 mukaisesti.

Generaattorin käyttämiä konfiguraatio- ja topologiatietoja ei voi muokata kontrollerin kautta. Käyttäjän pitää suorittaa muutokset tiedostoihin manuaalisesti, jonka jälkeen hän voi luoda testitapauksen kontrollerin kautta. Kontrolleri kirjaa generaattorin käytön omiin lokeihinsa. Sekä konfiguraatio- että topologiatiedot ovat JSON-muodossa. Konfiguraatiotietoja avataan tarkemmin taulukossa 2. Konfiguraatiotietojen luokat LoadObject, TopologyObject sekä SimulatorsObject kuvataan tarkemmin taulukon jälkeen.

Taulukko 2. *Generaattorin konfiguraatiotiedot*

Avain	Tyyppi	Selitys
dataSetName	String	Luodun testitapauksen nimi.
load	LoadObject[]	Lista LoadObjecteista, jotka määrittelevät simuloitavat tapahtumat
duration	float	Simulaation pituus durationUnits-yksikköinä.
durationUnits	String	Pituuden yksikkö, vaihtoehtoina: ”MILLIS”, ”SECONDS”, ”MINUTES”, ”HOURS”, ”DAYS”
topology	TopologyObject	Simuloitavan verkon topologia
simulators	SimulatorsObject	Simulaattorien configuraatiotiedot

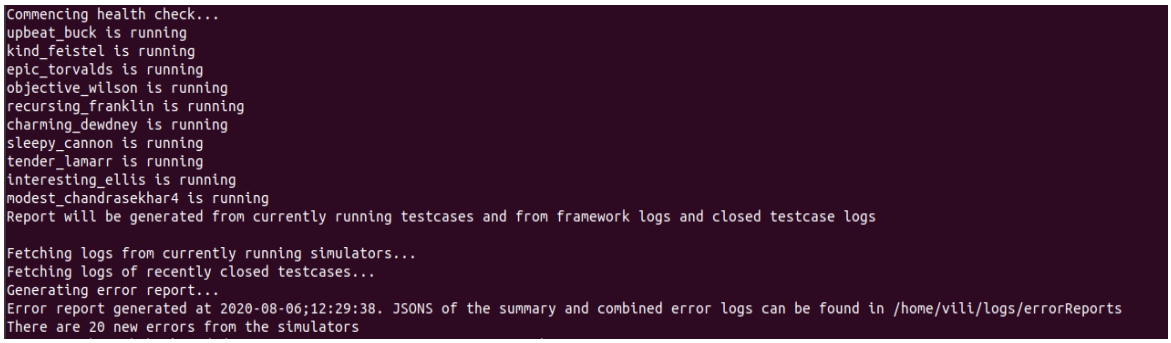
LoadObject-luokassa määritellään simuloitavan tapahtuman tiedot. Nämä tiedot sisältävät käytettävän generaattorin tyypin, datan määrä sekä datan lähetysintervallit. Erikoistapaukset, joissa datan määrä kasvaa tai laskee lähetyksen aikana, määritellään myös LoadObjectissa.

TopologyObject määrittelee testitapauksen simulaattorien topologian. Oletuksena TopologyObject on erillinen JSON-tiedosto, mutta nämä tiedot voidaan kirjata myös suoraan konfiguraatiotiedostoon. Topologia määritellään simulaattorien listana, jossa listataan simulaattorien tyyppi ja simulaattorityypistä riippuvat simulaattorit. Jotkin simulaattorityypit voivat johtaa datan luomisessa käytettyjä tietueita muista simulaattorityypeistä, esimerkiksi 5G-data voi olla riippuvaista 4G-datasta, jos testitapaukseen kuuluu molemman tyyppisiä simulaattoreita.

SimulatorObject sisältää yksittäisen simulaattorin konfiguraatiotiedot. Nämä tiedot vaihtelevat simulaattorien välillä ja niiden tarkempi määrittely on simulaattorikehittäjien vastuulla. Yleisesti tietoihin kuuluu vastaanottajan IP-osoite, portti sekä simulaattorikontin volume-kansion polku.

5.2.4 Terveystarkastus ja virheraportti

Järjestelmälle on mahdollista suorittaa terveystarkastus, jonka yhteydessä luodaan myös virheraportti, jos viime tarkastuksen jälkeen on tapahtunut virhetilanteita. Terveystarkastus voidaan suorittaa manuaalisesti erillisellä Run health check -käskyllä. Terveystarkastusta varten kontrollerissa on myös erillinen säie (engl. thread) joka suorittaa tarkastuksen minuutin väliajoin, jonka jälkeen käyttäjä voi jatkaa siitä mihin oli jäänyt ennen automaattitarkastusta. Sekä automaattisella että manuaalisella terveystarkastuksella on sama toiminnallisuus.



Kuva 15. *Järjestelmän terveystarkastus ja virheraportin luonti*

Kuva 15 esittää terveystarkastuksen ja virheraportin luomia tulosteita. Terveystarkastuksen tavoitteena on tarkistaa, että lokien mukaiset testitapaukset ovat ajossa ja kaikki käynnistetyt simulaattorikontit ovat myös ajossa. Tämä tapahtuu vertaamalla kontrollerin lokeihin tallennettujen testitapausten alaisten simulaattorien id:itä ajossa olevien konttien id:ihin. Jos yksittäinen kontti puuttuu, tämä virhe kirjataan ja käyttäjälle ilmoitetaan kyseisen kontin nimi. Jos kaikki testitapauksen kontit puuttuvat, virhe kirjataan ja käyttäjälle ilmoitetaan, mutta koko testitapaus siirretään lokeissa ”Closed tests”-avaimen alle. Tällaisessa tapauksessa testitapaus tulee käynnistää kokonaan uudelleen. Konttien puuttuminen voi johtua erinäisistä virheistä, kuten vastaanottajaan muodostetun yhteyden katkeamisesta tai simulaattorin sisäisestä ongelmasta. Virhetilanteet kerätään virheraporttiin ja toistuvien virheiden korjaaminen on kyseisestä simulaattorista vastaavan tiimin vastuulla.

Kun terveystesti on ajettu, suoritetaan myös virheraportin luonti. Jos uusia virheitä ei ole kirjattu, eli erinäisten virhelokien koko on nolla tai niitä ei ole, virheraporttia ei luoda. Virheitä kerätään kolmesta eri paikasta: suljettujen testitapausten simulaattoreilta, käynnissä olevien testitapausten simulaattoreilta ja kontrollerilta.

