Eemeli Kalliokoski

Kulunvalvontajärjestelmän skaalautumis-testausohjelman suunnittelu ja toteutus

Insinööri

Tieto- ja viestintätekniikka

Kevät 2023

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Kalliokoski Eemeli

Työn nimi: Kulunvalvontajärjestelmän skaalautumistestausohjelman suunnittelu ja toteutus

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: Kulunvalvontajärjestelmä, skaalautuminen

Opinnäytetyön tilaajalle Fisplay Oy:lle syntyi tarve mitata kulunvalvontajärjestelmänsä suorituskykyä. Työn aiheeksi muodostui kulunvalvontajärjestelmän skaalautumisohjelman suunnittelu ja toteutus. Työn tarkoituksena oli suunnitella sekä toteuttaa prototyyppiohjelma, jonka avulla kohdekulunvalvontajärjestelmän suorituskykyä pystytään mittaamaan. Opinnäytetyön tekijällä on työkokemusta kulunvalvontajärjestelmien parissa.

Työssä käytiin läpi, millaisista osista kulunvalvontajärjestelmä rakentuu sekä niiden yleinen toimintaperiaate. Lisäksi työssä tutustuttiin skaalautuvan ohjelmiston teoriaan. Kohdekulunvalvontajärjestelmän suorituskykyä mitattiin emuloimalla eli jäljittelemällä ovipäätteitä ja tarkkailemalla ovipäätteiden ja järjestelmän välisten tietoliikenteen vasteaikoja. Vasteajalle säädettiin järjestelmän käytettävyyden näkökulmasta raja, jonka yli mentäessä järjestelmän responsiivisuus sekä toimintavarmuus heikkenee. Prototyyppiohjelma toteutettiin käyttäen C++-kieltä ja ohjelma on terminaalipohjainen. Ohjelmassa hyödynnettiin Thread- sekä Asio-kirjastoja, jotka ovat osa Boost-kirjastokokoelmaa. Thread-kirjastoa käyttämällä testausohjelma säikeistettiin ja siitä saatiin suorituskykyinen. Jokaisen ovipäätteen emulointi suoritetaan omassa säikeessään ja säikeistämisen ansiosta ne tapahtuvat samanaikaisesti. Asio-kirjaston avulla hoidettiin kommunikointi emuloitavan ovipäätteen ja järjestelmän välillä käyttäen TCP-yhteyttä.

Työssä saatiin aikaiseksi prototyyppiohjelma, jonka avulla pystytään testaamaan työn kohdekulunvalvontajärjestelmän suorituskykyä sekä skaalautumista. Ohjelman avulla voidaan selvittää kulunvalvontajärjestelmän maksimikulunvalvontapisteiden määrää, jolla järjestelmä toimii responsiivisesti sekä toimintavarmasti. Testausohjelmaa voidaan käyttää osana kulunvalvontajärjestelmän ohjelmistokehittämistä. Ohjelman avulla voidaan varmistua jokaisen lähdekoodimuutoksen vaikutuksesta koko järjestelmän suorituskykyyn.

Prototyyppiohjelma suoriutui siihen suunnitellussa tehtävässään eli sitä voidaan käyttää kohdekulunvalvontajärjestelmän suorituskyvyn mittausvälineenä. Työn toimeksiantaja voi ennakoida prototyyppiohjelman avulla vaihtelevan kulunvalvontapisteiden määrän vaikutuksen järjestelmän toimintavarmuuteen sekä suorituskykyyn. Tämän ansiosta toimeksiantaja voi säästää palvelinkuluissa käyttämällä prototyyppiohjelmaa selvittääkseen järjestelmän komponenttien todelliset resurssitarpeet.

KAMK • University of Applied Sciences

Abstract

Author(s): Eemeli Kalliokoski

Title of the Publication: Design and Implementation of the Access Control System Scalability Testing Program

Degree Title: Bachelor of Engineering, Information and Communication Technology

Keywords: access control system, scaling

This thesis was commissioned by Fisplay Ltd, due to a need to measure the performance of their access control system. Designing and implementing a scaling testing software for access control system became the topic of this thesis. The goal of this thesis was to design and implement a prototype software that can measure the performance of the target access control system. The author of this thesis has previous experience in developing access control systems.

How access control systems are built and how do they work were explained in this thesis. How software scales was also explained. The performance of the target access control system was measured by emulating access points and observing the communication response times between the access points and the server. A limit was set for the response time considering the usability of the system. The system's responsiveness and reliability will suffer if the limit is crossed. The prototype software was made using C++ and it is terminal based. The software utilizes the Thread and Asio libraries which are part of the Boost library collection. By using the Thread library, the prototype software was threaded and because of that it is efficient. Every access point is emulated in their own threads at the same time. The Asio library was used to implement the TCP connection between the access points and the server.

The prototype software was achieved, and it can be used to test the performance and the scaling abilities of the target access control system. The maximum number of access points, in which case the system is still responsive and reliable, can be determined by using the prototype software. The prototype software can be used as a tool in the access control systems software development. The effects of every source code change to the performance of the target access control system can be ensured by using the prototype software.

The prototype software performed its task, and it can be used as a tool to measure the performance of the target access control system. The commissioner of this thesis can anticipate what effect a change in numbers of access points has on the target access control system's reliability and performance by using the prototype software. Thanks to this, the commissioner can save on server costs by using the prototype software to determine the true resource needs of the access control system's components.

Sisällys

1 Johdanto

Toimiva kulunvalvontajärjestelmä on loppukäyttäjälleen huomaamaton ja vaivaton. Käyttäjä voi kulkea kulunvalvontapisteen läpi ilman, että hänen tarvitsee tätä toimenpidettä noteerata. Epäluotettava tai epäkunnossa oleva kulunvalvontajärjestelmä vuorostaan aiheuttaa ylimääräistä huolta ja harmaita hiuksia käyttäjälleen.

Opinnäytetyön tilaajana toimii Fisplay Oy. Yritys tarjoaa erilaisia infonäyttöjärjestelmäratkaisuja erilaisiin tarpeisiin. Yritykseltä myös löytyy liiketoimintaa kulunvalvontajärjestelmä osa-alueelta, jota tämä opinnäytetyö käsittelee. Yrityksellä syntyi tarve mitata kulunvalvontajärjestelmänsä suorituskykyä, joten opinnäytetyöni aiheeksi muodostui kulunvalvontajärjestelmän skaalautumistestausohjelman suunnittelu ja toteutus.

Jokainen kasvava kulunvalvontajärjestelmä törmää ennemmin tai myöhemmin skaalautumisen aiheuttamiin kasvukipuihin. Järjestelmä voi oireilla epävarmana toimintakykynä tai jopa toimintakyvyttömyytenä. Esimerkiksi ylikuormittunut järjestelmä voi menettää reaaliaikaisen kulunvalvontaominaisuuden tai kulkuoikeuksien päivittäminen voi hidastua. Tämä alentaa suojeltavan kohteen turvallisuutta sekä järjestelmä menettää luotettavuuttansa.

