TIETO- JA SÄHKÖTEKNIIKAN TIEDEKUNTA

Jante Jompa

SUORITUSKYKYINEN PALVELINOHJELMISTO REAALIAIKAISELLE WEB-SOVELLUKSELLE

Diplomityö
Tietotekniikan tutkinto-ohjelma
Toukokuu 2023

Jompa J. (2023) Suorituskykyinen palvelinohjelmisto reaaliaikaiselle web-sovellukselle. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 67 s.

TIIVISTELMÄ

Internetin yleistymisen myötä selainten ominaisuudet ovat laajentuneet, ja nykyään web-teknologiat mahdollistavat alustariippumattoman ja kustannustehokkaan sovelluskehityksen. Interaktiiviset web-sovellukset luovat tarpeen palvelinohjelmistolle, joka kykenee tarjoamaan reaaliaikaisia resursseja suurelle määrälle samanaikaisia käyttäjiä.

Tässä työssä suunnitellaan ja toteutetaan skaalautuva palvelinjärjestelmä reaaliaikaiselle web-sovellukselle. Järjestelmän skaalauksen tarkoituksena on saavuttaa kyky palvella mahdollisimman suurta määrää yhtäaikaisia käyttäjiä. Käyttäjien tulee pystyä olemaan keskenään interaktiivisessa kanssakäymisessä jaettujen resurssien kautta.

Järjestelmän suunnittelufilosofiana on, että tapahtumat välittyvät järjestelmästä niistä kiinnostuneille tahoille ilman, että mikään järjestelmän osa hakee minkään resurssin tilaa toistuvasti uudestaan. Sen sijaan resurssien muutokset ja muut viestit lähetetään suoraan niistä kiinnostuneille tahoille heti, kun ne tapahtuvat. Järjestelmän arkkitehtuurin vaatimuksiksi muodostuvat reaaliaikaisuus, interaktiivisuus, skaalautuvuus ja suorituskyky.

Työssä käsitellään vaatimusten vaikutuksia järjestelmän arkkitehtuuriin ja sitä käyttävien ohjelmistojen suunnitteluun. Lisäksi arvioidaan järjestelmän soveltuvuutta erilaisten sovellusten käyttöön sekä sen arkkitehtuurin heikkouksia. Myös erilaisten ohjelmistokomponenttien skaalaaminen, suunnittelun kannalta olennaiset tekniikat sekä järjestelmän toteutuksessa käytetyt teknologiat käydään läpi.

Järjestelmän arkkitehtuurivaatimusten täyttymisen arvioimiseksi toteutetaan esimerkkisovelluksen suorittamiseen käytettävä järjestelmä, jota testataan simuloimalla suuria määriä sovellusta käyttäviä käyttäjiä. Testeissä mitataan järjestelmän suorituskykyä ja skaalautuvuutta, ja tulosten avulla voidaan arvioida, miten skaalauksen kasvattamisella voidaan parantaa järjestelmän suorituskykyä. Tulosten avulla arvioidaan myös järjestelmän suunnittelun vaatimusten täyttymistä ja skaalauksen toimivuutta käytännössä. Lisäksi käydään läpi mahdollisia tapoja parantaa järjestelmän suorituskykyä.

Tulokset olivat hyviä. Testien aikana järjestelmä pystyi käsittelemään parhaimmillaan lähes 5000 viestiä sekunnissa yli tuhannelta samanaikaiselta simuloidulta käyttäjältä. Tulosten perusteella järjestelmän suorituskykyvaatimuksen voidaan katsoa täyttyvän. Myös interaktiivisuus- ja skaalautumisvaatimusten katsotaan täyttyvän, sillä interaktiivisia toimintoja sisältävän esimerkkisovelluksen suorituskykyä voitiin merkitävästi parantaa muuttamalla järjestelmän osien skaalausta. Työn lopuksi käydään läpi järjestelmän jatkokehitysmahdollisuuksia pilvipalveluun siirtymisen osalta.

Avainsanat: web-sovellus, palvelinohjelmisto, hajautettu järjestelmä, reaaliaikaisuus, horisontaalinen skaalautuvuus

Jompa J. (2023) High-Performance Server Software for Real-Time Web Application. University of Oulu, Degree Programme in Computer Science and Engineering, 67 p.

ABSTRACT

With the spread of the Internet, the capabilities of browsers have expanded, and today's web technologies enable platform-independent and cost-effective application development. Interactive web applications create a need for server software, capable of providing real-time resources to a large number of concurrent users.

In this thesis, a scalable server system is designed and implemented for a real-time web application. The purpose of scaling is to achieve the ability to serve maximum number of simultaneous users. Users must be able to interact with each other through common shared resources.

The design philosophy of the system is, that events are transmitted from the system to interested parties without any part of the system repeatedly fetching the state of any resource. Instead, resource changes and other messages are sent directly to interested parties immediately, when they happen. The requirements for the system's architecture are real-time, interactivity, scalability and performance.

The thesis discusses the effects of requirements on the system's architecture and on to the design of the software that uses it. In addition, the system's suitability for different applications is evaluated as well as the weaknesses of its architecture. The scaling of different software components, technologies relevant to design and the technologies used in the system's implementation are also discussed.

In order to evaluate the architectural requirements of the system, a test system is implemented to execute an example application. The system is tested by simulating large numbers of users using the example application. The tests measure system performance and scalability, and the results are used to assess, how increasing scaling can improve system performance. The results are also used to evaluate the fulfillment of the system design requirements and functionality of scaling in practice. In addition, possible ways to improve system performance are reviewed.

The results were good. The system could handle almost 5000 messages per second with over a thousand simultaneous simulated users. The performance requirement of the system can therefore be considered met. Interactivity and scalability requirements are also considered met, as the performance of the example application containing interactive functions could be significantly improved by changing the scaling of system components. At the end of the thesis, future work is discussed in terms of moving the system to a cloud platform.

Keywords: web application, server software, distributed system, real-time, horizontal scalability

SISÄLLYSLUETTELO

ALKULAUSE

Tässä työssä käsiteltävä palvelinohjelmisto on osa vapaa-ajan peliprojektiani, jonka suunnittelu ja kehitys on alkanut jo vuonna 2019. Pelin luonne ja sen mekaniikka määrittelevät pitkälti palvelinohjelmiston vaatimukset ja niiden kautta järjestelmän rakenteen ja toiminnan periaatteet. Toisaalta olen ollut pitkään kiinnostunut tämänkaltaisista järjestelmistä, joten peli osin suunniteltiin vaatimaan tällainen palvelinjärjestelmä. Halusin kuitenkin jättää pelin yksityiskohdat työn ulkopuolelle ja keskittyä itse järjestelmän suunnitteluun. Tavoitteena on suunnitella järjestelmän yleinen arkkitehtuuri niin, että se soveltuisi pelin kaltaisille sovelluksille riippumatta niinkään sovelluksen vaatimuksista tarkemmin. Halusin myös käyttää kotona lojuvia palvelintietokoneita järjestelmän testaamisessa.

Diplomityön tekeminen oli hidasta muun elämän kiireiden vuoksi. Tekemistä hidasti myös se, että työ tehtiin itsenäisesti omasta aiheesta, sen sijaan, että se olisi tehty työajalla yritykselle. Oma työni viime vuosina on liittynyt vastaavanlaisen järjestelmän kehittämiseen, joten diplomityötä varten tehty aiheeseen perehtyminen on ollut erityisen hyödyllistä. Haluan kiittää kaikkia työn tekemisessä minua auttaneita. Erityiset kiitokset ansaitsee vaimoni, sekä ohjaajani Lauri Tuovinen ja tekninen ohjaajani Mika Oja.

Oulussa 31. toukokuuta 2023

Jante Jompa

LYHENTEIDEN JA MERKKIEN SELITYKSET

API	*Application Programming Interface*
AWS	*Amazon Web Services*
CDN	*Content Delivery Network*
CPU	*Central Processing Unit*
CRUD	*Create, Read, Update, Delete*
GPS	*Global Positioning System*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
MVC	*Model-View-Controller*
NAT	*Network Address Translation*
ORM	*Object-Relational mapping*
PWA	*Progressive Web Application*
RAM	*Random Access Memory*
REST	*Representational State Transfer*
SLA	*Service-Level Agreement*
STUN	*Session Traversal Utilities for NAT*
SQL	*Structured Query Language*
SSD	*Solid State Drive*
SSE	*Server Sent Events*
TCP	*Transmission Control Protocol*
TURN	*Traversal Using Relays around NAT*
UDP	*User Datagram Protocol*
W3C	*World Wide Web Consortium*
WWW	*World Wide Web*
YAML	*YAML Ain't Markup Language*

1. JOHDANTO

Internetin yleistymisen myötä yleistyi hajautetun tietojenkäsittelyn muoto, jossa ohjelman suoritus jaetaan palvelimen ja asiakasohjelman kesken. Tällaisessa asiakasohjelma-palvelin -mallissa (engl. *Client-server model*) oli sovellusten toiminnan kannalta erityisen hyödyllistä mahdollisuus jakaa suorituksen logiikkaa, kuormitusta sekä datan tallentamista palvelimen ja asiakasohjelman välillä. Toisaalta useat sovellukset suunniteltiin käyttämään internetiä näiden nimenomaisten etujen vuoksi ja internet tuskin olisi yhtä merkittävässä asemassa nykyisin ilman niitä. [1][2]

Logiikan ja etenkin datan tallentamisen siirtäminen mahdollisimman suurelta osin palvelimelle teki kuluttajien päätelaitteille asennettavista asiakasohjelmista yksinkertaisia ja helppoja ottaa käyttöön. Kuluttajakäytössä asiakasohjelma-palvelin-jaottelu kuitenkin näyttäytyi erityisen selvästi vasta kun hajautettu hypertekstijärjestelmä World Wide Web (WWW) sekä hypertekstiä visualisoivat selaimet yleistyivät. Selain mullisti internet-sovellusten helpon käytön, sillä asiakasohjelmaa ei tarvinnut enää erikseen asentaa laitteelle, vaan ohjelma toimi käytännössä kokonaan palvelimella. Käyttäjän tarvitsi vain siirtyä selainta käyttäen haluamansa resurssin verkko-osoitteeseen. Myöhemmin web-sovellusten monimutkaistuessa tarvittava asiakasohjelman logiikka siirrettiin internetin yli suoraan selaimeen suoritettavaksi. [3]

Selaimet pystyvät esittämään tekstin lisäksi grafiikkaa ja monentyyppistä mediaa sekä vastaanottamaan käyttäjän syötettä monin tavoin. Selaimissa suoritettavan skriptimuotoisen ohjelmakoodin avulla voidaan toteuttaa hyvin kehittyneitä ja interaktiivisia alustariippumattomia ohjelmia eli web-sovelluksia. Sovellusten suorittaminen selaimessa ei vaadi niiden asentamista tai konfigurointia, vaan sovellukset voidaan toteuttaa niin, että ne ovat käytettävissä heti vain siirtymällä selaimella tiettyyn verkko-osoitteeseen. [3]

Selaimista alkunsa saaneet web-teknologiat eivät kuitenkaan rajoitu selaimiin, vaan web-sovellusten suosio on tuonut web-teknologiat myös muun muassa työpöytä- ja mobiilisovelluksiin. Samojen teknologioiden käyttö lukuisilla alustoilla nopeuttaa sovelluskehitystä ja vähentää kustannuksia. Web-sovellusten käytön helppous auttaa sovellusten tuontia markkinoille sekä nopeuttaa niiden levitystä. [3][4]

1.1. Työn tarkoitus

Tämä työ on jossain määrin jatkoa vuoden 2020 kandidaatintyölleni *Skaalautuva palvelinarkkitehtuuri reaaliaikaiselle selainpelille* [5]. Kandidaatintyöhön verrattuna tämä työ käsittelee palvelinjärjestelmän toteutusta ja sen muodostavia komponentteja abstraktimmin, eikä keskity itse pelijärjestelmään tai tarkemmin sen toimintaan. Työ pureutuu kandidaatintyötä syvemmin hajautettuihin järjestelmiin, sekä erilaisiin hajautettavuuden ja skaalautuvuuden aiheuttamiin rajoitteisiin erilaisissa järjestelmän osissa. Lisäksi työssä tarkastellaan millaisella arkkitehtuurilla ja toteutuksen periaatteilla skaalautuvuuden etuja voidaan saavuttaa näistä rajoitteista huolimatta.

Työn tarkoitus on skaalautuvan palvelinjärjestelmän suunnittelu ja kehitys reaaliaikaiselle web-sovellukselle. Järjestelmän skaalauksen tarkoituksena on saavuttaa kyky palvella mahdollisimman suurta määrää yhtäaikaisia käyttäjiä.

Käyttäjien tulee pystyä olemaan keskenään interaktiivisessa kanssakäymisessä jaettujen resurssien kautta.