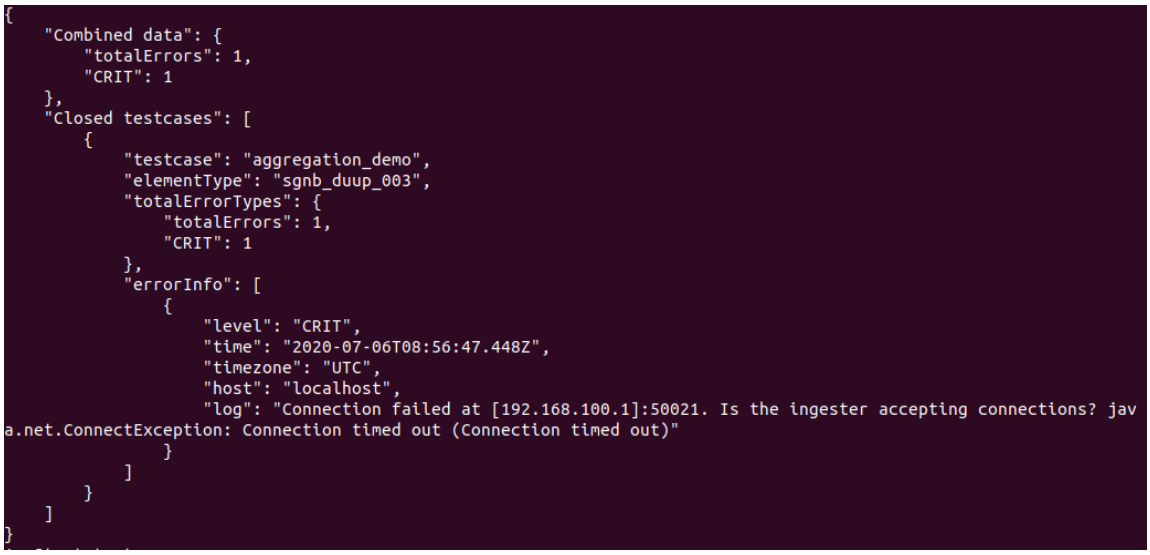
Kaikki simulaattorikontit lokittavat virheensä samaan kansioon, joka on asetettu volumeksi kontin käynnistyksen yhteydessä. Kun testitapaus suljetaan, simulaattorien luomat virhelokit kerätään väliaikaiseen virhelokiin. Tällä tavalla kaikki virheraportit säilyvät ja ne voidaan yhdistää oikeaan simulaattoriin ja testitapaukseen. Virheraporttia luodessa

tämä väliaikainen virheraportti kerätään yhdessä käynnissä olevien simulaattoreiden yksittäisten virheraporttien kanssa. Myös kontrollerin oma virheloki kerätään, ja kaikki erilliset lokit yhdistetään suureksi, keskitetyksi virheraportiksi.

Tämä keskitetty virheraportti pitää sisällään kaikkien virheiden tiedot, testitapaukset, simulaattorikontit, aikaleimat ja sijainnit. Tämän virhelokin ongelmana on kuitenkin sen koko. Vaikka kaikki virhetilanteet on kuvattu tarkasti, siitä voi olla vaikea saada yleiskuvaa järjestelmän toiminnasta ja virheiden vakavuudesta. Tätä varten luodaan myös pienempi tiivistelmäraportti.

Tiivistelmäraportti perustuu suuren virhelokin tietoihin, mutta on jäsennelty lukijaystävällisellä tavalla. Yksittäisestä virheestä on kirjattu vain tärkeimmät tiedot, kuten virheen vakavuusaste ja sijainti. Lisäksi virheiden määrästä ja vakavuudesta tehdään myös tiivistelmä. Koko järjestelmän virheiden määrä ja vakavuusaste kirjataan erikseen rekursiivisesti järjestelmätasolla, testitapaustasolla ja simulaattorikonttitasolla. Tällä tavalla tiivistelmälokia käyttämällä on mahdollista saada laaja kuva järjestelmän osien tilasta ja virheiden alkuperästä. Virheraportit tallennetaan aliluvussa 5.1 kuvatulla tavalla.

Kahdella erilaisella virhelokilla sama tapahtuma kirjataan useaan kertaan, jolloin lokien tarvitsema tallennustila kasvaa. Perustelu erikokoisille virhelokeille on järjestelmän tilan valvonnan helpottaminen, kun tiivistelmäraportin avulla saadaan selville järjestelmän tila yleisellä tasolla. Jos tiivistelmäraportista ei ilmene vakavia virhetilanteita, testien ja järjestelmän tilan valvojan ei tarvitse tehdä jatkotoimenpiteitä järjestelmän suhteen. Jos taas jossain testitapauksessa tai simulaattorikontissa ilmenee vakava virhe, valvoja voi tutkia laajan raportin tarkempia tietoja virhetilanteen selvittämiseksi.



```
{
    "Combined data": {
        "totalErrors": 1,
        "CRIT": 1
    },
    "Closed testcases": [
        {
            "testcase": "aggregation_demo",
            "elementType": "sgnb_duup_003",
            "totalErrorTypes": {
                "totalErrors": 1,
                "CRIT": 1
            },
            "errorInfo": [
                {
                    "level": "CRIT",
                    "time": "2020-07-06T08:56:47.448Z",
                    "timezone": "UTC",
                    "host": "localhost",
                    "log": "Connection failed at [192.168.100.1]:50021. Is the ingester accepting connections? jav
a.net.ConnectException: Connection timed out (Connection timed out)"
                }
            ]
        }
    ]
}
```