Opinnäytetyön tarkoituksena on suunnitella ja toteuttaa prototyyppiohjelma, jonka avulla pystytään testaamaan sekä mittaamaan kulunvalvontajärjestelmän suorituskykyä. Tämän myötä ohjelmaa voidaan hyödyntää järjestelmän skaalautumisen tehostamiseksi sekä mittaamaan teoreettisten maksimikulunvalvontapisteiden määrään, jota järjestelmä kykenee käsittelemään. Ohjelman avulla voidaan myös varmistua jokaisen lähdekoodimuutoksen vaikutuksesta koko järjestelmän suorituskykyyn.

Ohjelman tavoitteena on tarjota yritykselle säästöjä palvelinkuluihin tehostamalla palvelinresurssien käyttämistä järjestelmän skaalautuessa. Ohjelman avulla voidaan myös ennakoida muuttuvan markkinatilanteen vaikutusta koko järjestelmän toimintakykyyn. Näin osataan valmistautua kulunvalvontajärjestelmän käyttäjämäärien muutoksiin, ennen kuin niiden aiheuttamat vaikutukset näkyvät ongelmina tai ylisuurina palvelinkuluina.

2 Kulunvalvontajärjestelmä

Kulunvalvontajärjestelmän tarkoituksena on rajoittaa luvattomilta pääsyn suojeltavaan kohteeseen. Tämä tapahtuu tunnistamalla henkilö tai ohjelma, joka pyrkii saamaan pääsyn suojeltavaan kohteeseen. Kulunvalvontajärjestelmät voidaan jakaa fyysiseen sekä loogiseen kulunvalvontaan. Fyysisessä kulunvalvonnassa suojellaan fyysisiä kohteita, kuten rakennustyömaita tai yritystiloja, kun taas loogisessa kulunvalvonnassa suojellaan esimerkiksi tietokonejärjestelmiä tai dataa. Sähköisellä kulunvalvonnalla on fyysisen kohteen suojeleminen toteutettu sähköisesti, esimerkiksi kortinlukulaitteiden avulla. [1.]

Henkilöillä, joille on myönnetty kulkuoikeus järjestelmähaltijan toimesta, on pääsy suojeltavaan kohteeseen. Myönnetty kulkuoikeus määrittelee kohteen lisäksi muun muassa sen voimassaoloajan, kenelle kulkuoikeus on myönnetty sekä miten henkilön täytyy tunnistautua kulunvalvontapisteellä. Yleisiä tunnistautumistapoja sähköisessä kulunvalvonnassa ovat RFID-kortti, PIN-koodi sekä biometriset tunnistautumistavat, kuten kasvo- tai sormenjälkitunnisteet [2].

Jokaisesta kulkutapahtumasta tallennetaan järjestelmään lokitieto. Lokitiedosta voi esimerkiksi käydä ilmi kulkupiste, kulkija, tunnistautumistapa ja kulkuaika. Näitä lokitietoja tarkkailemalla järjestelmänhaltija voi seurata reaaliajassa kulunvalvontapisteitä ja tarvittaessa ryhtyä toimiin reagointia vaativissa tilanteissa. Tällaisia tilanteita ovat esimerkiksi oven oleminen auki-tilassa liian pitkään, ovesta on yritetty toistuvasti kulkea ilman voimassa olevaa kulkuoikeutta tai ovi on hälytys-tilassa.

Yksinkertaisimmillaan tyypillinen sähköinen kulunvalvontajärjestelmäratkaisu koostuu kahdesta osasta: ovipäätteestä sekä palvelimesta. Yksittäinen ovipääte koostuu vähintään lukosta, tunnistautumislaitteesta sekä niitä hallinnoivasta järjestelmästä [2]. Palvelinratkaisuja on taas monenlaisia. Yksi tyypillinen ratkaisu on, että verkkopalvelimella on järjestelmän päälogiikkaa suorittava ohjelma. Palvelimelta myös yleensä löytyy tietokanta, johon tallennetaan kaikki lokitiedot sekä web-rajapinta käyttöliittymälle. Verkkoselaimen kautta päästään käsiksi käyttöliittymään ja voidaan tehdä muutoksia muun muassa kulkuoikeuksiin sekä tarkastella lokitietoja.

Kulunvalvontajärjestelmään voidaan liittää myös sekundäärisiä toiminnallisuuksia. Sekundäärisiksi toiminnallisuuksiksi voidaan katsoa kaikki ne järjestelmän lisätoiminnallisuudet, jotka eivät varsinaisesti avusta suoraan kohteen suojelemisessa kulunvalvonnalla. Esimerkiksi työajanseu-

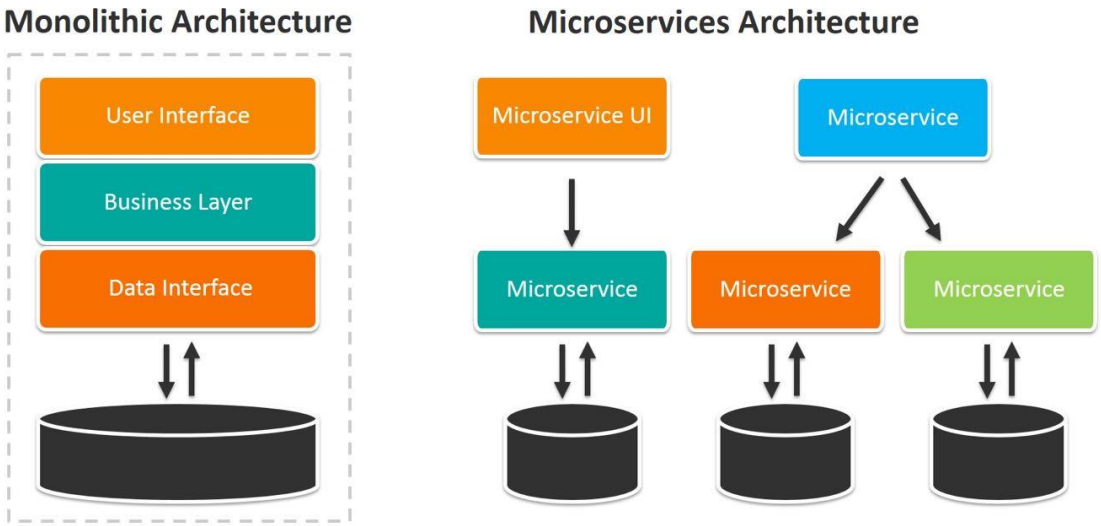
rantajärjestelmä usein liitetään osaksi kulunvalvontajärjestelmää. Kulunvalvontajärjestelmän lokitietoja voidaan myös hyödyntää palo- sekä henkilöturvallisuudessa. Lokitietojen perusteella voidaan luoda reaaliaikainen henkilökartta, jonka avulla voidaan kohdentaa sekä identifioida henkilöt jopa huonekohtaisesti. Lokitietoja voidaan lisäksi hyödyntää henkilöturvajärjestelmässä, jossa järjestelmä tarkkailee riskialttiissa tilassa olevaa henkilöä tai itse riskialtista henkilöä. Myös automaattiset paloilmoitinjärjestelmät, jotka ilmoittavat palosta suoraan palolaitokselle tai muulle taholle, ovat tyypillisiä kulunvalvontajärjestelmään lisättäviä paloturvallisuusratkaisuja.