Suorituskykyvaatimuksesta huolimatta itse sovellus pitää pystyä toteuttamaan mielekkäästi, vaikka vaatimus väistämättä rajoittaa sovelluksen toimintaa ja ominaisuuksia. Tasapaino ominaisuuksien ja suorituskyvyn välillä parantaa työn tulosten hyödyntämistä reaalimaailman sovelluksiin, sillä varsinkin kaupallisessa ympäristössä järkevä tasapaino on järjestelmän käytettävyyden keskiössä. Työn tavoitteena on myös osoittaa työssä kuvattujen periaatteiden ja komponenttien toimivuus, skaalautuvuus sekä suorituskyky esimerkkijärjestelmän avulla.

Työssä kuvatun järjestelmän suunnittelufilosofian keskiössä on kyky synkronoida resurssien tilaa palvelimen ja käyttäjien välillä reaaliaikaisesti ilman toistuvia tilan hakuja. Tällä synkronoinnilla pyritään maksimoimaan järjestelmän suorituskyky minimoimalla palvelimen käsittelemien viestien määrä joustamatta käyttöliittymän reaaliaikaisuusvaatimuksesta. Työn tutkimuskysymyksiksi voidaan siis tiivistää:

1. Kuinka hyvin palvelinjärjestelmän suunnittelufilosofia sopii käyttötarkoitukseensa?
2. Kuinka suorituskykyinen vaatimukset täyttävä palvelinjärjestelmä saadaan toteutettua?
3. Millaisiin muihin sovelluksiin järjestelmää voidaan käyttää ja millaisiin se ei sovellu yhtä hyvin?

1.2. Työn rakenne

Työ koostuu johdannosta, viidestä luvusta varsinaista sisältöä sekä yhteenvedosta. Johdannossa esitellään työn taustaa, sen tarkoitus ja rakenne. Toisessa luvussa käydään läpi hajautettujen järjestelmien ominaisuuksia sekä erilaisia hajautettujen järjestelmien arkkitehtuureita. Kolmannessa luvussa pohjustetaan web-ympäristö sekä käydään läpi web-sovelluksiin liittyviä teknologioita.

Palvelinjärjestelmän vaatimukset sekä sen suunnitteluun, tekniikoihin ja arkkitehtuuriin liittyvät asiat käydään läpi neljännessä luvussa. Viidennessä luvussa käsitellään järjestelmän toteutuksen teknologiat, käytännön ratkaisut sekä toteutetaan järjestelmän testaus. Testituloksia, vaatimusten toteutumista sekä tutkimuskysymyksiin vastaamista käydään läpi luvussa kuusi. Lisäksi luvussa käsitellään järjestelmän jatkokehityksen näkymiä pilvipalveluun siirtämisen muodossa. Lopuksi työ tiivistetään yhteenvedossa.

2. HAJAUTETUT JÄRJESTELMÄT

Hajautettu järjestelmä on kokonaisuus itsenäisiä laskentayksiköitä, esimerkiksi tietokoneita tai virtuaalikoneita, jotka vaikuttavat käyttäjälle yhdeltä yhtenäiseltä järjestelmältä [6]. Hajautettua järjestelmää on mahdollista suorittaa usealla fyysisellä tietokoneella samanaikaisesti. Hajautetun järjestelmän sisältämien laskentayksiköiden, eli solmujen (engl. *Node*), määrä voi olla hyvin suuri, eikä solmujen fyysiselle välimatkalle ole käytännössä ylärajaa. Yhden solmun ei kuitenkaan tarvitse olla kokonainen fyysinen tietokone, vaan nykyaikaisissa järjestelmissä virtuaalikoneiden käyttö on yleistä.

Jotta useasta itsenäisestä komponentista koostuva järjestelmä voi toimia yhtenäisesti, tulee komponenttien kyetä kommunikoimaan keskenään. Prosessien välisessä kommunikaatiossa voidaan jakaa tietokoneen sisäisiä alatason resursseja, kuten muistia ja tallennuskapasiteettia. Tällaiset yhteiset resurssit eivät ole saatavilla hajautetun järjestelmän komponenteille, vaan komponenttien välinen kommunikaatio on toteutettava tietokoneverkossa, yleensä internetissä.

Verkkokerroksen kommunikaatio on tietokoneen sisäisten resurssien jakamiseen verrattuna vaikeampaa ja monimutkaisempaa. Se vaatii verkon kanssa yhteensopivien protokollien käyttöä, yhteyksien hallintaa ja datan eheyden varmistamista. Lisäksi kommunikaatioon on toteutettava riittävä virheenkäsittely, sillä se on altis verkon toiminnan häiriöille.

2.1. Ominaisuudet

Vaikka on mahdollista rakentaa hajautettuja järjestelmiä, se ei tarkoita, että ne olisivat välttämättä hyödyllisiä. Hajautetun järjestelmän suunnittelussa ja rakentamisessa on pääsääntöisesti pystyttävä toteuttamaan neljä keskeistä ominaisuutta, jotta järjestelmä on vaivan arvoinen. Keskeiset ominaisuudet ovat resurssien jaettavuus, läpinäkyvyys, avoimuus sekä skaalautuvuus. [6]

Resurssien jako järjestelmässä tulee olla helppoa käyttäjille ja sovelluksille. Resurssit voivat olla periaatteessa mitä tahansa, mutta yleensä jaettavat resurssit ovat dataa tai erilaisia palveluja. Resurssien jakaminen käyttäjien ja sovellusten kesken mahdollistaa tehokkaamman resurssien käytön. Esimerkiksi on tehokkaampaa ja edullisempaa keskittää datan tallennus yhteen paikkaan, kuin tallentaa data erikseen jokaiselle käyttäjälle.

Läpinäkyvyys tarkoittaa järjestelmän kykyä peittää resurssien hajautettu luonne järjestelmän käyttäjältä. Toisin sanoen käyttäjä näkee resurssin, mutta ei tiedä onko se hajautettu esimerkiksi monistamalla (kopioimalla se useampaan kohteeseen) vai sijaintiin perustuen (missä resurssi fyysisesti sijaitsee). Läpinäkyvyyden avulla voidaan myös piilottaa järjestelmän häiriöitä käyttäjältä, mikäli järjestelmän vikasietoisuus ja palautuminen häiriöistä on toteutettu riittävän hyvin.

Avoimuus hajautetussa järjestelmässä tarkoittaa sitä, että järjestelmän osien käyttäminen ja integroiminen muihin osiin on helppoa. Komponenttien tulisi tarjota ominaisuuksiaan erityisten rajapintojen kautta. Rajapintojen tulisi piilottaa komponentin sisäinen toiminta ja rajapintojen tulisi olla keskenään yhtenäisiä, jotta erilaisia komponentteja voidaan liittää toisiinsa helposti sekä riippumatta

komponenttien sisäisistä toimintaperiaatteista tai toteutustavoista. Rajapintojen avulla mikä tahansa komponentti voi käyttää tarvitsemaansa toista komponenttia ainoastaan ymmärtämällä halutun komponentin rajapintaa. Näin mahdollistetaan toteutusriippumattomat komponentit sekä komponenttien vaihtaminen järjestelmässä ilman, että järjestelmän toiminta muuttuu. Toiminnan muuttumattomuus kuitenkin vaatii, että itse rajapinta pysyy muuttumattomana.

Näiden ominaisuuksien saavuttamisen lisäksi järjestelmän suunnittelun haasteita ovat myös heterogeenisyys, vikasietoisuus, tietoturva sekä palvelun laatu [7]. Heterogeenisyys tarkoittaa sitä, että järjestelmä voi koostua erilaisista verkoista, laitteista, käyttöjärjestelmistä tai eri ohjelmointikielillä toteutetuista komponenteista. Verkossa käytettävä tietoliikenneprotokolla piilottaa verkkojen ja laitteiden erot verkkokerroksessa. Mikäli komponenttien välillä on suurempia eroja, voidaan erot piilottaa väliohjelmiston (engl. *Middleware*) avulla. Heterogeenisyyttä ei tule sekoittaa järjestelmän avoimuuteen, joka tarkoittaa komponenttien sisäisen toiminnan piilottamista ja komponenttien välisen integraation helpottamista rajapintojen avulla.

Järjestelmältä voidaan vaatia tietyn tasoista toimintaa erilaisten ominaisuuksien suhteen. Vaatimukset palvelun laadusta (engl. *Quality of Service*) määräytyvät järjestelmän sovelluksen mukaan ja yleensä ne koskettavat suorituskykyä, saavutettavuutta tai tietoturvaa. Järjestelmän saavutettavuutta mitataan tyypillisesti prosenttiosuutena siitä ajasta, kun järjestelmä on toimintakykyinen ja käytettävissä. Erityisesti pilvipalveluympäristöissä käytetään palvelutasosopimusta (engl. *Service-level agreement* eli SLA), joka palvelun laadun ohella määrää palveluntarjoajalle sanktion palvelun laatuvaatimuksen alittamisesta.

2.2. Skaalautuminen

Palvelun skaalaus tarkoittaa palvelun suorituskyvyn muuttamista, yleensä sen kasvattamista, niin, ettei se muuta palvelun sisäistä toimintaa. Skaalaus voidaan jakaa horisontaaliseen skaalaukseen (engl. *Horizontal Scaling* tai *Scale-out*, skaalaus ulos) ja vertikaaliseen skaalaukseen (engl. *Vertical Scaling* tai *Scale-up*, skaalaus ylös). Vertikaalisessa skaalauksessa kasvatetaan palvelua suorittavan tietokoneen resursseja, jotta palvelu pystyy käsittelemään suuremman määrän kuormitusta. Vertikaalinen skaalaus on usein hyvin nopeaa, sillä esimerkiksi prosessorin kellotaajuuden kasvattaminen voidaan tehdä reaaliajassa. Lisäksi lisäresurssit ovat usein palvelun käytössä ilman muutoksia itse palveluun. Resursseja ei kuitenkaan voida lisätä loputtomasti ja raja saattaa tulla vastaan kuorman kasvaessa. [8][9]

Horisontaalisessa skaalauksessa palvelu hajautetaan ja hajautettuja palveluyksiköitä monistetaan usealle tietokoneelle suoritettavaksi. Horisontaalinen skaalaus on vaikeampi toteuttaa, koska palvelun pitää pystyä suoriutumaan monistamisesta huolimatta ja esimerkiksi pystyä kommunikoimaan hajautettujen palvelun osien välillä. Horisontaalisesti skaalattu järjestelmä pystyy usein käsittelemään huomattavasti suuremman kuorman verrattuna siihen, mitä vertikaalisella skaalauksella voidaan saavuttaa. Resurssien kasvattaminen taas on usein vertikaaliseen skaalaukseen verrattuna hitaampaa johtuen uusien tietokoneiden käyttöönotosta ja prosessien käynnistämisestä. Vertikaalisen skaalauksen valitseminen on kuitenkin joissain tilanteissa järkevää, sillä tietynlaisten sovellusten skaalaaminen

horisontaalisesti voi olla vaikeaa. Tällaisia sovelluksia ovat esimerkiksi tietokannat ja tilalliset palvelut. [10]

Horisontaalinen skaalaus voidaan jakaa edelleen paikalliseen skaalaukseen ja globaaliin skaalaukseen. Paikallisessa skaalauksessa laskentayksiköt sijaitsevat samassa fyysisessä sijainnissa, esimerkiksi samassa datakeskuksessa, ja muodostavat yhdessä paikallisen klusterin. Globaalissa skaalauksessa järjestelmän osia sijaitsee useammassa fyysisessä sijainnissa samanaikaisesti, esimerkiksi useassa eri kaupungissa tai jopa eri mantereilla. Hajautetun järjestelmän skaalautuminen ei tarkoita pelkästään järjestelmän resurssien koon skaalaamista, vaan järjestelmää voidaan skaalata myös hallinnollisesti. Hallinnollinen skaalaus tarkoittaa sitä, että järjestelmän hallinta onnistuu siitä huolimatta, että se kattaa useita itsenäisiä organisaatioita samanaikaisesti. [8][10]

2.2.1. Skaalaustarpeen määrittäminen

Järjestelmän hyvän toiminnan kannalta on olennaista, että järjestelmä on skaalattu käyttöön nähden riittävästi. Skaalaaminen kuitenkin sitoo resursseja ja aiheuttaa siten kustannuksia. Siksi on tärkeää tunnistaa skaalauksen tarve sekä se, kuinka paljon mitäkin resurssia tai palvelua on tarpeen skaalata. [9]

Skaalaustarpeen määrittämiseen voidaan käyttää erilaisia indikaattoreita. Indikaattorit perustuvat mittareihin eli mitattaviin arvoihin järjestelmän tilassa ja kuormituksessa. Alhaisen tason mittarit seuraavat tietokoneiden suorittimien käyttöä, vapaana olevan muistin määrää ja vastaavia alhaisen tason resursseja. Tällaisista mittareista voi kuitenkin olla vaikeaa suoraan päätellä järjestelmän suorituskyvyn tilaa. [9]

Korkean tason mittarit ovat sovellustason suorituskyvyn mittareita, kuten esimerkiksi palvelun vastausaika sekä pyyntöjen lukumäärä. Korkean tason mittareita on usein helpompi tulkita, mutta tulkitseminen vaatii usein ymmärrystä sovelluksen toiminnasta, eikä yleiskäyttöisiä mittareita ole. Lisäksi niiden mittaaminen on usein vaikeampaa, sillä palvelun pitää erikseen tukea niitä. Korkean tason mittareiden avulla ei myöskään voida arvioida resurssien tarvetta tarkasti, joka johtaa helposti joko yli- tai alimitoitettuun skaalaukseen. [9]