Kuva 16. *Esimerkki laajasta virheraportista*

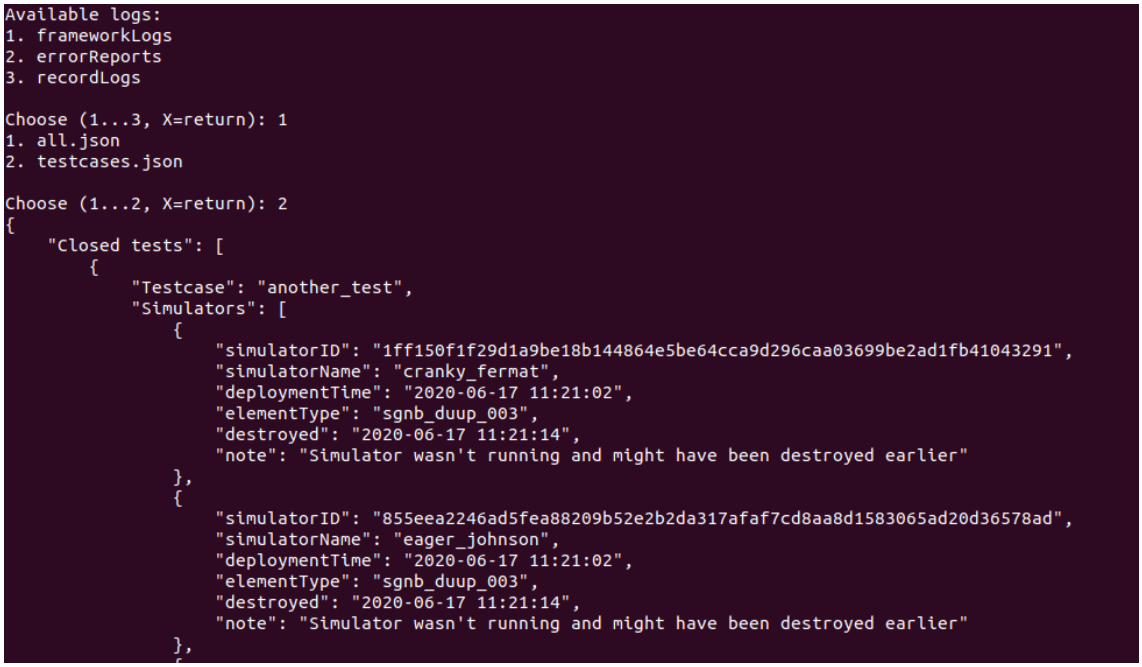
```
{
    "Closed testcases": [
        {
            "elementType": "sgnb_duup_003",
            "testcase": "aggregation_demo",
            "error logs": [
                {
                    "type": "log",
                    "host": "localhost",
                    "level": "CRIT",
                    "neid": "Samsung5GCslSimulator",
                    "system": "Samsung5gCslSimulator",
                    "time": "2020-07-06T08:56:47.448Z",
                    "timezone": "UTC",
                    "log": "Connection failed at [192.168.100.1]:50021. Is the ingester accepting connections? jav
a.net.ConnectException: Connection timed out (Connection timed out)\n\tat java.net.PlainSocketImpl.socketConnect(N
ative Method)\n\tat java.net.AbstractPlainSocketImpl.doConnect(AbstractPlainSocketImpl.java:350)\n\tat java.net.Ab
stractPlainSocketImpl.connectToAddress(AbstractPlainSocketImpl.java:206)\n\tat java.net.AbstractPlainSocketImpl.co
nnect(AbstractPlainSocketImpl.java:188)\n\tat java.net.SocksSocketImpl.connect(SocksSocketImpl.java:392)\n\tat jav
a.net.Socket.connect(Socket.java:607)\n\tat java.net.Socket.connect(Socket.java:556)\n\tat java.net.Socket.<init>(
Socket.java:452)\n\tat java.net.Socket.<init>(Socket.java:229)\n\tat com.nokia.ca4mn.simu.samsung.simulation.Samsu
ngSimulator.connectSocket(SamsungSimulator.java:134)\n\tat com.nokia.ca4mn.simu.samsung.simulation.SamsungSimulato
r.readProperties(SamsungSimulator.java:111)\n\tat com.nokia.ca4mn.simu.samsung.simulation.SamsungSimulator.<init>(
SamsungSimulator.java:66)\n\tat com.nokia.ca4mn.simu.samsung.simulation.SamsungSimulator.startSimulator(SamsungSim
ulator.java:56)\n\tat com.nokia.ca4mn.simu.samsung.simulation.RestApp.main(RestApp.java:15)\n"
                }
            ]
        }
    ]
}
```


Kuva 17. *Esimerkki keskitetystä virheraportista*

Kuvia 16 ja 17 vertailemalla näemme tiivistetyn raportin ja keskitetyn raportin erot. Molemmat raportit käsittelevät samaa tilannetta, jossa simulaattori ei ole saanut luotua yhteyttä vastaanottajaan. Tiivistetyssä raportissa on erottelu virheiden määrästä ja sijainnista. Tässä tapauksessa virheitä on vain yksi, mutta jos ajaisimme toisen simulaattorin eri testitapauksessa, kyseisen testitapauksen virheet eriteltäisiin ”Closed testcases” listan toiseen alkioon. Vain virheen tärkeimmät tiedot sisällytetään tiivistettyyn raporttiin. Vastaavasti keskitetty virheraportti sisältää virheen tiedot sen alkuperäisessä muodossa.

5.2.5 Lokien tutkiminen

Kontrollerilla voidaan myös lukea kaikkia järjestelmän luomia lokeja ja virheraportteja. Nämä lokit ja raportit näytetään kontrollerin kautta komentorivillä. Lokit tallennetaan isäntäpalvelimelle, joten niitä on mahdollista lukea myös ulkoisen ohjelman avulla. Tallennettuihin lokeihin voidaan kirjata lisää tapahtumia, mutta niitä ei poisteta automaattisesti. Suuressa testauskäytössä olevalla palvelimella lokien viemät tilaresurssit kasvavat, mutta tavallisessa testauskäytössä tämän ei pitäisi olla ongelma.



Kuva 18. *Lokien lukeminen komentoriviltä*

Kuvassa 18 käyttäjä on navigoinut haluamaansa lokikansioon ja valinnut luettavan lokin. Luettava loki näkyy osittain kuvan alaosassa ja kyseessä on kontrollerin luoma testcase-loki, joka pitää kirjaa ajettavista ja suljetuista testitapauksista ja niiden simulaattoreista. Kuvasta näemme ”Closed tests” -avaimen, jonka alle listataan suljetut testitapaukset, kuten ”another_test” -niminen tapaus. Kyseisen tapaukseen liittyvien simulaattorien tiedot näytetään myös, kuten simulaattorin käynnistys- ja sulkemisaika.

6. TOTEUTUKSEN TULOKSET JA ARVIOINTI

Voimme arvioida hallintajärjestelmän prototyypin onnistumista ja tuloksia käytännöllisellä ja teoreettisella tasolla. Hallintajärjestelmän vaatimukset esitettiin aliluvussa 4.1 ja ne jaettiin kolmeen osa-alueeseen: simulaattorit, monitorointi ja testidata. Näiden vaatimusten lisäksi tässä luvussa käsitellään kehityksen aikana syntyneitä lisävaatimuksia sekä hallintajärjestelmän saamaa palautetta.