3 Skaalautuva ohjelmisto

Ohjelman skaalautuvuus tarkoittaa ohjelman kykyä suoriutua kasvavasta työkuormasta. Lisäämällä ohjelmalle saatavilla olevia resursseja, kuten prosessointitehoa, voidaan parantaa ohjelman suorituskykyä ja tämä myötä sen skaalautumiskyky paranee. Ohjelman skaalautumiskyvyn arvioinnissa on hyvä ymmärtää, miten ohjelma kykenee toimimaan esimerkiksi kymmenellä tai miljoonalla käyttäjällä. Hyvin skaalautuvan ohjelman suorituskykyyn ei siis suoranaisesti vaikuta, montako käyttäjää sillä on. Se kykenee käyttäjämäärän lisääntyessä tarvittaessa saamaan lisää palvelinresursseja käytettäväkseen suorituskyvyn ylläpitämiseksi.

Monoliittinen ohjelma on nimensä mukaisesti yksi valtava ohjelmistokokonaisuus. Se koostuu yhdestä ohjelmainstanssista, joka on täysin itsenäinen ja toisista ohjelmista riippumaton. Muutokset ohjelman eri osiin vaativat koko ohjelman uudelleen kääntämisen ja ohjelman uudelleen käyttöönoton. [3.]

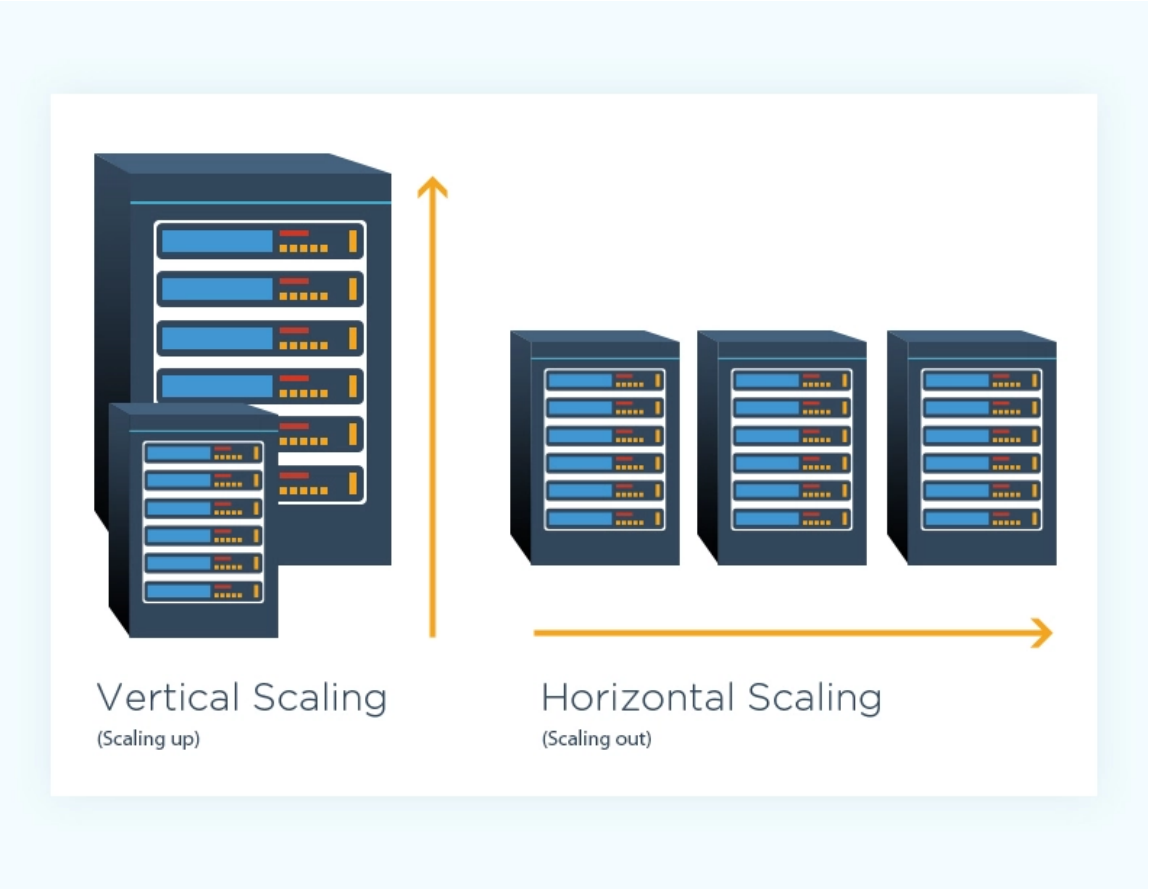
Mikropalveluarkkitehtuurinen ohjelma koostuu useasta itsenäisestä palvelusta. Yksittäiset palvelut ovat lähtökohtaisesti löyhästi kytketty toisiinsa eli ne ovat mahdollisimman vähän tietoisia sekä riippuvaisia toisistaan. Näihin yksittäisiin palveluihin voidaan tehdä lähdekoodimuutoksia ilman, että ne vaikuttavat ohjelman muihin palveluihin millään tavalla. Palvelut kommunikoivat keskenään ennalta määritettyjen ohjelmistorajapintojen kautta. Alla kuvassa 1 vasemmalla monoliittinen arkkitehtuuri, jossa ohjelman komponentit ovat niputettu yhdeksi ohjelmakokonaisuudeksi. Kuvassa oikealla on vuorostaan mikropalveluarkkitehtuuri, jossa ohjelman komponentit eli palvelut toimivat itsenäisesti ja kommunikoivat keskenään ohjelmistorajapintojen kautta.



Kuva 1. Monoliittisen - sekä mikropalveluarkkitehtuurin erot. [4]

3.1 Skaalautuvuus

Yksi monoliittisen ohjelman suurimpia haasteita on skaalautuvuus. Kyseisiä ohjelmia pystytään skaalaamaan joko horisontaalisesti lisäämällä uusia palvelimia tai vertikaalisesti lisäämällä palvelimen resursseja. Alla kuvassa 2 vasemmalla vertikaalinen sekä oikealla horisontaalinen skaalautuminen.