Resurssien tarpeen määritykseen voidaan käyttää useita tapoja. Sääntöpohjaisessa skaalauksessa lisättävien tai poistettavien resurssien lukumäärä on sidottu mitattuihin arvoihin etukäteen määritettyjen sääntöjen perusteella. Tällainen sääntö voisi olla esimerkiksi *lisää yksi resurssi, jos suorittimen käyttöaste on yli 80 % viiden minuutin ajan*. Toinen tapa arvioida resurssien tarvetta on käyttää ennustemalleja, jotka pyrkivät ennustamaan tulevaa kuormitusta perustuen aiempiin mittaustuloksiin. Myös koneoppimisen keinoja on mahdollista hyödyntää ennustemallien luomisessa. [9]

2.2.2. Automaattinen skaalaus

Automaattinen skaalaus tarkoittaa sitä, että järjestelmä kykenee skaalaamaan itseään ilman ihmisen väliintuloa. Automaattinen skaalaus mahdollistaa järjestelmän resurssien käytön optimoinnin muuttuvissa olosuhteissa niin, että järjestelmä pysyy

toimintakykyisenä ilman ylimääräisiä resursseja ja kustannuksia. Jotta automaattinen skaalaus on mahdollista toteuttaa, on järjestelmän kyettävä itse määrittämään skaalauksen tarpeellisuus sekä vaaditut muutokset resurssien määrissä. [9]

Usein resurssien lisääminen ja poisto ovat järjestelmälle raskaita tai kustannuksiltaan kalliita operaatioita. Tällöin on tärkeää, että automaattinen skaalaus ei tarpeettomasti lisää tai poista resursseja. Toistuvan edestakaisen skaalauksen minimointiin on olemassa erilaisia tapoja, joista yksinkertaisin on aikarajan asettaminen, ennen kuin skaalausta voidaan tehdä uudelleen. Ratkaisuksi voidaan käyttää myös dynaamisia parametreja, joissa skaalaussääntöjä muutetaan sen hetkisen skaalauksen mukaisesti niin, että tarpeeton skaalaus minimoidaan. Lisäksi voidaan pyrkiä tunnistamaan ja poistamaan mittaustuloksista ne tekijät, jotka aiheuttavat tarpeetonta skaalausta. [9]

2.3. Vikasietoisuus

Hajautettu järjestelmä erottuu hajauttamattomasta järjestelmästä siten, että sen on mahdollista olla osittain viallisessa tilassa. Tällaisessa tilassa osa järjestelmästä ei kykene toimimaan oikein, mutta muut osat jatkavat normaalia toimintaansa. Tärkeä tavoite hajautetun järjestelmän suunnittelussa on mahdollistaa järjestelmän automaattinen palautuminen osittaisesta vikatilasta, tehden järjestelmästä hajauttamatonta järjestelmää vikasietoisemman. [6]

Jotta vikasietoisuus osittaista vikatilaa vastaan voidaan saavuttaa, häiriö yhdessä solmussa ei saa suoraan aiheuttaa häiriöitä toisessa solmussa, estäen häiriön leviämisen verkossa. Häiriön tapahtuessa muiden solmujen pitää lähtökohtaisesti pystyä jatkamaan normaalia toimintaansa, mutta mikäli solmulla on muita solmuja riippuvuutenaan, esimerkiksi jaetun datan kautta, voi solmun toiminta muuttua toisen solmun häiriön seurauksena. Mikäli solmut kykenevät suorittamaan keskenään samanlaista toimintaa, voi toinen solmu korvata häiriön kohdanneen solmun ja näin taata kyseisestä solmusta riippuvaisten muiden solmujen normaalin toiminnan jatkumisen. [6][11]

Hajautetusta järjestelmästä voi siis tehdä hajauttamatonta järjestelmää vikasietoisemman niin ohjelmointivirheitä, datan virheitä kuin laitteistokerroksen ongelmiakin vastaan. Järjestelmän vikasietoisuuden kääntöpuolena on se, että solmujen rikkonaisen tilan tai virheellisen toiminnan tunnistaminen luotettavasti voi olla vaikeaa ja toiminnan seuraaminen ohjelmallisesti raskasta. Lisäksi solmujen dynaaminen hallinta verkossa, kuten häiriön seurauksena tapahtuva solmujen muokkaus, käyttöönotto tai poisto, voi olla monimutkaista toteuttaa. [11]

2.4. Mittaaminen

Hajautetun järjestelmän suorituskyvyn arvioimisessa käytetään kahta mittaria: verkon viivettä eli latenssia ja alustan suorituskykyä [11]. Hajautetun järjestelmän toiminta perustuu pohjimmiltaan verkkokommunikaatioon, minkä vuoksi sen suorituskyky on riippuvainen verkon toimintakyvystä. Verkon viiveen kasvu tai yhteysongelmat vaikuttavat suoraan järjestelmän toimintakykyyn ja saatavuuteen. Järjestelmän viiveen

mittaaminen antaa tietoa sen käytettävyydestä, mutta se ei yksinään riitä, koska järjestelmän viiveeseen vaikuttaa myös verkon viive, joka ei välttämättä riipu itse järjestelmästä. Viive mitataan tekemällä pyyntö järjestelmään, odottamalla järjestelmän vastausta tai operaation valmistumista ja mittaamalla tähän kulunut aika.

Alustan suorituskyvyn mittaaminen on toinen tärkeä osa hajautetun järjestelmän suorituskyvyn arviointia. Universaalia mittaustapaa ei ole, vaan menetelmä tulee suunnitella kohdejärjestelmälle sopivaksi. Suorituskykyä voidaan arvioida kuormittamalla järjestelmää operaatioilla tai pyynnöillä ja mittaamalla, kuinka monesta operaatiosta järjestelmä selviää tietyn ajan kuluessa. Yleensä järjestelmän kuormittaminen lisää viiveitä, joten suorituskyvyn määrittelyssä voidaan käyttää ehtona myös suurinta sallittua viivettä.

2.5. Arkkitehtuurit

Kun järjestelmän ei tarvitse toimia useamman tietokoneen välillä, sen rakenne voidaan suunnitella hyvin yksinkertaiseksi. Tällaisen keskitetyn järjestelmän kehitys ja hallinta on yleensä paljon helpompaa ja kustannustehokkaampaa verrattuna hajautettuun järjestelmään, siitä huolimatta, että sillä voidaan toteuttaa toiminnallisuudeltaan vastaavia järjestelmiä. Keskitetyn järjestelmän sisäiset viiveet ovat pieniä ja ennustettavia, koska viestit välittyvät vain yhden tietokoneen sisällä. [12]

Keskitetyn järjestelmän kyky vastata muuttuvaan kysyntään on huono, sillä yksittäisen tietokoneen suorituskyvyn muuttaminen on vaikeaa. Ylläpidettävyys keskitetyssä järjestelmässä on yksinkertaista, mutta järjestelmän käytettävyys voi kärsiä, sillä muutokset tietokoneeseen tai sen ohjelmistoon voivat aiheuttaa käyttökatkoja järjestelmään. Lisäksi keskitetyn järjestelmän vikasietoisuus on huono, sillä ainoan tietokoneen vikaantuminen voi saattaa koko järjestelmän toimintakyvyttömään tilaan. [12]

Kun järjestelmän ohjelmistokomponentit on toteutettu yhtenäisenä ohjelmana, voidaan puhua monoliittiarkkitehtuurista [13]. Monoliittisen sovelluksen etuna on se, että järjestelmän osien välistä kommunikointilogiikkaa ei tarvitse toteuttaa ollenkaan, sillä eri osia ei ole [13]. Tällaisella sovelluksella on myös mahdollista saavuttaa hyvä suorituskyky, sillä komponenttien väliseen yhteistyöhön ei kulu aikaa tai resursseja [14]. Monoliittinen sovellus ei myöskään tarvitse ympärilleen erilaisia tukipalveluita, jolloin järjestelmästä voidaan tehdä yksinkertainen ja luotettava.

Koska tässä työssä käsiteltävän järjestelmän on skaalauduttava tehokkaammaksi, kuin mitä monoliittisella arkkitehtuurilla voidaan saavuttaa, on järjestelmä toteutettava hajautetusti. Hajautettujen järjestelmien arkkitehtuurit vaihtelevat käyttötarkoituksen mukaan, ja eri tyyppejä voidaan katsoa olevan kolme: mikropalveluarkkitehtuuri, kerrosarkkitehtuuri sekä vertaisverkkoarkkitehtuuri [15]. Näistä mikropalvelu- sekä kerrosarkkitehtuurit ovat asiakasohjelma–palvelin-mallisia, kun taas vertaisverkkoarkkitehtuurissa järjestelmän osat ovat keskenään samanarvoisia asiakasohjelmia, eikä palvelimia ole.

Mikropalveluarkkitehtuurin pääperiaatteita ovat itsenäisyys ja, se että yhdellä mikropalvelulla on vain yksi vastuualue. Itsenäisyys tarkoittaa sitä, että mikropalvelu on itsenäisesti käyttöönotettava palvelu, joka on täysin vastuussa tietyn toiminnallisuuden toteuttamisesta. Itsenäisyytensä vuoksi ne sisältävät

kaikki riippuvuudet omien toimintojensa suorittamiseksi, eli niin kirjastot kuin suoritusympäristötkin. [14]

Kerrosarkkitehtuurissa järjestelmän toiminnallisuus on jaettu loogisiin osiin, jotka ovat riippuvaisia toisistaan. Näiden osien voidaan katsoa olevan päällekkäisiä kerroksia, joista jokainen tarjoaa palveluita sen yläpuolella oleville kerroksille, ja joissa data liikkuu pääosin alaspäin käsittelyn edetessä. Web-sovelluksissa yleisesti käytetyssä kerrosarkkitehtuurissa kerroksia on kolme: esitys-, sovellus- ja datakerros. Datakerros on alin kerros, joka vastaa tietojen tallentamisesta ja niiden käsittelystä. Datakerroksen yläpuolella on sovelluskerros, joka vastaa järjestelmän loogisesta toiminnasta. Ylimpänä on esityskerros, joka vastaa järjestelmän käyttöliittymästä. [15]

Myös tässä työssä käsiteltävän järjestelmän arkkitehtuuri on kolmikerroksinen kerrosarkkitehtuuri, jossa käytetyt kerrokset ovat samat kuin yllä esitetyt. Mikropalveluarkkitehtuuri ei sovellu käytettäväksi tässä työssä, sillä itsenäisesti toimivia järjestelmän osia ei juuri ole, vaan reaaliaikaisen viestin välityksen vuoksi järjestelmän osat ovat riippuvaisia toisistaan. Vertaisverkkoarkkitehtuuria ei myöskään voida käyttää, sillä järjestelmä vaatii toimiakseen keskitetyn hallinnan resursseihin ja järjestelmän tapahtumiin.

3. WEB-SOVELLUKSET

Web-selain on sovellus, jonka tarkoituksena on esittää hyperteksti helposti luettavassa muodossa. Toisin sanoen selain muuntaa verkkosivujen lähdekoodin visuaalisesti miellyttäväksi ja ymmärrettäväksi sisällöksi. Nykyaikaiset selaimet ovat kuitenkin huomattavasti monipuolisempia ja pystyvät esimerkiksi esittämään grafiikkaa, toistamaan erilaisia mediatyyppejä ja vastaanottamaan käyttäjän syötteitä eri tavoin.

Yksi web-selaimen keskeisistä ominaisuuksista on sen kyky suorittaa verkosta ladattavaa skriptimuotoista ohjelmakoodia. Ohjelmakoodin avulla voidaan luoda kehittyneitä ja interaktiivisia web-sovelluksia, jotka eivät vaadi asennusta tai erillistä konfigurointia. Käyttäjä voi käyttää näitä sovelluksia siirtymällä selaimellaan oikeaan verkko-osoitteeseen.

Nykyaikaiset web-selaimet hyödyntävät laajalti standardoituja teknologioita ja ohjelmointirajapintoja, mikä tekee web-sovelluksista luontaisesti lähes alustariippumattomia. Selainten väliset erot ovat nykyisin varsin pieniä, joten samat sovellukset toimivat usein hyvin eri selaimissa ja käyttöjärjestelmissä. Tämä mahdollistaa web-sovellusten käytön monilla erilaisilla älylaitteilla riippumatta laitteen tyypistä.

Web-teknologiat eivät kuitenkaan rajoitu vain selaimiin, vaan niitä hyödynnetään myös esimerkiksi työpöytä- ja mobiilisovelluksissa. Selain on tietokoneohjelma, joka suorittaa sisällään toista ohjelmaa, skriptillä ohjelmoitua web-sovellusta. Tämä voi johtaa tehottomuuteen, sillä selain on väliin tuleva suorituskerros.