Hallintajärjestelmän ensimmäinen osa-alue on simulaattorit, joka kattaa testeissä käytettävien simulaattorien ohjaamisen ja konfiguroinnin. Tämä on toteutettu kontrollerin suorittamalla orkestraatiolla ja yksittäisen kontin tarkempi toiminta on abstraktoitu kontrollerin käyttöliittymän taakse. Käyttäjän ei tarvitse tuntea simulaattoreiden toimintaa tai käyttöä niiden hallitsemiseksi. Simulaattorien ja kontrollerin väliset verkkokutsut on minimoitu, jolloin ajoympäristön resurssit saataisiin varattua simulaattoreiden käyttöön. Hallintajärjestelmän sisältämien mikropalveluiden rajapinnat on määritelty, minkä ansiosta järjestelmä ei ole sidottu mikropalveluiden tiettyihin versioihin ja palveluita voidaan päivittää ilman että hallintajärjestelmään tarvitsee tehdä muutoksia. Tällä tavalla voimme käyttää hyväksi mikropalveluiden itsenäisyyttä, teknologia-agnostisuutta ja ylläpidettävyyttä [4,6]. Sovitut rajapinnat auttavat myös mikropalveluiden sopimustestauksessa, kun rajapinnan mukainen sopimus on täytettävä [14].

Järjestelmän monitorointi tapahtuu säännöllisten terveystarkastusten ja erilaisten lokien avulla. Mikropalvelupohjaisten järjestelmien rakenne voi monimutkaistua nopeasti ja ilman kattavaa monitorointia järjestelmän tilanvalvonta on miltei mahdotonta [1,3,4]. Mikropalveluiden luomat lokit irrotetaan itse mikropalvelun elinkaaresta, jotta tiedot säilyisivät tulevaisuutta varten ja luotuja lokeja voidaan käyttää testiraporttien pohjana tai lisämateriaalina. Lokien syntaksin tulee samanaikaisesti näyttää lukijalle tarpeellinen tieto, että karsia epäoleellista tietoa. Usein tietojen tärkeys voi selkeytyä vasta järjestelmän käytön myötä, esimerkiksi tiivistetty virheloki ei kuulunut alkuperäisiin suunnitelmiin mutta suuren virhelokin koon kasvaessa syntyi tarve tietojen tiivistämiselle.

Testidatan luominen oman mikropalvelun avulla ratkaisee aikaisemmat dataan liittyvät ongelmat. Hallintajärjestelmän avulla on mahdollista luoda juuri halutun kokoista dataa testejä varten, mikä auttaa erityisesti suurta kuormaa vaativassa ei-funktionaalisessa testaamisessa [15,16]. Testien tulosten vertailu on myös helpompaa, kun testitapauksia

ja testidataa voidaan käyttää uudelleen eri ympäristöissä ja VAJ:n kokoonpanoilla. Testidatan luomisparametreja ja testitapauksen verkkotopologiaa voidaan myös hyötykäyttää testiraporteissa.

Kokonaisuutena järjestelmä ratkaisee useita VAJ:n testaamiseen liittyneitä ongelmia. Kontrollerin avulla simulaattorien valmisteluun kuluva aika saadaan tiputettua murto-osaan aikaisemmasta, jolloin testien suorittaminen voidaan aloittaa nopeammin. Testaajan ei myöskään tarvitse opetella useiden erityyppisten simulaattoreiden käyttöä, vaan pelkkä kontrollerin toiminnan ymmärtäminen riittää. Testien valmisteluun kuluva aika vähenee, minkä seurauksena testeihin voidaan käyttää enemmän aikaa tai ne saadaan suoritettua nopeammin. Testien tulosten perusteella tehdyt korjaukset parantavat tuotteen laatua ja voi ehkäistä kriittisiä ongelmia käytön aikana [9,15]. Hallintajärjestelmää voidaan käyttää myös pienemmissä testeissä, kuten palvelutestauksessa. Tällöin simulaattoreita olisi ajossa vain yksi, mutta datan generointi ja simulaattorin käyttö toimisi muuten samalla tavalla kuin isommissakin testeissä.

Kaikki järjestelmän ominaisuudet on dokumentoitu yhtiön sisällä. Tämä dokumentaatio kattaa hallintajärjestelmän rakenteen, kontrollerin toiminnan sekä simulaattorien rajapinnat. Kattavan dokumentaation avulla uusien simulaattorien ja hallintajärjestelmän lisäominaisuuksien kehittäminen on yksinkertaista. Dokumentaatiota seuraamalla hallintajärjestelmällä voidaan suorittaa halutunlaisia testejä.

Järjestelmässä on kuitenkin joitain puutteita ja jatkokehitystarpeita. Järjestelmän luomien lokien lukeminen tapahtuu komentorivin avulla, jonka käyttö voi olla raskasta. Lokeja voi myös tarkastella vain järjestelmän ajoympäristöstä ja niissä ei ole hakuominaisuutta tiedon nopeaa löytämistä varten. Kontrolleri voitaisiin myös muuttaa konttipohjaiseksi ja verkkokutsujen avulla ohjattavaksi simulaattoreiden tapaan. Tällöin kaikki hallintajärjestelmän osat voidaan noutaa Docker-varastosta ja kontrolleri voidaan ottaa osaksi mikropalveluiden DevOps-automaattitestausta [8]. Testien koon kasvaessa järjestelmän pitää ohjata satoja tai tuhansia simulaattoreita, jotka voivat viedä enemmän resursseja kuin mitä yhdellä palvelimella on tarjolla. Järjestelmää tulee ennen pitkää laajentaa käyttämään useita palvelimia. Kontteja ei myöskään ole mahdollista konfiguroida uudelleen niiden ollessa käynnissä. Tätä varten testitapaus tulee sulkea ja käynnistää uudestaan. Tällöin voi kuitenkin olla tarve generoida uusi testitapaus, jos uudelleen konfiguroinnin syynä oli virheellinen data tai simulaattorikonttien virheelliset konfiguraatiotiedot. Muutoin samat ongelmat jatkuvat testitapauksen uudelleenkäynnistyksestä huolimatta. Tätä ongelmaa on jossain määrin lievennetty testitapauksen nopean käynnistämisen ansiosta.