Kuva 2.Vertikaali- sekä horisontaalinen skaalautuminen. [5]

Molemmat skaalautumistavat ovat kuitenkin lähtökohtaisesti epätehokkaita, sillä ohjelman sisäisten komponenttien välillä on suorituskyvyissä eroja. Näistä eroista johtuvat pullonkaulat hidastavat ja rajoittavat koko ohjelman toimintaa. Pullonkaulat myös aiheuttavat paljon palvelinresurssien haaskaamista. Esimerkiksi yhden palvelun jarruttaessa koko ohjelmaa, joutuvat muut

palvelut odottamaan lähes toimettomina kyseisen palvelun toimintaa. Kaikki palvelut kuitenkin varaavat palvelinresursseja odottamisesta huolimatta.

Mikropalveluohjelma vuorostaan loistaa sen kyvyssä skaalautua. Kuten monoliittista ohjelmaa, voidaan mikropalveluohjelmaa skaalata sekä vertikaalisesti että horisontaalisesti. Erona monoliittisen ohjelman skaalautumiseen on, että ohjelman yksittäisiä palveluita pystytään skaalaamaan erikseen. Näin palveluiden välisiltä pullonkauloilta vältytään ja palvelinresursseja voidaan varata palveluiden todellisten käyttötarpeiden mukaan.

Palveluiden työkuormituksen kasvaessa voidaan hyödyntää automaatioskaalautumista. Tällöin määritellään skaalautumisen säännöt eli sen rajat, ehdot sekä palvelinresurssin käyttöaste, jota tarkkailemalla palvelin pystyy automaattisesti lisäämään tai vähentämään palvelulle käytettävissä olevien resurssien määrä. Esimerkiksi kulunvalvontajärjestelmän ovipäätteet ovat pääsääntöisesti päiväsaikaan enemmän aktiivisia kuin yöaikaan. Automaatioskaalautumisella voidaan määritellä horisontaalinen skaalautuminen tasaamaan vuorokausivaihtelun aiheuttaman työkuormituksen eroja.

Manuaalisella skaalautumisella taas ennakoidaan suurempia järjestelmän työkuorman muutoksia. Esimerkiksi ovipäätteiden määrän moninkertaistuessa voidaan ennakoida järjestelmän työkuormaa kasvattamalla manuaalisesti palvelimen suorituskykyä tai lisäämällä palvelimien oletusmäärää. Samalla automaatioskaalautumisen rajat tulee tarkistaa.

3.2 Skaalautumisen mittaaminen ja testaaminen

Monoliittisen ohjelman skaalautumisen testaaminen on hyvin suoraviivaista: pystytetään ohjelmasta yksi instanssi palvelimelle ja testataan sekä mitataan sen kykyä suoriutua tehtävässään. Tämän mittaustuloksen perusteella voidaan myös karkeasti arvioida, miten ohjelma skaalautuu horisontaalisesti. Skaalaamalla ohjelmaa vertikaalisesti ja mittaamalla sen vaikutus ohjelman suorituskykyyn, voidaan arvioida, miten koko ohjelmaa voidaan kaiken kaikkiaan skaalata.

Mikropalveluisen ohjelman kykyä skaalautua on vuorostaan hieman monimutkaisempaa mitata. Vaikkakin yksittäisiä palveluita voidaan lähtökohtaisesti skaalata, todellisuudessa ohjelman skaalautuessa vastaan tulee eri palveluiden aiheuttamia pullonkauloja. Tämän vuoksi joitain palveluita voidaan joutua skaalaamaan aggressiivisesti, kun taas jotkin palvelut eivät vaadi skaalausta juuri

lainkaan. Mikropalveluohjelman skaalauksen testaaminen onkin pitkälti eri palveluiden skaalaussuhteiden selvittämistä. Mittaamalla voidaan myös selvittää ohjelman palveluiden automaatio- sekä manuaaliskaalaussääntöjen teoreettiset käyttökuormarajat.

4 Testausohjelman määrittely

Opinnäytetyön tavoitteena on kehittää testausohjelman prototyyppi, jonka avulla voidaan testata kulunvalvontajärjestelmän suorituskykyä sekä skaalautumista. Ohjelman tarkoituksena on emuloida eli jäljitellä ovipäätteitä ja mitata tietoliikenteen vasteaikoja ovipäätteiden ja palvelimen välillä. Lisäämällä emuloitavien ovipäätteiden määrää voidaan stressitestata koko järjestelmää ja selvittää enimmäisovipäätteiden määrä, jolloin järjestelmä on toimintakykyinen sekä responsiivinen. Mikäli enimmäisovipäätteiden määrä ylitetään, ovat tietoliikenteen vasteajat liian suuria tai ne alkavat kumulatiivisesti kasvamaan ja järjestelmän toimintakyky heikkenee. Vasteajalla tarkoitetaan ovipäätteen lähettämän TCP-viestin ja siihen palvelimelta saadun vastauksen välistä aikaa.

Liian suuret vasteajat näkyvät koko järjestelmän toimintavarmuudessa. Vasteaikojen kasvaessa uusimpia ovipäätetietoja ja -tapahtumia ei saada päivitettyä palvelimelle ja järjestelmä käyttää tällöin jo vanhentuneita tietoja. Tämä voi näkyä esimerkiksi virheellisenä ovitilatietona (ovi auki/kiinni) sekä vanhentuneina lokitietoina järjestelmän käyttöliittymässä. Jos vasteajat kasvavat entisestään, vaarantuu koko järjestelmän toimintakyky sekä suojeltavien kohteiden turvallisuus. Käyttöliittymän tietoihin ei tällöin voi luottaa ja reaaliaikanainen kulunvalvontapisteiden seuranta on mahdotonta. Testausohjelman avulla voidaan ennaltaehkäistä tällaisia tilanteita syntymästä selvittämällä enimmäisovipäätemäärä, jonka ylittämisen jälkeen järjestelmä alkaa menettämään toimintakykyänsä. Tietämällä enimmäisovipäätemäärän voi järjestelmän haltija arvioida muuttuvan ovipäätteiden määrän vaikutusta koko järjestelmän toimintavarmuuteen ja välttyä järjestelmän joutumisen reagoimattomaan tilaan.

Testausohjelman avulla voidaan myös etsiä kulunvalvontajärjestelmäratkaisun pullonkauloja, jossa järjestelmän yksi komponentti hidastaa koko järjestelmää. Tämä voidaan toteuttaa esimerkiksi vertailuanalyysillä järjestelmän eri kehitysiteraatioiden välillä. Tekemällä muutoksia järjestelmän yksittäisiin komponentteihin ja testausohjelmaa käyttämällä voidaan poissulkemisperiaatteen avulla selvittää järjestelmän todelliset pullonkaulat. Samalla voidaan varmistua jokaisen lähdekoodimuutoksen vaikutuksesta koko järjestelmän suorituskykyyn.