Yhtenäisyyttä ja yhteensopivuutta, jotka saavutetaan web-teknologioiden avulla, ei tarvitse uhrata selaimen tehottomuuden takia. Sen sijaan on mahdollista suorittaa web-sovelluksia ilman selainta. Käyttämällä web-sovelluksia suoraan tietokoneella perinteisen ohjelman tavoin, sovellusten suorituskykyä voidaan parantaa ja alustan resursseja hyödyntää paremmin. Nämä selainriippumattomat web-teknologiat mahdollistavat ohjelmistokehityksen tehostamisen ja sovellusten levittämisen nopeuttamisen, mikä tarjoaa merkittäviä etuja monipuolisten sovellusten ja käyttöympäristöjen maailmassa. [16]

3.1. HTTP

Selainten ja web-palvelinten väliseen tiedonsiirtoon käytetään laajasti HTTP-protokollaa (lyhenne sanoista *Hypertext Transfer Protocol* eli hypertekstin siirtoprotokolla). HTTP-tiedonsiirto perustuu web-palvelimelle kohdistuvaan TCP-yhteyteen (engl. *Transmission Control Protocol*), joka avataan aina selaimen puolelta. Yhteyden avulla tehdään johonkin web-resurssiin liittyvä pyyntö, kuten vaikkapa tietyn resurssin hakeminen, uuden luonti tai resurssin muutos. Palvelin vastaa pyyntöön, minkä jälkeen TCP-yhteys suljetaan. [17]

HTTP on tilaton protokolla, mikä tarkoittaa sitä, että pyynnön vastaanottajan ei tule tallentaa tietoja kyselyn tekijästä niin, että se vaikuttaisi kyselyn suorittamiseen. Tilattoman protokollan jokaisen pyynnön on oltava ymmärrettävissä yksinään sellaisenaan ja tuottaa aina sama lopputulos ilman edellisistä pyynnöistä riippuvaa kontekstia. Tilattoman protokollan etuna on yksinkertaisempi skaalaaminen, sillä pyynnön kontekstia ei tarvitse käsitellä. Muita etuja ovat muun muassa luotettavuus,

sillä virheestä palautuminen on yksinkertaista, ja näkyvyys, sillä pyyntö itsessään sisältää kaiken mitä sen ymmärtämiseen tarvitaan.

Tilattomuuden toteuttamiseen liittyy kuitenkin ongelmia, sillä jos käyttäjän istunnon tilaa säilytetään palvelimella, saattaa se vaikuttaa pyyntöjen lopputulokseen. Tällöin pyynnöt ovat tilallisia ja tilattomuuden etuja voidaan menettää.

3.2. Palvelinlähtöiset viestit

Interaktiivisissa sovelluksissa ongelmaksi nousevat palvelinlähtöiset viestit, joita HTTP-protokolla ei suoraan tue. Kun selainympäristö aikanaan kehittyi ja selaimelle julkaistiin sovelluksia, kuten pikaviestimiä ja verkkopelejä, palvelinlähtöisiä viestejä kuitenkin tarvittiin HTTP-protokollan rajoitteista huolimatta. Ratkaisuksi vakiintui protokollan väärinkäyttö HTTP-pyyntöjä pitkittämällä (engl. *HTTP Long Polling*). [18]

Pyyntöjen pitkittäminen perustuu siihen, että selain avaa palvelimelle yhteyden normaalisti lähettämällä HTTP-pyynnön, mutta palvelin ei vastaa pyyntöön heti vaan jättää yhteyden auki. Kun selaimelle halutaan lähettää palvelinlähtöinen viesti, voidaan tähän avoimeen yhteyteen vastata. Vastaus välittyy selaimelle normaalin HTTP-pyynnön vastauksen tapaan ja yhteys sulkeutuu. Tämän jälkeen selain avaa uuden yhteyden, jonka palvelin vastaavasti jättää auki ja tätä jatketaan.

Jos selain haluaa lähettää viestin palvelimelle, täytyy auki oleva yhteys sulkea ja lähettää uusi pyyntö tai avata kokonaan uusi yhteys avoimen pyynnön rinnalle. HTTP-pyyntöjen pitkittämisen ongelmana on se, että jokainen viesti, suunnasta riippumatta, aiheuttaa uuden yhteyden avaamisen. Lisäksi uuden yhteyden avaamisen aikana palvelimella ei välttämättä ole avointa yhteyttä, jota käyttää viestien lähettämiseen, ja viestejä voidaan menettää.

3.3. WebSocket

Yhteyden avaaminen jokaisen viestin vuoksi erikseen on raskasta, etenkin interaktiivisissa sovelluksissa, joissa viestejä liikkuu paljon. Kun mukaan myöhemmin otettiin HTTPS-protokolla, jossa HTTP-liikenne on kryptografisesti salattu, yksittäinen pyyntö muuttui entistä hitaammaksi ja raskaammaksi, jolloin ongelma korostui.

Ratkaisuksi kehitettiin aluksi SSE (lyhenne sanoista *Server Sent Events*). SSE:n avulla selain voi kuunnella palvelimelta tulevia tekstimuotoisia viestejä, mutta ei lähettää viestejä takaisin. Palvelimen viestin saadessaan selain voi kuitenkin tehdä sopivan HTTP-pyynnön, jolloin kaksisuuntainen viestinvälitys voidaan saavuttaa luotettavasti ilman HTTP-protokollan väärinkäyttöä. Kuitenkin jokainen selaimen lähettämä viesti vaatii edelleen uuden yhteyden muodostamisen.

Lopulta ratkaisuksi kehitettiin WebSocket-protokolla, joka mahdollistaa yhden avoimen TCP-yhteyden läpi tapahtuvan, selaimen ja palvelimen välisen, molempisuuntaisen kommunikaation. Jotta protokolla olisi yhteensopiva olemassa olevien web-palvelinten, palomuurien ja välityspalvelinten kanssa, se käyttää HTTP- ja HTTPS-yhteyksien kanssa yhteneviä portteja. WebSocket-yhteys muodostetaan

erityisen HTTP-pyynnön avulla, jotta yhteyden avaaminen olisi yksinkertaista selainympäristössä. Tästä huolimatta itse WebSocket-yhteys on oma itsenäinen TCP-yhteytensä, jolla ei muilta osin ole yhteyttä HTTP-yhteyksiin. [19]

3.4. JavaScript

JavaScript on vuonna 1995 julkaistu skriptikieli, joka alun perin kehitettiin selaimen käyttöliittymän ohjelmointiin. Vuosien varrella JavaScriptin käyttö laajeni, kun käyttöliittymien ohjelmointi muuttui monimutkaisemmaksi ja kun sovelluksista piti saada aiempaa interaktiivisempia. JavaScriptin yhteensopivuusstandardin, ECMAScriptin, ansiosta JavaScript toimii kaikilla selaimilla ja on nykyisin ainoa selaimissa käytettävissä oleva ohjelmointikieli, jolla voidaan ohjelmoida selaimen käyttöliittymää. Kieli on saavuttanut suurta suosiota myös selainympäristön ulkopuolella, yhteensopivuuden, käytön helppouden ja kielen ympärille syntyneen suuren ekosysteemin ansiosta. JavaScript oli vuonna 2021 maailman suosituin ohjelmointikieli kysymys- ja vastaussivusto Stack Overflow:n kyselytutkimuksessa. [20][21]

JavaScriptin dynaaminen ja heikko tyypitys altistaa vaikeasti havaittaville ohjelmointivirheille ohjelmien koon ja monimutkaisuuden kasvaessa. TypeScript on Microsoftin kehittämä ohjelmointikieli, joka lisää JavaScriptiin tyypityksen sekä muita ominaisuuksia. Tyyppijärjestelmä mahdollistaa ohjelmakoodin staattiseen analyysiin perustuvan virheiden havaitsemisen ennen ohjelman suorittamista. TypeScript-koodia ei voi suorittaa JavaScript-tulkilla, vaan se on muunnettava ennen suoritusta JavaScriptiksi TypeScript-kääntäjällä. [22][23]

Web-sovelluksia kehitetään puhtaasti JavaScriptiä kirjoittamalla enää hyvin harvoin. Moderniin web-sovelluskehitykseen kuuluu yleensä web-kehityskehyksiä ja monenlaisia työkaluja, joiden avulla alkuperäinen koodi esimerkiksi paketoidaan, pienennetään, optimoidaan, yhteensopivuutta parannetaan tai tarkoituksellisesti monimutkaistetaan vaikeasti luettavaksi ennen koodin lähettämistä selaimeen suoritettavaksi. On myös olemassa työkaluja, jotka muuntavat täysin eri ohjelmointikielen koodin samalla tavalla toimivaksi JavaScriptiksi. Vaikka teknisesti selaimessa ei voikaan suorittaa muuta kuin JavaScriptiä, muistuttaa modernien sovellusten lähdekoodi jatkuvasti vähemmän itse suoritettavaa koodia. [24]

3.5. Uudet teknologiat

Progressiiviset web-sovellukset (engl. *Progressive Web Apps* eli PWA) ovat web-sovelluksia, joissa hyödynnetään natiivien sovelluksien ominaisuuksia. Natiivit sovellukset ovat suoraan käyttöjärjestelmään asennettavia sovelluksia, joiden ominaisuuksia on nykyään saatavilla modernissa selainympäristössä. Tällaisia ominaisuuksia ovat esimerkiksi kyky toimia ilman aktiivista verkkoyhteyttä, kyky käyttää järjestelmän resursseja ja laitteita, kuten kameraa, mikrofonia tai GPS:ää, kyky tallentaa suuria määriä tietoja laitteen muistiin sekä kyky käynnistää taustalla suoritettavia palveluprosesseja. PWA mahdollistaa natiivisovellusten tasoisen

käyttökokemuksen selainympäristössä ilman, että käyttäjän tarvitsee erikseen asentaa sovellusta laitteelleen.

Myös käyttöjärjestelmään asennettavia PWA-sovelluksia on olemassa. Tällöin sovellus, joka vaikuttaa olevan natiivisovellus, onkin todellisuudessa omassa ikkunassaan oleva selainikkuna. Tällaiset sovellukset ovat yleensä kooltaan pieniä ja kevyitä asentaa, sillä sovelluksen pohjana käytetty selain on jo asennettuna käyttöjärjestelmään. Web-sovellusten kehittäminen on pääosin huomattavasti nopeampaa ja helpompaa kuin natiivisovellusten kehittäminen, johtuen selaimen tarjoamista ominaisuuksista ja hyvästä yhteensopivuudesta laitteiden välillä. [4]

Viime vuosina JavaScriptin rinnalle on noussut WebAssembly, joka mahdollistaa raskaiden ohjelmistojen suorittamisen selainympäristössä. WebAssembly ei ole ohjelmointikieli, vaan käskykanta, jonka tarkoitus on olla kääntämiskohde erilaisille käännettäville ohjelmointikielille, kuten vaikkapa C:lle. Se käyttää prosessoinnissaan hyödyksi kaikkea saatavilla olevaa laitteistoa, ja pyrkii lähes natiivitason suorituskykyyn. JavaScriptiä se ei kuitenkaan vielä korvaa, sillä WebAssemblyllä ei toistaiseksi ole pääsyä verkkoon tai sivun käyttöliittymään. [25]

Kun salauskerros aikanaan lisättiin TCP-pohjaiseen HTTP-protokollaan, lisäsi se yhteyden muodostamisen vaiheita ja pyynnöt hidastuivat entisestään. Vuonna 2022 Googlen kehittämä UDP-pohjainen QUIC-protokolla valittiin HTTP/3:n perustaksi ratkaisemaan suorituskykyongelmia. UDP-pohjainen protokolla vähentää pyyntöjen viivettä ja lyhentää latausaikoja, ja joissain tapauksissa nopeusparannus voi olla jopa kolminkertainen. HTTP/3:n käyttöönotto on kuitenkin vielä varhaisessa vaiheessa, vaikka pääosa selaimista tukeekin sitä jo. [26][27]

Selainten ominaisuudet ovat laajentuneet niin paljon, että hyvin suuri osa ohjelmistoista voidaan nykyisin toteuttaa web-sovelluksina. Selainohjelmistojen kehitys on tehokasta laajan yhteensopivuuden, erilaisten alustojen, web-kehityskehyksien ja työkalujen avulla. Valmiin ohjelmiston jakelu loppukäyttäjille on helppoa web-sivujen luonteen vuoksi. Lisäksi markkinoilla on jatkuvasti enemmän web-kehittäjiä.