Järjestelmää on testattu oikeaa testiympäristöä muistuttavalla palvelimella. Yhdessä testissä luotiin neljä erilaista testitapausta, jotka kattoivat 34 simulaattoria. Testitapaukset aloitettiin peräkkäin ja suljettiin yksi kerrallaan epätasaisin väliajoin. Kontrolleri pystyi pitämään kirjaa ja hallitsemaan eri testitapauksia siten, että yhden testitapauksen toiminta ei vaikuttanut muiden testitapausten toimintaan. Simulaattoreissa ei ilmennyt ongelmia tänä aikana ja ympäristön resurssit riittivät. Tämä voi kuitenkin vaihdella simulaattorityyppien, testissä käytetyn datan koon ja palvelimen resurssien mukaan.

Testitapauksen käynnistyksen nopeutta testattiin käynnistämällä kymmenen simulaattorin kokoinen testitapaus. Sama testitapaus käynnistettiin myös manuaalisesti ilman erillistä valmistautumista sekä manuaalisesti valmiiksi kirjoitettujen komentojen avulla. Käytetty testidata oli sama eri testeissä ja käytettävät simulaattorit toimivat aliluvussa 4.4 vaatimusten mukaisesti. Simulaattorien käyttämä data oli myös valmiina eikä vastaanottajiin tai ympäristöihin tarvinnut tehdä muutoksia. Pelkästään näiden ominaisuuksien ansiosta testin valmistelu on yksinkertaisempaa kuin aikaisemmin. Taulukossa 3 vertaillaan testien tuloksia.

Taulukko 3. *Testien käynnistystapojen nopeusvertailu*

	Manuaalisesti ilman valmisteluja	**Manuaalisesti valmiilla komennoilla**	**Automatisoitu järjestelmä**
Testidatan luonti	-	-	1.54s
Testitapauksen käynnistys	10min	76s	5.34s
Testitapauksen sammuttaminen	112s*	112s*	2.44s

*Manuaalisessa sammuttamisessa käytettiin samoja komentoja.

Testien tulosten perusteella voimme todeta järjestelmän olevan huomattavasti nopeampi testien käynnistämisessä kuin kumpikaan manuaalinen tapa. Kun testejä käynnistetään manuaalisesti ilman valmisteluja, konteille asetettavien kansioiden tiedot piti tarkistaa jokaisen käynnistyksen yhteydessä ja komentoja kirjoittaessa voi tapahtua käyttäjästä johtuneita virheitä. Toisessa manuaalisessa testissä komennot oli kirjoitettu valmiiksi ja niitä voitiin käyttää sellaisenaan konttien käynnistämisessä. Kontrollerin avulla voimme luoda ja käynnistää useita testitapauksia samassa ajassa, mikä menisi yhden tapauksen manuaaliseen käynnistämiseen. Kontrolleri luo myös lokeja ja testitapausten orkestrointiin liittyviä toimenpiteitä, jotka vievät osan testitapauksen käynnistämiseen ja sulkemiseen kuluvasta ajasta.

Tapausten sammuttaminenkin on nopeaa, mikä lieventää aikaisemmin mainittua ongelmaa uudelleenkonfiguroinnissa. Kun testitapaus voidaan sulkea, sille voidaan luoda uusi

data ja konfiguraatiotiedot ja käynnistää testitapaus näillä uusilla tiedoilla kymmenen sekunnin sisällä. Testitapausten nopea käynnistäminen ja sulkeminen auttaa myös tilanteissa, joissa ei voida olla varmoja testausympäristön tai testattavan järjestelmän resurssien määrästä. Tällöin resurssien kulutusta voidaan testata yhdellä kuormalla ja tulosten mukaan voidaan luoda sopivampi kuorma ja käynnistää kyseinen testitapaus nopeasti.

Koska simulaattoreita käytetään mikropalveluina, on vain luonnollista käyttää hyödyksi mikropalveluarkkitehtuurin tekniikoita hallintajärjestelmän toiminnassa. Orkestraation avulla simulaattoreiden hallinta on helppoa niiden määrästä ja tyypistä riippumatta. Monitoroimalla simulaattorien tilaa ja luomalla lokeja myös testausohjelmistossa tapahtuvat virhetilanteet voidaan havaita ja ottaa huomioon testejä analysoitaessa. Koko testausjärjestelmä on itsenäinen, skaalautuva ja helppokäyttöinen kokonaisuus.

Käyttämällä mikropalveluille ominaisia tekniikoita on mahdollista yksinkertaistaa monimutkaisiakin testausjärjestelmiä. Mikropalveluiden käytön ei tulisi olla rajattua pelkästään asiakkaille tuotettaviin tuotteisiin, vaan niitä voidaan käyttää hyväksi myös muilla ohjelmistokehityksen osa-alueilla. Tässä työssä kehitetyn kontrollerin kaltainen mikropalvelu on mahdollista yleistää käyttämään erilaisia matkijoita tai testausohjelmistoja, tai vastaavaa ohjelmaa voidaan käyttää testattavan järjestelmän hallitsemisessa.

7. HALLINTAJÄRJESTELMÄN JATKOKEHITYS

Hallintajärjestelmän kehityksen aikana saadun palautteen perusteella järjestelmälle on suunniteltu jatkokehitystä. Jatkokehityksen tarkoituksena on tehdä järjestelmästä helpokäyttöisempi ja parantaa sen jo olemassa olevia ominaisuuksia. Tässä luvussa käydään läpi nämä jatkokehityssuunnitelmat.

7.1 Kontrolleri

Kontrollerin ohjaaminen komentorivin avulla vaatii käyttäjältä pääsyä kontrollerin ajoympäristöön. Tämä ei aina ole mahdollista, joten kontrollerin ohjaustapa vaihdetaan tulevaisuudessa simulaattorien tapaan REST-pohjaiseksi [30]. REST-rajapinnalle voidaan kehittää oma selaimella ajettava käyttöliittymä [35], jolla voidaan ohjata helposti asennusympäristön ulkopuolelta. Testausjärjestelmän asennusympäristöstä voidaan tehdä eräänlainen keskus, jolle eri käyttäjät voivat lähettää REST-komentoja testitapausten luomista ja ajamista varten.