Testausohjelma myös mahdollistaa järjestelmän eri palveluiden skaalautumissääntöjen määrittämisen sekä testaamisen. Lähtökohtaisesti palvelut skaalautuvat yhden resurssin, esimerkiksi prosessorin tai muistin käyttöasteen avulla. Testausohjelmalla voidaan löytää palveluiden todelliset

resurssitarpeet sekä niiden ankkuriresurssit, joiden perusteella ne ovat tehokkainta skaalata. Tasapainottamalla palveluiden automaatioskaalautuminen vältytään palvelinresurssien haaskaamiselta ja tämän myötä säästetään palvelinkuluissa. Testausohjelmalla voidaan myös selvittää voimassa olevien skaalautumissääntöjen mukainen teoreettinen enimmäisovipäätemäärä.

Opinnäytetyössä varsinaista testausohjelman käyttötarkoituskulunvalvontajärjestelmää käsitellään yleisenä sähköisenä kulunvalvontajärjestelmäratkaisuna. Järjestelmää käsitellään ainoastaan pintapuolin ja ainoastaan testaustyökalun kannalta välttämättömät tiedot tuodaan julki. Testattava kulunvalvontajärjestelmä tullaan pystyttämään paikallisesti testausympäristöksi.

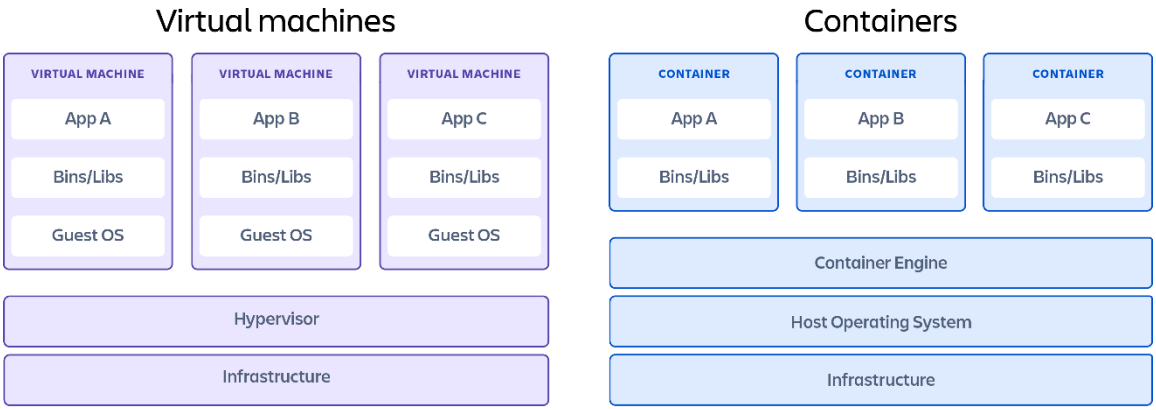
5 Käytettävät ohjelmistokielet ja käytänteet

5.1 Docker

Docker-kontit (engl. containers) tarjoavat sovelluskehittäjälle standardin kehyksen ajaa kehitettäviä ohjelmia. Perinteiset virtuaalikoneet emuloivat fyysistä tietokonetta, jossa jokaisella virtuaalikoneinstanssilla on oma käyttöjärjestelmänsä. Kontit vuorostaan eivät sisällä käyttöjärjestelmää, vaan ne tarjoavat käyttöjärjestelmätason virtualisointia jakamalla niiden alla olevan käyttöjärjestelmäytimen. [6.]

Jakamalla käyttöjärjestelmäytimen yksittäiset kontit vaativat huomattavasti vähemmän resursseja niiden isäntäjärjestelmältään [7]. Resurssien käyttäminen on myös huomattavasti joustavampaa konteilla kuin virtuaalikoneilla. Virtuaalikoneessa määritetään etukäteen sille sallitut resurssit, kuten käytettävien prosessoriytimien, muistin sekä kiintolevytilan määrän. Näiden määrien muokkaaminen vaatii virtuaalikoneen uudelleen konfigurointia ja uudelleen käynnistämisen. Virtuaalikoneen ollessa käynnissä nämä resurssit eivät ole isäntäjärjestelmän käytettävissä. Konteilla vuorostaan on lähtökohtaisesti rajattomat pääsyt isäntäkoneensa resursseihin. Tämän avulla yksittäinen kontti skaalautuu virtuaalikonetta joustavammin, sillä sen vaatiessa enemmän resursseja, esimerkiksi prosessoritehoa tai muistia, pystyy se vapaasti niitä isäntäkoneelta hyödyntämään. Tarvittaessa yksittäisille konteille voidaan säätää resurssien käyttörajat. Säätämällä konttien resursseille kiinteät käyttörajat voimme varmistua siitä, että opinnäytetyössä suoritettavat suorituskykytestit ovat toistettavia. Ilman resurssien kiinteää käyttörajaa isäntäkoneen vaihteleva saatavilla olevien resurssien määrä vaikuttaisi jokaiseen ajettavaan testiin ja testin tulokset olisivat epäluotettavia.

Jokainen kontti on toisistaan riippumaton, ja ne eivät ole lähtökohtaisesti tietoisia toisistaan. Yksittäinen kontti pitää sisällään vain ajettavan ohjelman ja sen tarvitsemat binäärit ja kirjastot. Tämän vuoksi yksittäisen kontin pystytys on huomattavasti virtuaalikonetta nopeampi. [8.] Seuraavassa kuvassa 3 on havainnollistettu, miten vasemmalla jokaisessa virtuaalikoneessa on oma käyttöjärjestelmänsä (engl. OS), kun vuorostaan oikealla konteissa on ainoastaan ajettava ohjelma (engl. app) ja sen tarvitsemat binäärit (kuvassa bins) sekä kirjastot (kuvassa libs).



Kuva 3. Virtuaalikoneen sekä kontin eroavaisuudet. [9]

Yksi pääsyy siihen, että kontit ovat niin tehokkaita ohjelmistokehityksessä, on niiden kyky kommunikoida keskenään [10]. Konteille voidaan määritellä avoimia isäntäkäyttöjärjestelmän portteja, joista kontit ohjaavat liikenteen kontin sisälle sitä tarvitsevalle palvelulle. Tämän vuoksi kontit ovat erinomainen ratkaisu pystyttää testiympäristössä mikropalveluarkkitehtuurisia ohjelmia, joissa ohjelmakokonaisuus muodostuu pienistä itsenäisistä palveluista. Opinnäytetyössä testattava kulunvalvontajärjestelmä edustaa mikropalveluarkkitehtuurista ohjelmaa, joten kontit soveltuvat hyvin järjestelmän pystytykseen skaalautumistestaamista varten.