Selainalustaa käytetään nykyään myös käyttöjärjestelmänä esimerkiksi Googlen Chromebook -tuoteperheen kannettavissa tietokoneissa, sekä valikoimassa pienempiä tietokoneita. Tulevaisuudessa todennäköisesti nähdään entistäkin enemmän web-sovelluksia ja selainalustoja yhä erilaisimmissa ratkaisuissa ja laitteissa. Myös web-sovellusten suorituskyvyn voidaan odottaa nousevan natiivisovellusten rinnalle. [28][29]

3.6. WebRTC

QUIC-protokolla ei kuitenkaan ole nykyselainten ainoa poikkeus perinteiseen HTTP-protokollaan. Kansainvälisen WWW-standardeja ja suosituksia kehittävän World Wide Web Consortiumin (W3C) suosituksiin vuonna 2021 lisätty WebRTC (engl. *Web Real-Time Communication*) on selaimen ohjelmointirajapinta, joka mahdollistaa kommunikaation selainten välillä. WebRTC mahdollistaa reaaliaikaiset yhteydet, kuten äänipuhelut ja videopuhelut sekä tiedonsiirron suoraan selainten välillä ilman erikseen asennettavia selainliitännäisiä. WebRTC on nykyään saatavilla kaikissa moderneissa selaimissa. [30][31]

WebRTC:n avulla voidaan saavuttaa korkea reaaliaikaisuus, sillä ylimääräiset viiveet voidaan minimoida, kun tietoja ei tarvitse kierrättää palvelimen kautta. Palvelimen kuormitus vähenee kun, palvelimen resursseja ei tarvitse sitoa tiedon välittämiseen. WebRTC:n avulla voidaan toteuttaa myös vertaisverkkoja (engl. *peer-to-peer networks*), joissa tieto kulkee pelkästään käyttäjien välillä eikä palvelimia tarvita ollenkaan. Tällaisia palveluita ovat esimerkiksi tiedostojen jakopalvelut, kuten BitTorrent, sekä kryptovaluuttojen lohkoketjut, kuten Bitcoin [32][33]. Lisäksi jotkut sisällönjakeluverkot (engl. *Content Delivery Network*, CDN) ovat ottaneet WebRTC:n käyttöön hyödyntääkseen asiakkaan internetyhteyttä sisällön jakamiseen muille käyttäjille tehden asiakkaan selaimesta palvelimen [34].

Koska suurin osa internettiin kytketyistä laitteista sijaitsee osoitteenmuunnoksen (engl. *Network Address Translation*, NAT) takana, ei asiakasohjelmien ole yleensä mahdollista muodostaa suoria yhteyksiä toisiinsa, koska sisäverkon laite ei ole tietoinen omasta internetissä näkyvästä osoitteestaan. Lisäksi osoitteenmuunnos voi aiheuttaa sen, ettei julkinen verkko-osoite riitä paikantamaan laitetta sisäverkosta. WebRTC käyttää kolmea eri tekniikkaa NAT-ongelman ratkaisemiseksi: ICE (engl. *Interactive Connectivity Establishment*) tarvitaan määrittämään millaiset eri yhteysmuodot ovat mahdollisia laitteiden välillä niiden verkkojen muodon ja palomuurisääntöjen puitteissa. STUN (engl. *Session Traversal Utilities for NAT*) avulla voidaan muodostaa yhteys yksinkertaisessa tapauksessa, jossa yhteys voidaan muodostaa osoitteenmuunnoksesta huolimatta suoraan laitteiden välille. STUN-palvelimen tehtävä on kertoa laitteelle sen julkinen verkko-osoite. Vaikeampiin tapauksiin käytetään TURN:ia (engl. *Traversal Using Relays around NAT*), jossa yhteys reititetään TURN-palvelimen kautta, ohittaen palomuurit ja kiertäen osoitteenmuunnoksen aiheuttamat ongelmat. [35]

Palvelinkuormituksen säästö menetetään TURN-palvelua käytettäessä, kun selainten väliset yhteydet kulkevat kuitenkin palvelimen kautta. Vaikka TURN ja STUN tarjotaan samalta palvelimelta, pyritään TURN-palvelun käyttöä välttämään sen raskauden vuoksi [35]. Kuitenkin sen käytöllä kuormitus voidaan siirtää pois itse sovelluksen palvelimelta säästäen sen resursseja.

WebRTC:tä voitaisiin käyttää reaaliaikaiseen tiedonvälitykseen REST- tai WebSocket-rajapintojen sijaan, mikäli tieto on luonteeltaan sellaista, ettei sitä tarvitse tallentaa tai käsitellä palvelimella. Tällaista tietoa on kertakäyttöinen tieto, kuten ääni- ja videopuhelut, pikaviestit, joiden historiaa ei tarvitse säilyttää asiakasohjelman ulkopuolella, sekä erilaiset nopeasti muuttuvat tiedot, kuten vaikkapa pelihahmon tarkka sijainti.

WebRTC:tä voitaisiin käyttää myös palvelinta kiinnostavien tietojen reaaliaikaiseen välitykseen. Tällaisessa tilanteessa palvelimen suorituskykyä voidaan kasvattaa keräämällä viestejä ja käsittelemällä viestit myöhemmin ryppäissä tyyppinsä mukaan. Palvelimen ei tarvitse reagoida viesteihin välittömästi, sillä asiakasohjelmat ovat saaneet viestit reaaliajassa muilta asiakasohjelmilta. Tässä työssä esitetyssä järjestelmässä ei käytetä WebRTC:tä, sillä se on puhtaasti selaimen ominaisuus, eikä työ käsittele asiakasohjelmia lainkaan.

4. PALVELINJÄRJESTELMÄ

Tässä työssä käsiteltävä palvelinjärjestelmä on useammalla palvelintietokoneilla suoritettava ohjelmistokokonaisuus, joka toimii taustajärjestelmänä web-pohjaiselle asiakasohjelmalle. Palvelinjärjestelmän toteuttamiseksi järjestelmä on ensin suunniteltava. Suunnittelu aloitetaan tunnistamalla järjestelmän vaatimukset, ja valitsemalla käytetyt tekniikat, joiden avulla vaatimukset voidaan täyttää. Järjestelmän arkkitehtuuri ja sen komponentit suunnitellaan valittujen tekniikoiden puitteissa, samalla valiten käytettävät teknologiat ohjelmistokomponenttien toteuttamiseksi.

4.1. Suunnittelufilosofia

Ennen järjestelmän suunnittelun aloittamista, voidaan tutkia järjestelmän tilaa, tilan muuttumista sekä käyttäjän vaikutusta järjestelmän tilaan. Tehokas tilan hallinta hajautetussa ympäristössä johtaa perustavanlaatuisiin periaatteisiin, joiden ympärille järjestelmä tulee suunnitella. Näiden periaatteiden jäsentely helpottaa tekniikoiden ja teknologioiden mielekästä valintaa, ja siten toimivan kokonaisuuden saavuttamista. Suunnittelun periaatteiden muodostamaa kokonaisuutta voidaan kutsua järjestelmän suunnittelufilosofiaksi.

4.1.1. Ajantasainen tila

Järjestelmä voidaan ajatella tilakoneena, johon tallennettu tieto, eri komponenttien konfiguraatio ja niiden ajoaikainen tieto muodostavat järjestelmän tilan. Järjestelmä siirtyy tilasta toiseen sisäisen ohjelmointinsa perusteella, erilaisten rajapintojen kautta suoritettujen toimintojen avulla. Toimintoja on kahdenlaisia: ulkopuolelta tulevia käyttäjän tekemiä toimintoja sekä järjestelmän sisäisiä ajastettuja toimintoja. Vastaavasti, jos toimintoja ei suoriteta, järjestelmän tila ei muutu.

Jotta käyttäjä voisi luotettavasti havainnoida järjestelmän tilan muutoksia, pitää järjestelmän tila synkronoida jollain tavalla palvelinjärjestelmän ja asiakasohjelman välillä. Tilan ja tilasiirtymien informaation välittämiseen käytetään ohjelmointirajapintoja, joiden kautta asiakasohjelma voi hakea ja vastaanottaa tietoja internetyhteyden avulla. Lähetettävä informaatio pitää kuvata sellaisessa muodossa, että sen lähettäminen internetin yli onnistuu niin, että vastaanottaja pystyy hyödyntämään lähetetyn informaation.

Sen lisäksi, että koko järjestelmän tilan lähettäminen asiakasohjelmalle on raskasta ja vaikea toteuttaa, on se tyypillisesti myös turhaa. Normaalissa käytössä on myös tyypillistä, että kaikki järjestelmän tilan osat eivät ole käyttäjän havainnoitavissa samanaikaisesti. Lisäksi järjestelmässä on taustapalvelukomponentteja, joiden tila on asiakasohjelman ja käyttäjän näkökulmasta pitkälti merkityksetöntä. Suorituskykysyistä on siis kannattavaa minimoida siirrettävän informaation määrä, ja pyrkiä siirtämään asiakasohjelmalle vain se informaatio, mitä järjestelmän luonteva käyttö välttämättä edellyttää. Lähetettävää informaatiota voidaan myös joutua rajaamaan esimerkiksi tietoturvasyistä tai silloin jos järjestelmän suunniteltu toiminta vaatii käyttäjä- tai organisaatiokohtaista tietojen eristämistä (engl. *multitenancy*).

Verkon ja laitteiston viiveiden vuoksi todellista reaaliaikaisuutta on mahdotonta saavuttaa, mutta hyvään käyttäjäkokemukseen riittää vähempikin. Jotta asiakasohjelma saadaan pidettyä ajan tasalla palvelinjärjestelmän tilasta, voidaan tilan tiedot lähettää asiakasohjelmalle toistuvasti. Tällainen lähestyminen ei kuitenkaan ole kovin tehokasta, sillä muuttumattomien tietojen osalta asiakasohjelmalle lähetetään toistuvasti samaa tietoa.

Suorituskykyä voidaan lähteä parantamaan lähettämällä tilatieto asiakasohjelmalle aluksi kerran ja jatkossa vain tieto kyseisen tilan muutoksesta. Asiakasohjelma voi paikallisesti laskea uuden tilan alkutilasta sekä vastaanotetusta tilan muutoksesta. Toisaalta, jos viesti tilan muutoksesta jää toimittamatta, tila jää vanhentuneeksi, ja jos kaksi samaan tietoon kohdistuvaa muutosta käsitellään väärässä järjestyksessä, jälkimmäinen jää voimaan ja tila on eri kuin palvelinjärjestelmässä. Siispä, jotta asiakasohjelman voidaan taata olevan oikeassa ja ajantasaisessa tilassa, pitää tiedonsiirtokanavan pitää huolta siitä, että kaikki viestit pääsevät varmasti perille ja että saapuvat viestit käsitellään samassa järjestyksessä kuin ne lähetettiin.

4.1.2. Synkronointi

Jotta järjestelmän koko tilaa ei tarvitse synkronoida asiakasohjelman kanssa, tila jaetaan loogisiin pienempiin kokonaisuuksiin, resursseihin, joita tarjotaan rajapintojen kautta asiakasohjelmien saataville. Asiakasohjelma pyytää rajapintojen kautta sellaisten resurssien tietoja, joita se tarvitsee sen hetkistä näkymää tai toimintaansa varten. Skaalautumisen kannalta on tärkeää, ettei asiakasohjelma pyydä käyttöönsä kuin vain niitä resursseja, joita se sillä hetkellä tarvitsee.

Jokaiseen resurssiin voidaan sitoa yksi tai useampi kanava, joiden kautta resurssiin liittyvät muutokset välitetään. Kanavan tarkoitus on pitää kirjaa kanavan resurssista kiinnostuneista asiakasohjelmista. Kun viesti syntyy palvelinohjelmistossa, valitaan viestille yksi tai useampi kanava, jonka kautta viesti välitetään asiakasohjelmille.

Kaikki palveluyksiköt pitävät kirjaa omista aktiivisista yhteyksistään asiakasohjelmiin. Kun asiakasohjelma pyytää resurssin tietoja, tai muulla tavalla ilmoittaa olevansa kiinnostunut kyseisen resurssin muutoksista, palveluyksikkö rekisteröi asiakasohjelman kyseisen resurssin kanavaan. Resurssin muutoksen yhteydessä luodaan viesti, jossa kuvataan tapahtuneet muutokset, ja joka lähetetään kaikkiin kanaviin, joihin resurssi on rekisteröity. Vastaavasti, mikäli asiakasohjelma ei ole enää kiinnostunut resurssin muutoksista ja ilmoittaa siitä, asiakasohjelma poistetaan pyydetyltä kanavalta eikä viestejä muutoksista enää lähetetä.

4.1.3. Tavoitellut hyödyt

Vaihtoehtoinen tapa toteuttaa tilan synkronointi olisi lähettää resurssin muutoksen tietojen sijaan vain tieto siitä, että resurssiin on tullut muutos. Tämän jälkeen asiakasohjelma voisi tehdä uuden hakupyynnön resurssista ja vastaanottaisi resurssin päivitetyn tilan. Vaikka tällainen tapa olisi luultavasti helpompi toteuttaa, on se vähemmän tehokas, sillä jokainen asiakasohjelma tekisi oman pyyntönsä resurssin tilan saamiseksi, ja kaiken lisäksi samaan aikaan.

Merkittävin tavoiteltu hyöty tilan muutosten lähettämisessä on, että muutoksesta kertovan viestin sisältö pitää luoda vain kerran. Sama viesti lähetetään jokaiselle muutoksesta kiinnostuneelle asiakasohjelmalle kanavien perusteella, jolloin tarvittava prosessointi minimoidaan. Lähetettävien viestien koko ja verkon kuormitus saadaan minimoitua, kun viesti sisältää ainoastaan muutokset eikä koko resurssin tilaa. Lisäksi suurikaan määrä vastaanottajia ei juuri lisää prosessointia, sillä mitään vastaanottajakohtaisia tarkistuksia tai muutakaan prosessointia ei tarvita. Kaikki tarpeellinen on jo tehty, mikäli asiakasohjelma on kanavalla.