Rajapinnan implementoiminen vaatii muutoksia kontrollerin käyttöön. Kaikki kontrollerin tulosteet täytyy muuttaa siten, että tuloste palautetaan osana REST-vastausta JSON-muodossa. Osa kontrollerin toiminnoista tulostaa valittavia vaihtoehtoja ja odottaa käyttäjältä lisäsyötettä, kuten käynnistettävää testitapausta valittaessa. Tämänkaltaista toiminnallisuutta tulee muuttaa niin, että valittavat vaihtoehdot noudetaan erillisellä REST-komennolla, jonka jälkeen käyttäjä voi antaa valintansa osana toista komentoa.

Hallintajärjestelmän sisäinen orkestrointi perustuu itseluotuun logiikkaan, mutta tämä voidaan muuttaa käyttämään Docker Compose-ominaisuutta [22,23]. Käytännössä konttien orkestrointi tapahtuisi edelleen testitapausten perusteella generaattorin luoman Docker Compose-tiedoston kautta. Tämä orkestrointimenetelmä automatisoi kontrollerin toimintaa ja erillistä Docker-kirjastoa ei tarvitse asentaa. Kontrolleri pitää muuttaa myös konttimuotoon, jolloin koko hallintajärjestelmä voidaan noutaa samasta varastosta.

7.2 Monitorointi

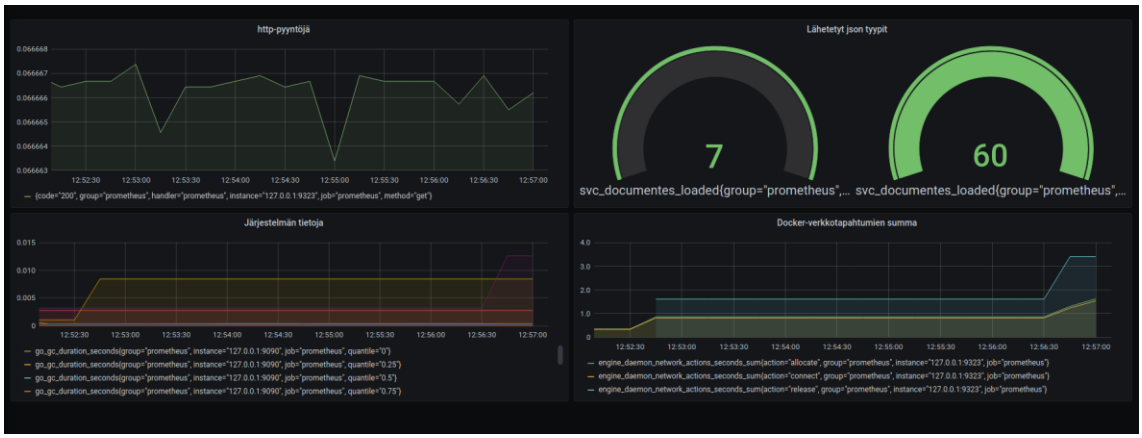
Kontrollerin luomat ja keräämät lokit voivat kasvaa vaikeasti luettaviksi ja tutkittaviksi simulaattorien ja testitapausten määrien kasvaessa. JSON-muodossa olevista lokeista ei myöskään suoraan näy muuttuvia trendejä tai testiympäristön resurssien statusta. Nämä ominaisuudet vaativat erillisiä komentoja tai tulkintoja testien valvojan puolesta. Järjes-

telmän erilaiset metriikat pitäisi kerätä, ryhmitellä ja esitellä testien valvojille, jotta järjestelmän tilan valvonta olisi yksinkertaista ja suoraviivaista. Tätä tarkoitusta varten on tehty suunnitelmat Prometheus- [36] ja Grafana-ohjelmien [18] integroinnista järjestelmään.

Prometheus on avoimen lähdekoodin monitorointijärjestelmä, joka voi kerätä järjestelmän tietoja tasaisin väliajoin erilaisten lähettäjäohjelmien (engl. exporter) avulla. Lähettäjiä on valmiina monia erilaisia ja niitä on mahdollista kehittää lisää omia tarkoituksia varten [37]. Sekä Prometheus-palvelinta että sen lähettäjiä on mahdollista käyttää Docker-kontteina tai erillisinä ohjelmina. Prometheuksen avulla kerättyä dataa voidaan visualisoida ja sille voidaan suorittaa erilaisia hakuja Prometheuksen oman kielen avulla [38]. Tällä tavalla lähettäjien antama ”raakadata” voidaan muuttaa luettavampaan ja hyödyllisempään muotoon, esimerkiksi järjestelmän keskimääräisen resurssinkulutuksen näyttämiseksi.

Osa järjestelmään suunnitelluista lähettäjistä on jo valmiina joko Prometheus-tiimin tai kolmannen osapuolen toimesta. Nämä lähettäjät liittyvät isäntäjärjestelmän ja konttien tietojen keräämiseen. Järjestelmän itse luomien lokien lähettämistä varten täytyy kuitenkin luoda omat JSON-lähettäjät. Näiden lähettäjien tulee lukea järjestelmän luomia lokeja, valita lokeista Prometheukseen lähetettävät tiedot ja muuttaa ne sopivaan muotoon. Tämän jälkeen Prometheus voi noutaa ne samalla tavalla kuin muistakin lähettäjistä.

Prometheus sisältää visualisointityökalun, mutta Grafanan avulla voidaan luoda erilaisia paneeleja, joita on mahdollista muokata monipuolisemmin kuin Prometheuksen vastaavia ominaisuuksia. Grafana toimii omalla palvelimellaan ja sen ominaisuuksia voi käyttää selaimen avulla. Grafana tukee Prometheus-integraatiota, joten järjestelmien yhdistäminen on helppoa. Toimiakseen Grafana vaatii vain ajossa olevan palvelimen, johon järjestelmän oma Prometheus-palvelin yhdistetään.



Kuva 19. *Esimerkki Grafanan paneeleista*

Kuten kuva 19 osoittaa, Grafanan avulla on mahdollista visualisoida erilaiset tiedot selkeästi ja toisistaan erottuvasti. Järjestelmässä tapahtuvat muutokset ja trendit ovat myös helposti seurattavissa. Testien suorittajat voivat luoda valmiita paneeleja, joita on mahdollista käyttää uudelleen eri ympäristöissä, tai jokaista testiä varten voidaan luoda oma paneelinsa juuri testille tärkeiden tietojen tarkkailemiseksi. Tietoja voidaan säilyttää Prometheuksessa pitkiä aikoja ja Grafanan paneelit voivat noutaa ja näyttää näitä tietoja viimeisten minuuttien tai päivien ajalta. Integroimalla Prometheuksen ja Grafanan osaksi järjestelmän tilanvalvontaa ja metriikan tutkintaa ratkaisemme lokien tutkimisen kömpelyyden ja parannamme järjestelmän monitorointiominaisuuksia.