5.2 TCP/IP

TCP eli ”Transmission Control Protocol” on tietoliikenneprotokollan kuljetuskerros ja sitä käytetään tietokoneiden väliseen kommunikointiin. IP eli ”Internet Protocol” on vuorostaan tietoliikenneprotokollan verkkokerros, joka vastaa TCP-paketin reitittämisestä oikeaan osoitteeseen. [11.] Näiden kerrosten yhdistelmästä käytetään lyhennettä TCP/IP.

TCP-yhteyden luominen alkaa ns. kolmisuuntaisella kädenpuristuksella. Prosessin tarkoituksena on varmistua siitä, että molemmat tahot ovat valmiina aloittamaan tiedonsiirron. [12, s286.] Tiedonsiirron alettua TCP huolehtii, että molemmat tahot vastaanottavat kaikki lähetetyt paketit. Molemmat tahot kuittaavat vastaanotetut viestit ja paketin hukkuessa myös kuittaus jää uupumaan. Tällöin TCP uudelleen lähettää hukkuneen paketin. [12, s290.] Juuri virhetilannepalautumiskyvyn vuoksi TCP on hyvin varma kommunikointitapa.

Opinnäytetyössä testattava kulunvalvontajärjestelmä sekä ovipäätteet kommunikoivat keskenään TCP/IP-rajapinnan kautta. Viestintä tapahtuu TCP-paketeilla ja ovipääte avaa TCP-yhteyden.

5.3 C++

C++ on korkean tason olio-ohjelmointiin suuntautunut ohjelmointikieli. Kieli sai alkunsa professori Bjarne Stroustrupin toimesta, joka kehitti kielen C-ohjelmointikielen pohjalta. Kieli eroaa alkuperäisestä C-ohjelmointikielestä muun muassa sen luokkarakennemallillaan, vahvalla muuttujien tyypityksellään sekä laajoilla muistinhallintavälineillään. [13.] C++ soveltuu moneen eri käyttötarkoitukseen ja se tunnetaan tehokkaana yleiskielenä.

Kieli soveltuu erinomaisesti TCP-sokettiohjelmointiin eli TCP-yhteyksien luomiseen tietokoneiden välille. C++ on erittäin lähellä konekieltä ja onkin tämän vuoksi erittäin suorituskykyinen kieli. Opinnäytetyön tekijällä on myös aikaisempaa kokemusta C++-kielen käytöstä sokettiohjelmoinnissa, joten se on erinomainen valinta testausohjelman kieleksi.

TCP-yhteyksien hallintaan testausohjelma tulee hyödyntämään C++-kielen kykyä hyödyntää säikeitä ohjelman ajossa. Testausohjelmassa tulee jokaisesta TCP-yhteydestä huolehtimaan oma säikeensä, jonka ansiosta voidaan samanaikaisesti emuloida useita ovipäätteitä samanaikaisesti rinnakkain. Säikeistäminen toteutetaan käyttäen Boost-kirjastosta löytyvää Thread-alikirjastoa ja sokettiohjelmointiin käytetään Boost-kirjastosta löytyvää Asio-alikirjastoa. Boost on joukko C++-kirjastoja, jotka tarjoavat ratkaisuja muun muassa yksikkötestaukseen, näennäissatunnaislukugenerointiin, sokettiohjelmointiin ja edellä mainittuun säikeistämiseen.

Testausohjelman kääntämisen automatisoinnissa tullaan hyödyntämään CMake-ohjelmaa. Ohjelman ansiosta testausohjelman käyttämien kirjastojen linkittäminen sekä hallinta on helppoa. Testausohjelman hierarkkinen tiedostorakenne ei tuota ongelmia CMake-ohjelmalle ja testausohjelman eri iteraatioiden kääntäminen ajettavaksi binääriksi tapahtuu käden käänteessä.

6 Testausohjelma

6.1 Testaustapa

Testausohjelman tarkoituksena on emuloida ovipäätteitä, jotka luovat samanlaista tietoliikennettä kuin todelliset ovipäätteet. Emuloitavat ovipäätteet muodostavat palvelimeen yhteyden ja lähettävät erilaisia tapahtumia TCP-viesteinä, kuten ovipäätteen käynnistyminen, kulkijan identifioiminen, oven aukeaminen ja oven sulkeutuminen.

Itse kulunvalvontajärjestelmää testataan musta laatikko -testausperiaatteella (engl. Black-box testing). Testausperiaatteen idea on käsitellä testattavaa kohdetta mustana laatikkona ilman, että sen toimintaperiaatetta tarvitsee tuntea. Tarvitsee ainoastaan ymmärtää sekä tarkkailla laatikon syötettä (engl. input) sekä tulostetta (engl. output). [14.] Testattavan järjestelmän tapauksessa syötteenä toimii emuloitavilta ovipäätteiltä tuleva tietoliikenne ja tulosteena järjestelmän kuittaus ovipäätteiden viesteihin.

Testausohjelma pyrkii löytämään järjestelmän enimmäisovipäätemäärän, jolla koko kulunvalvontajärjestelmä on toimintakykyinen. Emuloitavan ovipäätteen näkökulmasta määrän ylittäminen tarkoittaa, että yksittäisen ovipäätteen lähettämän viestin edestakainen vasteaika on yli ennalta määritetyn rajan. Raja on määritetty järjestelmän käytettävyyden näkökulmasta. Esimerkiksi, mikäli järjestelmän käyttöliittymästä katsotut kulkutiedot, eli ovipäätteiltä saatavat lokitiedot, eivät päivity tarpeeksi nopeasti, ei järjestelmää voida käyttää reaaliaikaiseen kulunvalvontaan. Palvelimen näkökulmasta taas määrän ylittäminen tarkoittaa, että järjestelmä vastaa ovipäätteiden viesteihin liian hitaasti tai se vastaanottaa ovipäätteiltä enemmän viestejä kuin ehtii niitä käsittelemään. Tällöin viestit alkavat kasaantumaan ja vasteajat kasvavat entisestään.

Testausohjelma aloittaa ajonsa emuloimalla yhtä ovipäätettä. Emuloitu ovipääte lähettää tasaisesti eri tapahtumista TCP-paketteja palvelimelle ja vastaa palvelimen lähettämiin viesteihin. Seuraavaksi seurataan, ylittääkö viestien vasteajat sallitun rajan heti tai alkavatko vasteajat kasvamaan ajan kuluessa ilman, että ovipäätteitä lisätään. Esimerkiksi hitaasti kasvavien vasteaikojen aiheuttamat ongelmat voivat tulla esiin vasta jopa päivien päästä ovipäätteiden lisäämisten jälkeen. Emuloitavien ovipäätteiden määrää aletaan inkrementaalisesti kasvattamaan ja aina jokaisen ovipäätelisäyksen jälkeen tarkkaillaan vasteaikoja hetki. Emuloitavien ovipäätteiden lisäämistä sekä vasteaikojen tarkkailua jatketaan, kunnes jonkin emuloitavan ovipäätteen lähettämien

viestien vasteajat alkavat kasvamaan tai ne ylittävät sallitun rajan. Tällöin on jo ylitetty enimmäisovipäätteiden määrä ja testausohjelma on suorittanut tehtävänsä ja se lopettaa testin.