Erityisesti jos järjestelmässä on resursseja, joita muokataan paljon ja joiden muutoksista on kiinnostunut suuri määrä asiakasohjelmia, on saman viestin välitys erityisen tehokasta. Mikäli istunnot järjestelmässä ovat pitkiä tai istunnossa aktiiviset resurssit ovat pitkälti samoja koko istunnon ajan, vähentää tällä tavoin tehty tilan synkronointi voimakkaasti tarvittavien hakujen määrää ja sitä kautta järjestelmän lukuoperaatioiden määrää. Vastaavasti jos kukaan ei tee mitään, ei järjestelmänkään tarvitse tehdä mitään, eikä resurssien tila silti vanhene eikä niihin liittyviä pyyntöjä tarvitse tehdä ollenkaan.

Sovelluksia, joihin ratkaisu sopii erityisen hyvin, ovat resurssien reaaliaikaista ja jaettua muokkausta vaativat sovellukset, kuten modernit tekstinkäsittelyohjelmat, taulukkolaskentaohjelmat, sekä projektien ja organisaatioiden hallintaohjelmistot. Lisäksi ratkaisu sopii nopeita muutoksia ja korkeaa interaktiivisuutta vaativiin sovelluksiin, kuten viestintäsovelluksiin ja erilaisiin peleihin. Ratkaisu sopii myös reaaliaikaisuutta vaativaan ja erityisen suuren yleisön käyttöön tarkoitettuun tiedotuskanavakäyttöön ja uutispalveluihin.

4.1.4. Tunnistetut heikkoudet

Muutosten välittäminen resurssin kanavien kautta on tehokasta, mutta vaikeuttaa järjestelmän kehitystä. Kun yhteys muodostetaan uudelleen sen katkeamisen jälkeen, on muutoksia resursseihin voinut tapahtua yhteyden ollessa poikki. Yhteyden katkeamisen jälkeen ei siis voida enää luottaa siihen, että asiakasohjelman ylläpitämä tila resursseista olisi enää ajantasainen.

Tila voidaan synkronoida hakemalla jokainen resurssi uudelleen tai siten, että palvelin tallentaa katkenneelle yhteydelle suunnatut viestit vielä jonkin aikaa yhteyden katkeamisen jälkeen. Viestien tallentamiseen on liitettävä vanhenemisaika, sillä ei voida tietää milloin uusi yhteys muodostetaan. Tallennetut viestit lähetetään asiakasohjelmalle heti, kun uusi yhteys on muodostettu. Uuden yhteyden muodostamisen ohessa palvelimelle lähetetään vanhan yhteyden tunniste, jotta tallennetut viestit voidaan löytää ja lähettää. Mikäli aikaa on kulunut liikaa, eikä tallennettuja viestejä enää löydy, tai jos uuden yhteyden ohessa ei ole lähetetty vanhan yhteyden tunnistetta, on asiakasohjelman nollattava oma tilansa pyytämällä kaikki sen tarvitsemat resurssit uudelleen.

Mikäli resurssien haku tehdään eri rajapinnan kautta kuin muutokset vastaanotetaan, esimerkiksi, jos käytössä on HTTP hakuihin ja WebSocket-yhteys muutosten vastaanottamiseen, täytyy tieto kanavalle liittämisestä välittyä sille palveluyksikölle, jolle muutosten lähettämiseen tarkoitettu yhteys on muodostettu. Riippuen skaalauksesta ja kuorman jakamisen logiikasta, etenkin HTTP-pyynnöt voivat

jakautua satunnaisesti eri palveluyksiköille eikä avoimiin yhteyksiin välttämättä edes käytetä samaa palvelua kuin hakuihin. Tieto kanavalle liittämisestä kulkee palvelinjärjestelmän viestipalvelun kautta, joka välittää tietoja palveluyksiköiden välillä. Toinen ratkaisu kanavalle liittymisen seuraamiseen on se, että asiakasohjelma pyytää liittyä resurssin kanavalle sitä hakiessaan ja käyttää tämän pyynnön tekemiseen samaa rajapintaa, jota muutosten vastaanottamiseen käytetään. Tällöin tietoa kanavalle liittämisestä ei tarvitse välittää palvelinjärjestelmässä, mutta kanavien hallinta asiakasohjelman puolella vaikeutuu.

Jotta muutoksien lähettäminen olisi mahdollisimman tehokasta, ei asiakasohjelmia tarkastella ollenkaan viestejä lähettäessä. Resursseihin liittyvät käyttöoikeudet täytyy siis tarkistaa kanavalle liittymisen yhteydessä. Käyttöoikeudet eivät myöskään saa vaikuttaa mitenkään viestin sisältöön, sillä sama viesti lähetetään kaikille kanavalle liittyneille asiakasohjelmille. Siispä käyttäjän käyttöoikeuksien muuttuessa on tarkistettava mihin resursseihin muutos vaikuttaa, ja poistaa käyttäjän asiakasohjelmat sellaisilta kanavilta, joihin liittyviä resursseja käyttäjä ei saa enää nähdä. Kanavalta poistamisesta on lisäksi ilmoitettava asiakasohjelmalle, jotta se ei virheellisesti luule resurssin olevan edelleen ajan tasalla tai käytettävissä. Tilannetta, jossa vain osa resurssista tai resurssin osista on tietyn käyttöoikeuden takana ei saa syntyä, sillä viestiä ei voida räätälöidä käyttäjän käyttöoikeuksien mukaan. Tällaisessa tilanteessa resurssi pitää jakaa useampaan resurssiin, joista jokainen on kokonaisuudessaan tiettyjen käyttöoikeuksien takana.

Viestejä ei voida räätälöidä minkään muunkaan käyttäjään liittyvän perusteella, sillä viesti ei tällöin olisi sellaisenaan lähetettävissä kaikille siitä kiinnostuneille. Tällaisen tietojen räätälöinnin, kuten käännökset, arvojen esitystavat, aikavyöhyketieto ja niin edelleen, täytyy siis tapahtua kokonaisuudessaan asiakasohjelmassa. Toisaalta vastaavan lähestymistavan motiivit pätevät myös tilattomaan ohjelmointirajapintaan, sillä pyyntöjen vastausten tallentaminen välimuistiin ei ole mahdollista, mikäli vastaukset ovat riippuvaisia käyttäjästä.

Mitä enemmän kanavien hallinta jätetään asiakasohjelman vastuulle, sitä suurempi on riski siitä, että asiakasohjelmalla jää kanavia auki sellaisiin resursseihin, joita ei enää tarvita. Jos istunnot ovat pitkiä, voi avoimia kanavia kertyä huomattavia määriä ja palvelimen kuorma kasvaa, kun viestejä lähetetään turhaan. Toisaalta ylimääräisen asiakasohjelman vaikutus kanavalla on hyvin pieni, sillä pelkästään kanavalla oleminen ei luo muutoksia järjestelmässä ja viestien lähetys on tehty mahdollisimman kevyeksi.

Jos järjestelmän resursseista vain harva on useamman käyttäjän käytössä samanaikaisesti, ei muutosten lähettämisestä reaaliajassa ole suurta hyötyä, sillä jokainen muutos näkyy pääasiassa vain muutoksen tekijälle itselleen, eikä tähän tarvita reaaliaikaista viestintää. Mikäli istunnot järjestelmässä ovat hyvin lyhyitä tai jos resursseissa tapahtuu vain vähän muutoksia, ei reaaliaikainen tilan synkronointi ole silloinkaan erityisen hyödyllistä. Esimerkkejä sovelluksista, joihin ratkaisu ei parhaiten sovellu ovat tietojen katseluun pääasiassa käytetyt tietojärjestelmät, kuten kirjaston katalogi tai verkkokauppa, tai sellaiset sovellukset, joissa resurssilla on hyvin harvoin useampi käyttäjä samanaikaisesti, kuten potilastietojärjestelmä tai sähköpostiohjelma.

4.2. Vaatimukset

Järjestelmän tarkoitus on palvella reaaliaikaista ja interaktiivista verkossa toimivaa web-sovellusta. Ideaalisti tämä tarkoittaisi, että kaikki informaatio olisi käyttäjän saatavilla sillä hetkellä, kun se luodaan. Valon nopeudesta sekä laitteiston ja protokollien aiheuttamista viiveistä johtuen, informaation välittäminen komponenttien välillä ottaa aina aikaa. Siispä ideaaliin reaaliaikaisuuteen ei voida päästä, mutta viiveitä voidaan silti pyrkiä minimoimaan.

Reaaliaikaisuutta vaativan sovelluksen käyttökokemus yleensä kärsii voimakkaasti viiveiden noustessa. Käyttöliittymä voi tuntua tahmealta ja sovelluksen mielekäs käyttö voi jopa kokonaan estyä, jos viiveet kasvavat liian suuriksi. Internet-puhelu on hyvä esimerkki pieniä viiveitä vaativasta reaaliaikaisesta sovelluksesta, sillä luonteva vuoropuhelu ei onnistu, jos osapuolten välillä on viivettä edes muutamia sekunteja. Lisäksi nopeasti muuttuvien tietojen ajantasainen välitys on tärkeää esimerkiksi peleissä ja sosiaalisissa sovelluksissa.

Järjestelmän käyttäjämäärää tai kuormitusta on yleensä vaikea ennustaa, jolloin on kustannustehokasta voida vaikuttaa järjestelmän käyttämiin resursseihin dynaamisesti tarpeiden mukaan. Etenkin käyttäjämäärän nopean kasvun yhteydessä skaalautuvuus on tärkeää, sillä mikäli järjestelmä ei voi vastata kasvaneeseen kysyntään, käyttökokemus voi kärsiä ja järjestelmä voi jopa olla kokonaan käyttökelvoton. Järjestelmän täytyy siis olla skaalautuva, ja sen suorituskykyä on voitava parantaa lisäämällä resursseja. Vaatimuksiksi muodostuvat siis reaaliaikaisuus, interaktiivisuus, skaalautuvuus ja suorituskyky.

4.2.1. Reaaliaikaisuus

Reaaliaikaisuus tarkoittaa järjestelmän kykyä reagoida tapahtumiin mahdollisimman nopeasti, ja kykyä välittää muutoksia käyttäjälle mahdollisimman pienellä viiveellä. Reaaliaikaisuus mahdollistaa myös nopeasti muuttuvien tietojen ajantasaisen välittämisen käyttäjälle. Reaaliaikaisuusvaatimuksen toteutumista arvioidaan mittaamalla järjestelmän toimintojen viiveitä.

4.2.2. Interaktiivisuus

Interaktiivisuus tarkoittaa järjestelmän kykyä olla vuorovaikutuksessa käyttäjän kanssa. Luonteva interaktiivisuus vaatii järjestelmältä reaaliaikaisuutta, mutta myös järjestelmän kykyä reagoida käyttäjän toimintoihin nopeasti. Koska järjestelmän pitää tarjota samoja resursseja usean käyttäjän käyttöön samanaikaisesti, on interaktiivisuuden vaatimuksena myös se, että käyttäjät voivat olla vuorovaikutuksessa järjestelmän kautta toistensa kanssa. Siispä interaktiivisuusvaatimuksen toteutumista voidaan arvioida tarkastelemalla käyttäjien kykyä olla reaaliaikaisessa vuorovaikutuksessa toistensa kanssa.

4.2.3. Skaalautuvuus

Skaalautuvuus tarkoittaa järjestelmän kykyä muuttaa suorituskykyään ja kapasiteettiaan vastatakseen muuttuviin käyttäjämääriin ja tehtäviin. Tässä työssä käsitellään ainoastaan ohjelmiston skaalautuvuutta, vaikka skaalautuvuus sisältää ohjelmiston skaalautuvuuden lisäksi myös laitteiston skaalautuvuuden. Skaalautuvuusvaatimuksen toteutumista arvioidaan järjestelmän eri komponenttien skaalautuvuuden tarkastelulla sekä testaamalla komponenttien skaalaamista.

4.2.4. Suorituskyky

Suorituskyky tarkoittaa järjestelmän kykyä suorittaa tehtäviä mahdollisimman nopeasti ja mahdollisimman suurissa määrissä. Tässä työssä pyritään maksimoimaan järjestelmän suorituskyky skaalaamalla palveluita, jotta järjestelmä voisi palvella mahdollisimman montaa käyttäjää samanaikaisesti. Suorituskykyvaatimuksen toteutumista arvioidaan mittaamalla järjestelmää kuormatestauksen avulla.