7.3 Automatisoitu komponenttitestaus

Testausjärjestelmä on suunniteltu pitkäkestoiseen, usean mikropalvelun kattavaan testaukseen. Testausjärjestelmää voidaan kuitenkin käyttää myös yksittäisen mikropalvelun testauksessa. Vastaanottajien kehityksessä käytetään DevOps [8] ja Jenkins automaatioputkea [39] jossa vastaanottajille suoritetaan automaattisia testejä jokaisen päivityksen yhteydessä.

Testausjärjestelmä voidaan asettaa osaksi Jenkins-automaatioputkea esimerkiksi Robot-kehyksen [40] avulla. Robotin avulla vastaanottajalle voidaan määritellä automatisoituja testejä, jotka käyttäisivät testausjärjestelmän toimintoja esimerkiksi hyväksymistestauksessa. Testausjärjestelmä voi luoda testidatan vastaanottajan tyyppiselle simulaattorille ja käynnistää simulaattorin hetkeksi. Vertailemalla simulaattorin luomia lokeja vastaanottajan lokeihin voimme tarkistaa vastaanottajan oikean toiminnan testidatan käsittelyn ja vastaanottamisen suhteen.

7.4 Konttien hallinta ja skaalaaminen

Testitapausten koon ja konttien määrän kasvaessa niiden hallinta muuttuu vaikeammaksi. Pienellä määrällä simulaattorikontteja yllättäen sammunut kontti voidaan käynnistää uudestaan helposti, mutta konttien määrän kasvaessa myös virhetilanteiden määrä kasvaa. Järjestelmän isäntäympäristön resurssit rajoittavat myös samanaikaisessa ajossa olevien konttien määrää. Testitarpeiden muuttuessa ja tarvittavien simulaattorien määrän kasvaessa hallintajärjestelmälle täytyy varata enemmän resursseja, esimerkiksi jakamalla sen toiminta usealle palvelimelle.

Kubernetes on konttienhallintaohjelmisto, jonka avulla konttien skaalaaminen ja tilantarkkailu on helppoa ja automatisoitua. Ajossa olevien konttien määrä voidaan määritellä erikseen ja Kubernetes pitää huolen, että kyseinen määrä kontteja on ajossa ja vastaa kutsuihin. Ongelmatilanteissa Kubernetes pyrkii käynnistämään kontin uudelleen. Kuberneteksessa on myös valmis toiminnallisuus kuorman jakamista, tallennustilan asettamista sekä salaisten tietojen hallitsemista varten [41].

Kuberneteksen avulla kontit voidaan jakaa solmujen (engl. node) välille [42]. Nämä solmut voivat sijaita erillisillä isännillä, jolloin testitapaukset voidaan jakaa eri palvelimien välillä, moninkertaistaen käytössä olevat resurssit. Kubernetes tarkkailee myös solmujen tilaa ja käynnistää kontteja vain terveille solmuille, ja jakaa konttien kuorman automaattisesti solmujen välillä. Kuormanjako-ominaisuutta voidaan käyttää simulaattoreiden kanssa siten, että generaattori luo suuren määrän dataa, joka jaetaan automaattisesti usean saman tyyppisen simulaattorin lähetettäväksi. Tällä tavalla voimme testata mikropalveluiden toimintaa niiden saadessa dataa useasta lähteestä ja testattavan järjestelmän reagointia tähän.

8. YHTEENVETO

Mikropalveluiden suosio on kasvanut viime aikoina ja yhä useampi järjestelmä perustuu mikropalveluiden käyttöön. Mikropalvelupohjaisten järjestelmien testaus eroaa kuitenkin perinteisestä testauksesta ja voi vaatia erikoistuneen testausjärjestelmän kehittämistä. Tämän työn tarkoituksena oli suunnitella ja kehittää mikropalveluiden testauksen hallintajärjestelmä pitkäaikaista ja vaihtelevaa testausta varten. Tämä järjestelmä koostuu itsekin mikropalveluista, jonka seurauksena järjestelmän tulee myös kattaa testauksessa käytettyjen palveluiden orkestrointia ja monitorointia.

Eri henkilöt ja tiimit toteuttivat järjestelmän eri osat. Itse toteutin kontrollerin ja osallistuin simulaattorien rajapintojen ja toiminnallisuuden suunnitteluun. Generaattori suunniteltiin ja toteutettiin tämän työn ulkopuolella ja sen toimintaa ei ole sen takia käyty läpi tarkemmin. Tämä työ keskittyi juuri kontrollerin toteuttamiseen ja järjestelmän muiden osien yhdistämiseen kontrollerin avulla.

Uuden testausjärjestelmän tuli olla skaalattavissa erikokoisiin ja -tyyppisiin testeihin. Järjestelmän piti myös korjata aikaisemman, manuaalisesti valmisteltavan testaustavan puutteita. Testausjärjestelmä yhdistäisi testidatan luomisen, testauksessa käytettyjen simulaattoreiden valmistelun ja testitapausten valvonnan yhteen ohjelmaan, jonka käyttö vähentäisi testien valmisteluun kuluvaa aikaa ja mahdollistaisi laajoja testejä sopivalla datalla. Kaikki tämä nostaa testien ja niiden tulosten laatuja. Yhdessä testausjärjestelmän suunnittelun kanssa simulaattoreille suunniteltiin yhteinen toimintatapa ja rajapinta, jotta erilaiset simulaattorit voisivat toimia osana testausjärjestelmää.

Hyödyntämällä mikropalveluiden orkestraatiota, Docker-konttien ominaisuuksia ja kattavaa monitorointia hallintajärjestelmää voidaan käyttää laajoissa ja monimutkaisissa testeissä. Testauksessa käytettävien simulaattoreiden määrää ja tyyppiä voidaan vaihdella testein tarpeiden mukaan. Yhtenäiset lokit niin testausjärjestelmän kuin VAJ:nkin välillä auttavat testien analysoinnissa ja virhetilanteiden paikantaminen on yksinkertaisempaa. Helppokäyttöisen käyttöliittymän avulla kokematonkin käyttäjä voi suorittaa testien ajon alusta loppuun ja hallita useita samanaikaisia testitapauksia. Hallintajärjestelmää voidaan kuitenkin vielä kehittää testaajilta saadun palautteen ja muuttuvien testaustarpeiden perusteella.