6.2 Toteutus

6.2.1 Pääsilmukka

Testausohjelma toteutettiin terminaalipohjaisena C++-ohjelmana, jota käännettiin käyttäen CMake-käännösautomatisointityökalua. Ohjelma aloittaa ajonsa kysymällä käyttäjältään, monellako ovipäätteellä emulointi aloitetaan. Tämän avulla säästetään aikaa, sillä emulointia ei tarvitse aina aloittaa yhdestä ovipäätteestä, vaan se voidaan aloittaa suoraan läheltä aikaisemmin tunnettua enimmäisovipäätemäärää. Seuraavaksi ohjelman pääsilmukassa luodaan annettujen ovipäätteiden määrän verran säikeitä. Jokaisen ovipäätteen emulointi tapahtuu siis omissa säikeissään. Alla kuvassa 4 säikeiden luonti on toteutettu käyttäen for-silmukkaa.

```
std::vector<boost::thread> threads;
for (int i = 1;; i++)
{
  std::cout << "Started thread number:" << i << std::endl;
  threads.push_back(boost::thread(emulateDevice, i));
  if (i >= startDeviceAmount)
  {
    sleep(10);
  }
  if (hasMaximumResponseTimePassed())
  {
    break;
  }
}
```

Kuva 4. Säikeiden luonti.

Kuvassa säikeitä luodaan silmukassa niin paljon, että on saavutettu annettu testauksen aloittamisen ovipäätemäärä. Tämän jälkeen silmukassa lisätään säie ja odotetaan kymmenen sekuntia sleep-funktion avulla. Tätä silmukkaa jatketaan, kunnes hasMaximumResponseTimePassed-funktio palauttaa arvon tosi. Funktio tarkistaa, onko jokin säikeissä emuloitavan ovipäätteen viestin

vasteaika ylittänyt sallitun rajan. Kun vasteajan sallittu raja on ylitetty, lopettavat säikeet ovipäätteiden emuloinnin sulkemalla TCP-yhteyden. Säikeet lopettavat tämän jälkeen toimintansa ja pääsilmukka tulostaa terminaaliin säikeiden määrän juuri ennen kuin vasta-ajat ylittyivät.

6.2.2 Säikeet

Säikeissä luodaan ovipääteinstanssi, jossa tarvittavat arvot, kuten ovipäätteen sekä protokollan versio, asetetaan vastaamaan aitoja ovipäätteitä. Vastaavanlaiset ovipääteinstanssit ajetaan komennolla paikallisen testausympäristön kulunvalvontajärjestelmän tietokantaan. Tämän ansioista emuloitavat ovipäätteet löytyvät järjestelmästä. Säikeet jatkavat toimintaansa luomalla TCP-yhteyden kulunvalvontajärjestelmän palvelimeen. Alla kuvassa 5 TCP-soketin avaaminen käyttäen Boost-kirjastosta löytyvää Asio-alikirjastoa.

```
try
{
  boost::asio::ip::tcp::endpoint ep(boost::asio::ip::address::from_string(m_ip),
                                    m_port);
  m_socket = std::make_unique<boost::asio::ip::tcp::socket>(m_service, ep.protocol());

  boost::system::error_code ignored_error;
  m_socket->connect(ep);
  return true;
}
catch (boost::system::system_error &e)
{
  std::cout << "Error occured while trying to create TCP-connection. Error code:" << e.code()
            << ". Message: " << e.what();
  return false;
}
```

Kuva 5. TCP-yhteyden luonti.

Säie pyrkii avaamaan TCP-yhteyden palvelimelle, mutta virhetilanteen sattuessa se tulostaa virheviestin terminaaliin ja lopettaa toimintansa. Onnistuneen TCP-yhteyden luonnin jälkeen aloitetaan varsinainen ovipäätteiden emulointi. Palvelin tarkistaa, että TCP-yhteyden avannut ovipääte löytyy tietokannasta ja suorittaa ovipäätteen käynnistykseen kuuluvan synkronoinnin. Mikäli ovipääte ei löydy järjestelmästä, palvelimen katkaisee TCP-yhteyden välittömästi. Synkronoinnissa tarkistetaan muun muassa, että ovipäätteen kello on oikeassa sekä siirretään ovipäätteellä syntyneet lokitiedostot palvelimelle. Synkronoinnin ajan emuloitava ovipääte toimii kuuntelutilassa eli se odottaa palvelimelta TCP-paketteja ja vastaa niihin todellisen ovipäätteen tapaan. Kun ovipäätteen ”käynnistys” on suoritettu, asetetaan se aktiivitilaan.

Aktiivitilassa ovipääte on täysin toimintavalmis ja se alkaa lähettämään palvelimelle erilaisia tapahtumia TCP-paketeilla. Tapahtumat vastaavat normaalista ovipäätteen toiminnasta syntyviä tapahtumia, kuten käyttäjän identifiointi, oven aukeaminen ja oven sulkeminen. Tapahtumia lähetetään keskimäärin saman verran, kuin oikea ovipääte lähettää ruuhka-aikana.

Jos palvelimen vastausten vasteajat lähetettyihin TCP-paketteihin eivät ylitä ennalta määritettyä rajaa eivätkä vasteajat kasva, voidaan pääsilmukassa lisätä yksi säie lisää. Jokaisen säielisäyksen jälkeen pääsilmukka odottaa kymmenen sekuntia ja jatkaa säikeiden lisäämistä, kunnes jossain säikeessä palvelimen vastauksen vasteaika ylittää sallitun rajan. Tämän sattuessa tallennetaan muuttujaan säikeiden määrä eli emuloitavien ovipäätteiden määrä. Säikeet lopettavat ovipäätteiden emuloinnin sekä toimintansa ja pääsilmukka tulostaa terminaaliin edellä mainitun ovipäätemäärän, josta on vähennetty yksi. Tämä määrä oli siis kulunvalvontajärjestelmän se enimmäisovipäätemäärä, jolla järjestelmä oli vielä toimintakykyinen. Tämän jälkeen testausohjelma lopettaa ajonsa.