4.3. Tekniikat

Vaatimukset toteuttava järjestelmä on väistämättä hyvin monimutkainen kokonaisuus. Suunnittelu- ja kehitystyötä voidaan merkittävästi helpottaa rakentamalla järjestelmä käyttäen erilaisia tietyn ongelman ratkaisemiseksi kehitettyjä tekniikoita. Tekniikat abstrahoivat eli piilottavat alleen paljon järjestelmän kokonaiskuvan kannalta epäolennaisia teknisiä yksityiskohtia yksinkertaistaen järjestelmän suunnittelua. Yleisien tekniikoiden toteuttamiseen on usein saatavilla erilaisia ohjelmistokomponentteja, joiden avulla myös järjestelmän kehitystä voidaan yksinkertaistaa ja nopeuttaa.

4.3.1. Konttiteknologia

Standardoidut tavarankuljetussäiliöt, rahtikontit, mullistivat rahtitavaraliikenteen yleistyessään, sillä niiden avulla rahdin kuljettajan ei tarvitse välittää kuljetettavasta tavarasta, vaan keskittyä vain rahtikonttien kuljetukseen. Lisäksi konttien avulla voidaan kuljettaa suuria määriä rahtia, sillä kontteja voidaan pinota ja liikutella nopeasti. Rahtikonteille löytyy analogia tietotekniikan puolelta, ja tällaisten konttien avulla voidaan saavuttaa samankaltaisia tehokkuusetuja.

Ohjelmistojen ja etenkin järjestelmien jakelussa ongelmana on suorituksen ajoympäristöriippuvuudet. Alustajärjestelmän asettaminen juuri vaadittuun konfiguraatioon, jotta ajettava ohjelmisto toimisi oikein, voi olla erittäin vaikeaa ja hidasta. Ajoympäristön konfigurointitarvetta voidaan vähentää liittämällä ohjelmiston jakelun yhteyteen myös itse tarvittava ajoympäristö riippuvuuksineen valmiiksi oikein konfiguroituna. Tällaista valmista ajoympäristöohjelmisto -kokonaisuutta kutsutaan kontiksi.

Teknisestä näkökulmasta tällainen tietotekninen kontti on eristetty ympäristö, jossa voidaan suorittaa muusta käyttöjärjestelmästä ja muista konteista riippumattomia prosesseja. Kontti ei kuitenkaan ole virtualisointiratkaisu vaan toimii konttia suorittavan käyttöjärjestelmän ytimen päällä. Koska kontti ei saa alustalta käyttöönsä kuin käyttöjärjestelmäytimen, tulee sen itse sisältää kaikki oman suorituksensa kannalta tarvittavat osat järjestelmäpalveluista alkaen.

Tarkkaan ottaen kontti on konttikuvan (engl. *image*) ajoaikainen ilmentymä, mutta usein puhuttaessa konteista, tarkoitetaan tosiasiassa konttikuvia. Konttikuva sisältää kontin tiedostot ja suoritettavan sisällön erityisinä muuttumattomina päällekkäisinä kerroksina. Kerrokset mahdollistavat konttiteknologian helpon käytön siitä huolimatta, että kontit ovat usein sisällöltään monimutkaisia. Konttikuvia on valmiina saatavilla esimerkiksi erilaisille Linux-jakeluille, ja ohjelmiston sisältävä kerros voidaan yksinkertaisesti lisätä tällaisen Linux-kerroksen päälle.

Sen lisäksi, että kontit ovat rahtianalogiansa tavoin rakenteeltaan standardoituja ja helposti siirrettäviä, ne ovat kevyitä. Kontin tarvitsee sisältää vain sen itsensä suorittamiseen tarvittavia ohjelmia ja tiedostoja, mikä tekee siitä pienikokoisen sekä minimoi muuhun kuin itse kohdeohjelman suoritukseen käytettävät resurssit. Tehokkuuden vuoksi kontteja voidaan suorittaa merkittäviä määriä yhdellä tietokoneella ja niiden käynnistäminen ja sammuttaminen on kevyttä ja nopeaa. Kontti on yleensä sidottu tiettyyn prosessoriarkkitehtuuriin ja tiettyyn käyttöjärjestelmäytimeen, mutta muilta osin se on alustariippumaton. [36]

4.3.2. Orkestrointi

Harva järjestelmä koostuu yhdestä ainoasta kontista, ja etenkin konttien avulla rakennetut hajautetut järjestelmät voivat sisältää suuria määriä kontteja. Kun kontti keskittyy yhden asian hoitamiseen, voidaan järjestelmän horisontaalinen skaalautuminen hoitaa konttien avulla kasvattamalla näiden konttien määrää. Kun järjestelmä koostuu useasta kontista, on olennaista, että kontit voivat kommunikoida keskenään. Kontit eivät voi itse hoitaa tällaisen useamman kontin muodostaman kokonaisuuden hallintaa, vaan siihen tarvitaan keskitetty ratkaisu.

Konttien hallintajärjestelmä eli orkestraattori (engl. *Orchestrator*) huolehtii, että järjestelmän eri palveluita tarjoavat kontit on käynnistetty ja että ne ovat toimintavalmiita. Orkestraattori huolehtii myös, että konttien välinen kommunikaatio toimii ja että niiden välillä jaetut resurssit ovat konteille saatavilla. Lisäksi orkestraattori huolehtii sekä hajautetun järjestelmän skaalaamisesta monistamalla kontteja että skaalaukseen liittyvistä toimista, kuten nimeämisestä ja kuormantasauksesta. Mikäli järjestelmä koostuu useamman tietokoneen muodostamasta klusterista, on orkestraattorin tehtävinä myös varmistaa näiden klusterin eri osien toimintakyky, ja tarvittaessa jakaa konttien suoritusta klusterin sisällä. [37]

Orkestraattorin avulla voidaan myös seurata järjestelmän tilaa keräämällä logitietoja ja tietoja resurssien käytöstä, päivittää kontteja uudempiin versioihin ja palauttaa vanhoja versioita käyttöön, mikäli järjestelmässä havaitaan ongelmia. Useat orkestraattoriratkaisut tarjoavat mahdollisuuden kuvata järjestelmän rakennetta ja sen toimintaa konfiguraationa erityisissä tekstitiedostoissa. Tällöin järjestelmän rakenteen

ja toiminnan kehitys onnistuu ohjelmakoodin tapaan ja ohjelmistokehityksen työkalut ja prosessit ovat käytettävissä myös itse järjestelmän kehittämiseen. [38]

4.3.3. Kuormantasaus

Järjestelmä tarvitsee kuormantasaimen, jos järjestelmässä on horisontaalisesti skaalattu palvelu. Kuormantasauslaitteisto tai -ohjelmisto huolehtii saapuvien yhteyksien tai pyyntöjen jakamisesta eri palveluille järjestelmässä, tarvittaessa eri tietokoneidenkin välillä. Liikenne välitetään eteenpäin käyttötapaukseen valitulla tavalla, yleensä tavoitellen tasaista kuormitusta järjestelmän hyvän toimintakyvyn takaamiseksi.

Hajautetuissa järjestelmissä kuormantasaus on erityisen tärkeää, sillä palvelun muodostavia yksiköitä voi olla suuri määrä. Kuormantasauksen toteuttavan komponentin, eli kuormantasaimen, on hyvä pitää kirjaa saatavilla olevista palveluyksiköistä, joille kuormaa voi jakaa, sillä niiden toimintakyky tai yksiköiden lukumäärä voi muuttua käytön aikana. Tällainen älykäs kuormantasaus on olennainen osa korkean saatavuuden järjestelmien toteutusta, sillä järjestelmä pitää rakentaa kestämään esimerkiksi laiterikkoja. [39]

Kuorman jakamiseen on olemassa useita erilaisia tapoja käyttötapauksen mukaan. Palvelun yksiköiden kuormituksen seuraamisen lisäksi yleisiä tapoja on esimerkiksi jakaa pyynnöt yksiköille vuorotellen järjestyksessä (engl. *Round-Robin*) täysin satunnaisesti tai sen mukaan montako yhteyttä kyseisellä yksiköllä on sillä hetkellä auki. Mikäli saman asiakkaan kaikki liikenne pitää päätyä samaan yksikköön, on liikenne ohjattava asiakkaan IP-osoitteen (lyhenne sanoista *Internet Protocol*) mukaan tai käytettävä ohjelmallista valintaa, jossa kohdeyksikkö valitaan asiakkaan mukaan muilla tavoin. Lisäksi, jos yksikön muistiin tallennetaan tietoa istunnon tilasta, on yhteyden katketessa hyödyllistä pystyä yhdistämään uudelleen samalle yksikölle. Tämän tyyppistä kuormantasausta, jossa sama asiakas päätyy toistuvasti samalle yksikölle, kutsutaan tahmaiseksi kuormantasaukseksi (engl. *Sticky Session*).

Esimerkiksi tilattomille HTTP-pyynnöille ei ole yleensä merkitystä mihin palveluyksikköön pyyntö ohjataan. Tällöin pyynnöt voidaan jakaa yksiköille satunnaisesti, vuorotellen tai kuormituksen mukaan. Lisäksi tilattomien pyyntöjen kuormantasaus on yksinkertaista toteuttaa dynaamisesti, sillä pyyntöjen kohdeyksikkö voidaan vapaasti valita pyynnön saapumisen yhteydessä.

Sen sijaan tahmainen kuormantasaus voi johtaa epätasaiseen kuormaan järjestelmässä, etenkin, jos käytössä olevien palveluyksiköiden määrä vaihtelee. Palveluyksiköiden lisääminen ei paranna palvelun suorituskykyä heti, sillä tahmainen kuormantasaus ei siirrä palvelun sen hetkisiä käyttäjiä uusille palveluyksiköille, vaan niiden käyttöön tulevat vain uudet käyttäjät. Mikäli yhteys pidetään auki pidemmän aikaa, ei kuormituksen jakaminen onnistu yhtä dynaamisesti, sillä käyttäjän siirto toiselle yksikölle vaatisi yhteyden uudelleen muodostamisen, mikä voi olla teknisesti hankalaa tai sen vaikutukset sovelluksen käyttäjäkokemukseen voivat olla liian suuria. [40]

4.3.4. Rajapinnat

Ohjelmointirajapinta (engl. *Application Programming Interface* eli API) on ohjelmistojen tai ohjelmistojen komponenttien välisen tiedonvaihdon väline. Rajapinta peittää taakseen sitä tarjoavan sovelluksen sisäisen toiminnan, eikä rajapintaa käyttävän ohjelman siksi tarvitse tietää muuta kuin rajapinnan rakenne sitä käyttääkseen. Useat yleisesti käytetyt rajapinnat noudattavat standardeja, jolloin niiden rakenne on ennalta määrätty ja niiden käyttöönotto helppoa.

HTTP-pyyntöihin perustuva tilaton REST (engl. *Representational State Transfer*) on nykyään yksi yleisimmistä rajapinnoista web-ympäristössä. REST toteuttaa tyypillisen CRUD-toiminnallisuuden (engl. *Create*, *Read*, *Update*, *Delete*) HTTP-pyynnöillä, jossa pyynnön tyyppi määritellään HTTP-metodilla, kohderesurssi polulla ja pyynnön sisältö pääasiassa tekstimuotoisena JSON-objektina (engl. *JavaScript Object Notation*) pyynnön mukana. REST-rajapintojen luomiseen, käyttöön, dokumentointiin sekä testaamiseen on olemassa suuri määrä erilaisia työkaluja ja kirjastoja tehden REST-sovellusten kehittämisestä helppoa ja nopeaa. [41]

WebSocket-protokollan päälle on olemassa useita erilaisia rajapintamalleja standardimuotoisen ohjelmointirajapinnan toteuttamiseksi. Erityisen rajapintamallin käyttäminen WebSocket-rajapintojen pohjaksi ei kuitenkaan ole yhtä hyödyllistä kuin HTTP-protokollan kanssa, sillä WebSocket-protokolla ei ole yhtä yleiskäyttöinen kuin HTTP. Selainten sisäänrakennettu WebSocket-toteutus tarjoaa hyvät työkalut yhteyden hallintaan, ja data voi liikkua tekstimuotoisena ja siten esimerkiksi JSON-muotoisten viestien käyttö on helppoa. Tästä syystä WebSocket-protokollaa usein käytetään sovellukselle räätälöidyllä tavalla, eikä tällöin sovellusten välillä ole yhteistä rakennetta tai käyttölogiikkaa. [42]

HTTP-rajapinnat, mukaan lukien REST-rajapinnat, ovat palvelinpuolen toteutukseltaan perinteisesti synkronisia. Kun pyyntö saapuu palvelimelle, palvelu suorittaa logiikan viestin käsittelemiseksi, ja suorituksen lopputuloksena palvelu lähettää vastauksen pyynnön lähettäjälle. Synkroninen palvelu voi käsitellä vain yhtä pyyntöä kerrallaan, ja jos suorituksen aikana palvelimelle saapuu uusi pyyntö, se jää jonoon odottamaan vuoroaan. Synkronisen palvelun heikkoutena on se, että jos palvelu joutuu odottamaan jotain toista palvelua, esimerkiksi tietokantaa, se ei sillä aikaa voi tehdä mitään hyödyllistä, vaikka jonossa olisi muitakin pyyntöjä.