Käytännön tasolla luotu hallintajärjestelmä auttaa testien valmistelussa ja suorittamisessa. Yhtenäinen simulaattorirajapinta helpottaa uusien simulaattorien kehitystä ja testausjärjestelmä voi käyttää kaikkia rajapinnan toteuttaneita simulaattoreita. Järjestelmää

voidaankin skaalata kattamaan uusia simulaattoreita ja erilaisia testaustilanteita. Käyttämällä kehitetyn kontrollerin kaltaista lähestymistapaa muunkinlaisten mikropalvelujärjestelmien ohjaus on mahdollista keskittää yhden mikropalvelun alle. Mikropalveluarkkitehtuurin ominaisuudet ja potentiaaliset ongelmat tulee myös ottaa huomioon kaikenkokoisissa ja laisissa mikropalveluista koostuvissa järjestelmissä. Tämä perusteella johdannossa ja aliluvussa 4.1 työlle esitetyt tavoitteet on saavutettu.

LÄHTEET

[1] Soldani J, Tamburri DA, Van Den Heuvel W. The pains and gains of microservices: A Systematic grey literature review. The Journal of systems and software 2018;146:215-232.

[2] J. Thönes. Microservices. IEEE Software 2015;32(1):116.

[3] Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. - 2017 IEEE International Conference on Software Architecture (ICSA); 2017.

[4] Newman S. Building Microservices. : O'Reilly Media, Inc; 2015.

[5] Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. - 2017 IEEE International Conference on AI & Mobile Services (AIMS); 2017.

[6] X. Larrucea, I. Santamaria, R. Colomo-Palacios, C. Ebert. Microservices. IEEE Software 2018;35(3):96-100.

[7] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, N. Josuttis. Microservices in Practice, Part 1: Reality Check and Service Design. IEEE Software 2017;34(1):91-98.

[8] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano. DevOps. IEEE Software 2016;33(3):94-100.

[9] Crispin L, Gregory J. Agile testing : a practical guide for testers and agile teams. Upper Saddle River, NJ: Addison-Wesley; 2009.

[10] Fowler M. Mocks aren't stubs. 2007; Available at: https://martinfowler.com/articles/mocksArentStubs.html. Viitattu 29.10., 2020.

[11] Spadini D, Aniche M, Bruntink M, Bacchelli A. Mock objects for testing java systems: Why and how developers use them, and how they evolve. Empirical software engineering : an international journal 2019;24(3):1461-1498.

[12] Automation of regression test in microservice architecture. - 2018 4th International Conference on Web Research (ICWR); 2018.

[13] Fellows M. What is contract testing and why should I try it? 2019; Available at: https://pactflow.io/blog/what-is-contract-testing/. Viitattu 29.10., 2020.

[14] Heckel R, Lohmann M. Towards Contract-based Testing of Web Services. Electronic notes in theoretical computer science 2005;116:145-156.

[15] Agile Team Members Perceptions on Non-functional Testing: Influencing Factors from an Empirical Study. - 2016 11th International Conference on Availability, Reliability and Security (ARES); 2016.

[16] A. Akbulut, H. G. Perros. Performance Analysis of Microservice Design Patterns. IEEE Internet Computing 2019;23(6):19-27.

[17] Gremlin: Systematic Resilience Testing of Microservices. - 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS); 2016.

[18] What is Grafana? Available at: https://grafana.com/docs/grafana/latest/getting-started/what-is-grafana/. Viitattu 3.8., 2020.

[19] Overview of virtualization in cloud computing. - 2016 Symposium on Colossal Data Analysis and Networking (CDAN); 2016.

[20] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi. Cloud Container Technologies: A State-of-the-Art Review. IEEE Transactions on Cloud Computing 2019;7(3):677-692.

[21] Container and Microservice Driven Design for Cloud Infrastructure DevOps. - 2016 IEEE International Conference on Cloud Engineering (IC2E); 2016.

[22] Nickoloff Jeffrey KS. Docker in Action. 2nd ed.: Manning Publications Co.; 2019.

[23] Schenker GN. Learn Docker - Fundamentals of Docker 19.x - Second Edition. 2nd ed.: Packt Publishing; 2020.

[24] Yādava SC, Singh SK(. An introduction to client/server computing. New Delhi: New Age International P Ltd., Publishers; 2009.

[25] Beazley DM. Python. 3rd ed. Place of publication not identified: Sams; 2006.

[26] Python Package Index. Available at: https://pypi.org/. Viitattu 3.8., 2020.

[27] Docker object labels. Available at: https://docs.docker.com/config/labels-custom-metadata/. Viitattu 11.8., 2020.

[28] Docker: Container Networking. Available at: https://docs.docker.com/config/containers/container-networking/. Viitattu 3.8., 2020.

[29] Docker SDK for Python. Available at: https://docker-py.readthedocs.io/en/stable/. Viitattu 3.8., 2020.

[30] What is REST? Available at: https://restfulapi.net/. Viitattu 3.8., 2020.

[31] Introducing JSON. Available at: https://www.json.org/json-en.html. Viitattu 3.8., 2020.

[32] Apache Log4j. Available at: https://logging.apache.org/log4j/2.x/. Viitattu 3.8., 2020.

[33] Log4python. Available at: https://pypi.org/project/log4python/). Viitattu 3.8., 2020.

[34] Docker: Use Volumes. Available at: https://docs.docker.com/storage/volumes/. Viitattu 3.8., 2020.

[35] OpenAPI Specification. Available at: https://swagger.io/resources/open-api/. Viitattu 3.8., 2020.

[36] Prometheus Overview. Available at: https://prometheus.io/docs/introduction/overview/). Viitattu 3.8., 2020.

[37] Exporters and Integrations. Available at: https://prometheus.io/docs/instrumenting/exporters/. Viitattu 17.8., 2020.

[38] Querying Basics. Available at: https://prometheus.io/docs/prometheus/latest/querying/basics/).

[39] Jenkins Handbook. Available at: https://www.jenkins.io/doc/book/. Viitattu 3.8., 2020.

[40] Robot Framework. Available at: https://robotframework.org/. Viitattu 29.10., 2020.

[41] What is Kubernetes? Available at: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Viitattu 20.8., 2020.

[42] Nodes. Available at: https://kubernetes.io/docs/concepts/architecture/nodes/. Viitattu 20.8., 2020.