7 Pohdinta ja yhteenveto

Opinnäytetyössä käytiin läpi kulunvalvontajärjestelmän yleinen toimintamalli, miten ne skaalautuvat sekä miten niiden suorituskykyä ja skaalautumista pystytään mittaamaan. Opinnäytetyössä toteutettiin testausohjelman prototyyppi, jonka avulla voidaan selvittää kulunvalvontajärjestelmän teoreettinen enimmäisovipäätemäärä, jolla järjestelmän katsotaan olevan toimintakykyinen. Testausohjelman avulla voidaan siis ennakoida muuttuvan markkinatilanteen vaikutusta koko järjestelmään. Aihetta tarjottiin syntyneen tarpeen vuoksi Fisplay Oy:n toimesta.

Testausohjelman toteuttaminen osoittautui opinnäytetyön aikana oletettua haastavammaksi. Tämä osittain johtui kulunvalvontajärjestelmän mikropalveluarkkitehtuurisesta luonteesta. Kulunvalvontajärjestelmän pystyttäminen testiympäristössä vaati opinnäytetyön tekijältä paneutumista syvällisesti järjestelmän eri palveluiden toimintamalleihin sekä niiden päälogiikkaan. Yksittäiset palvelut vaativat paljon konfigurointia, jotta ne saatiin pystytettyä sekä kommunikoimaan keskenään paikallisessa testiympäristössä.

Haasteista huolimatta kulunvalvontajärjestelmän skaalautumisen testauksen prototyyppiohjelma saatiin aikaiseksi. Testausohjelmalla voidaan selvittää käyttökohdejärjestelmän enimmäisovipäätemäärä ja tämän myötä sitä voidaan käyttää järjestelmän suorituskyvyn mittaustyökaluna. Testausohjelmalla ajettavat testit ovat toistettavia pienellä virhemarginaalilla. Testausohjelman avulla voidaan myös selvittää palveluiden resurssitarpeita sekä säätää niiden automaatio- sekä manuaaliskaalautumisen sääntöjä.

Testausohjelmaa on mahdollista jatkokehittää monella eri tapaa. Koska kyseessä oli prototyyppiohjelma, oli sen tärkein ominaisuus todistaa, että ovipäätteitä emuloimalla voidaan mitata kulunvalvontajärjestelmän suorituskykyä. Prototyyppiluonteensa vuoksi testausohjelma on yksinkertainen ja sen toteutuksessa on keskitytty sen ydintoiminnallisuuteen. Testausohjelmaa voitaisiin kehittää parantamalla testausohjelman käytettävyyttä sekä laajentamalla ohjelman testille annettavia parametreja. Esimerkiksi testausohjelmaa voisi olla mahdollista käyttää stressitestitilassa, jolloin ohjelma jatkaisi ovipäätteiden emulointia, vaikka enimmäisovipäätemäärä olisi saavutettu. Myös testi, miten palvelin palautuu ylikuormitustilasta, olisi hyvin hyödyllinen. Ohjelmaan voitaisiin myös lisätä mahdollisuus suorittaa testi kustomoiduilla ovipäätteillä. Tämän avulla voidaan esimerkiksi testata, miten järjestelmä käyttäytyy, kun toimivien ovipäätteiden sijasta järjestelmää testataan virheellisillä ovipäätteillä tai jopa virheellisillä tapahtumaviesteillä. Ovipäätteiden kustomointi voitaisiin toteuttaa reaaliaikaisesti, jolloin käyttäjän olisi mahdollista

testin aikana lisätä haluamansa laisia ovipäätteitä, kuten virhetilassa olevia, rikkinäisiä tai ruuhkautuneita ovipäätteitä. Tämän avulla voitaisiin nähdä, miten erilaiset ovipäätteet rasittaisivat järjestelmää. Testausohjelma myös kaipaisi lokitietojen tallentamista sen sijaan, että syntyneet tulokset sekä virheet tulostettaisiin terminaaliin. Testin aikana syntyneiden virheiden, poikkeusten sekä testin tuloksen tallentaminen tiedostomuotoon helpottaisi testin ajonjälkeistä testien vertailua sekä ongelmien korjausta.

Kaiken kaikkiaan testausohjelma oli onnistunut ja se suoriutuu siihen suunnitellussa tehtävässään. Ohjelmaa voidaan hyödyntää Fisplay Oy:n kulunvalvontajärjestelmän kehityksessä ja sen avulla voidaan ennakoida muuttuvan markkinantilanteen vaikutusta kulunvalvontajärjestelmän toimintakykyyn. Ohjelman avulla voidaan myös optimoida palvelinresurssien käyttöä ja se tarjoaa tämän avulla säästöjä palvelinkuluissa.

Lähteet

1 Lutkevich Ben. access control; 2022 [Viitattu 1.2.2023]. Saatavilla: https://www.techtarget.com/searchsecurity/definition/access-control

2 Palter Jay. Physical Access Control: How It Works In 2023; 2023 [Viitattu 4.5.2023]. Saatavilla: https://www.realtimenetworks.com/blog/physical-access-control-how-it-works

3 Harris Chandler. Microservices vs. monolithic architecture; [Viitattu 24.3.2023]. Saatavilla: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

4 Johnson Jonathan. Shiff Laura. What Is Microservice Architecture? Microservices Explained; 2021 [Viitattu 24.3.2023]. Saatavilla: https://www.bmc.com/blogs/microservices-architecture/

5 CloudZero. Horizontal Vs. Vertical Scaling: How Do They Compare?; 2021 [Viitattu 24.3.2023]. Saatavilla: https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling

6 Kasireddy Preethi. A Beginner-Friendly Introduction to Containers, VMs and Docker; 2016 [Viitattu 28.3.2023]. Saatavilla https://www.freecodecamp.org/news/a-beginner-friendly-introduction-to-containers-vms-and-docker-79a9e3e119b/

7 Hines Chris. The 10 Most Common Questions IT Admins ask About Docker; 2016 [Viitattu 28.3.2023]. Saatavilla: https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/

8 Tunggal Abi Tyas. Docker vs VMWare: How Do They Stack Up?; 2022 [Viitattu 28.3.2023]. Saatavilla: https://www.upguard.com/blog/docker-vs-vmware-how-do-they-stack-up

9 Buchanan Ian. Containers vs. virtual machines; [Viitattu 28.3.2023]. Saatavilla: https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

10 Docker docs. Networking overview; [Viitattu 28.3.2023]. Saatavilla: https://docs.docker.com/network/

11 TCP/IP protocols; 2023 [Viitattu 30.3.2023]. Saatavilla: https://www.ibm.com/docs/en/aix/7.2?topic=protocol-tcpip-protocols

12 Goralski Walter. The Illustrated Network; 2009

13 Rouse Margaret. C plus plus Programming Language; 2021 [Viitattu 31.3.2023]. Saatavilla: https://www.techopedia.com/definition/26184/c-plus-plus-programming-language

14 Hamilton Thomas. Black Box Testing; 2023 [Viitattu 1.4.2023]. Saatavilla: https://www.guru99.com/black-box-testing.html