Nykyään on kuitenkin yleistä, että synkroninen palvelu pystyy käsittelemään useampaa pyyntöä samanaikaisesti käyttämällä useita säikeitä [43]. Synkronista palvelua ei voida toteuttaa WebSocket-protokollan kanssa, sillä avattu yhteys on auki koko istunnon ajan. Jos WebSocket-rajapintaa yrittäisi käyttää synkronisesti, jokainen yhteys varaisi itselleen kokonaisen säikeen. Säikeiden hallinta on liian raskasta eikä skaalaudu riittävästi rajapinnan tarpeisiin etenkin, kun huomioidaan, kuinka pienen osan ajasta prosessointia WebSocket-istunnon aikana lopulta tarvitaan.

Rinnakkaisuusongelmat WebSocket-rajapinnassa voidaan ratkaista käyttäen asynkronista palvelua. Asynkroninen palvelu prosessoi jonoa, johon kaikki sisäiset toiminnot työnnetään. Mikäli palvelu joutuu odottamaan jonkin toisen palvelun vastausta, se ei jää odottamaan, vaan jatkaa jonon käsittelyä ja palaa jatkamaan edellistä käsittelyä, kun toiselta palvelulta saapuu vastaus. Asynkroninen palvelu on merkittävästi synkronista suorituskykyisempi sellaisissa tilanteissa, joissa palvelu joutuu odottamaan muita palveluita ja pyyntöjä on kesken useita samaan aikaan.

WebSocket-rajapinnassa asynkronisuus mahdollistaa sen, että tiettyyn yhteyteen liittyvää prosessointia tarvitaan vain silloin kun käsitellään vastaanotettua tai lähtevää viestiä. Koska pääsääntöisesti WebSocket-yhteyden yli siirretään viestejä vain murto-osan siitä ajasta, kun yhteys on auki, voidaan yhteyksiä avata yhdelle palvelun yksikölle huomattavia määriä. Asynkronisuus on siis vaatimus, mikäli WebSocket-rajapinta halutaan skaalautuvaksi. [44]

Istunnon alkaessa asiakasohjelma avaa yhteyden WebSocket-rajapintaan, ja yhteys on auki koko istunnon ajan. WebSocket-rajapintaa käytetään välittämään tietoa kanavista sekä resurssien päivityksistä reaaliajassa. Asiakasohjelma ei lähetä WebSocket-rajapintaan ollenkaan viestejä, vaan rajapinta on puhtaasti viestien vastaanottamista varten. Järjestelmässä kaikki asiakasohjelman tekemät pyynnöt tehdään REST-ohjelmointirajapintaa käyttäen.

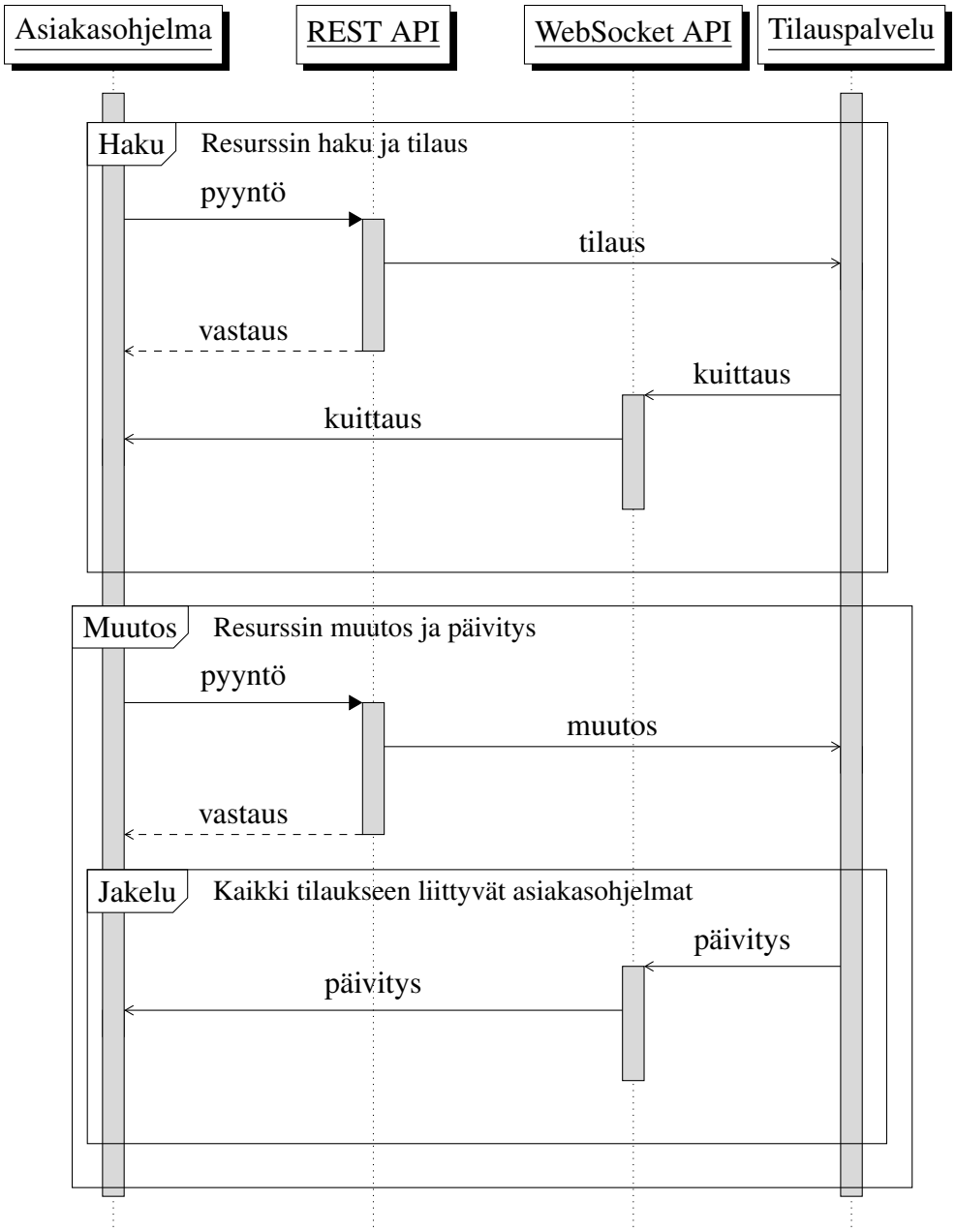
4.3.5. Haut ja muutokset

Asiakasohjelma luo kahdenlaisia pyyntöjä: hakuja ja muutoksia. Hakuja käytetään, kun halutaan hakea tietoja puhtaasti lukevalla tavalla. Muutoksia käytetään, kun halutaan muuttaa tietoja tai suorittaa toimintoja, joilla voi olla sivuvaikutuksia.

Hakupyyntö palauttaa pyydetyn resurssin ja lisäksi aiheuttaa tilauksen resurssiin. Resurssin tilaus liittää asiakasohjelman session resurssiin niin, että kun resurssiin tulee muutos, asiakasohjelma saa päivittyneen version resurssista. Jos resurssin tilaus ei ole asiakasohjelmalle enää ajankohtainen, voi asiakasohjelma peruuttaa tilauksen käyttäen tilauksien hallintaan tarkoitettua rajapintaa.

Muutospyyntö muuttaa resurssia pyydetyllä tavalla ja palauttaa muutetun resurssin. Muutos voi myös olla toiminto, joka ei välttämättä kohdistu mihinkään tiettyyn resurssiin. Muutoksen tai toiminnon tapahtuessa tarkastellaan, mihin resursseihin muutos tai toiminto vaikuttaa. Mikäli näillä resursseilla on tilauksia, niihin lähetetään päivitykset ja sama päivitysviesti saapuu kaikille asiakasohjelmille, jotka ovat mukana tilauksessa. Tilauksen alkamisen jälkeen asiakasohjelman ei enää tarvitse huolehtia resurssin tilasta, vaan voi luottaa siihen, että mikäli resurssiin tulee muutos, palvelin lähettää resurssin päivittyneet tiedot.

Kuva 1 esittää rajapintojen ja asiakasohjelmien kommunikaation sekvenssikaaviona. Hakuvaiheessa WebSocket-rajapinta lähettää kuittauksen uudesta tilauksesta asiakasohjelmalle, jotta asiakasohjelma voi pitää kirjaa avoimista tilauksista. Muutospyynnössä REST-rajapinnan vastauksen ei tarvitse sisältää muuta kuin tieto onnistuneesta pyynnöstä HTTP-tilakoodin muodossa, sillä muutospyynnön aiheuttama päivitys resurssiin lähetetään myös muutoksen tehneelle asiakasohjelmalle.



Kuva 1. Hakuun, muutokseen ja jakeluun liittyvä kommunikaatio asiakasohjelmien, rajapintojen ja tilauspalvelun välillä.

4.3.6. Rajoitukset

Tilauksen tekeminen kaikenlaisiin hakuihin ei välttämättä ole järkevää, tai edes käytännössä mahdollista. Tällaisia hakuja ovat esimerkiksi listaustyyppiset haut käyttäen haku- ja suodatusparametreja. Jotta listaushakuun olisi mahdollista tehdä tilaus, on haku pystyttävä tekemään käänteisesti. Tämä tarkoittaa sitä, että yksittäisestä resurssista on voitava selvittää kaikki haut, joissa resurssi esiintyy, jotta tällaisten hakujen tilauksiin voidaan lähettää tieto resurssin muutoksista.

Esimerkiksi sivutus on mahdollista toteuttaa käänteisesti tallentamalla resurssiin sen järjestysnumero sivutetussa näkymässä, ja käyttämällä tätä järjestysnumeroa resurssin sisältävän sivun löytämiseen. Ongelmaksi muodostuu kuitenkin se, että resurssin luonnin tai poiston yhteydessä järjestysnumero voi muuttua hyvin suuressa määrässä resursseja tehden operaatiosta hyvin raskaan. Jos resurssille tallennetaan järjestysnumeron sijaan sivunumero, on sivunumeroiden muuttaminen järjestyksen muuttuessa hieman kevyempää, sillä vain murto-osa resursseista vaihtaa sivua. Sen sijaan näiden sivua muuttavien resurssien löytäminen voi olla tietokannalle raskasta.

Lista-hakujen tukemisessa on myös se ongelma, että jokaista tuettua erilaista listan järjestelyä varten tarvitaan oma järjestys- tai sivunumeronsa, ja mikäli järjestykseen vaikuttavaa tietoa muutetaan, voi järjestysnumero taas muuttua suuressa määrässä resursseja. Järjestysnumeroperusteiselle sivutukselle vaihtoehtona on kursoriperusteinen sivutus, eli sivutus, jossa sivun sisältämät resurssit haetaan alkaen kohderesurssista, eli kursorista. Kursoriperusteinen sivutus ei ole tehokas ratkaisu myöskään, sillä sen suorittaminen käänteisesti ei ole mahdollista käymättä läpi kaikkia resursseja.

Näistä syistä listahaut katsotaan pitkälti kertakäyttöisiksi, eikä niiden tilauksia tueta. Listan saa kuitenkin tilattua niin, että listan muodostavat resurssit tilataan erikseen. Tällöin resurssit saadaan pidettyä ajan tasalla, mutta kyseisen listanäkymän muodostavien resurssien kokoonpano voi huomaamatta muuttua.

4.4. Arkkitehtuuri

Järjestelmä koostuu tilallisista ja tilattomista palveluista. Palvelun tilallisuus tarkoittaa sitä, onko palvelun yksittäisellä yksiköllä suorituksen ulkopuolista tilaa, jota olisi syytä pitää tallessa, kun yksiköitä luodaan, tuhotaan tai siirretään. Vastaavasti tilaton palvelu voidaan tuhota ja luoda uudelleen ilman, että tällä on vaikutusta järjestelmän toimintaan. Suorituksen aikaisia toimintoja voidaan tällaisessa tilanteessa joutua tekemään uudelleen, kuten esimerkiksi yhteyksien avausta tai muistiin tallennettujen tietojen uudelleenlaskemista, mutta nämä eivät vaikuta järjestelmän loogiseen toimintaan, vaan korkeintaan aiheuttavat tilapäistä lisäkuormitusta. Tilattomien palvelujen yksittäistä yksikköä ei voi erottaa muista yksiköistä palvelun ulkopuolelta, sillä tilattoman palvelun edessä on aina kyseiselle palvelulle tarkoitettu kuormantasain, joka jakaa pyynnöt tasaisesti eri yksiköille. Tilattomien palveluiden horisontaalinen skaalaaminen on erittäin helppoa, sillä siihen riittää se, että palveluiden lukumäärää kasvatetaan ja uudet palveluyksiköt rekisteröidään kuormantasaimelle.

Tilallisia palveluita ovat esimerkiksi tietokantapalvelut, sillä tietokannan on säilytettävä tietonsa siirron tai uudelleenkäynnistyksen jälkeenkin. Tällaisia tilallisia palveluita ei myöskään voi vain skaalata niiden lukumäärää lisäämällä, sillä palvelun yksittäiset yksiköt tarjoaisivat silloin omia tietojaan muista yksiköistä riippumatta, eikä tällaista kokonaisuutta voisi käyttää mielekkäästi. Tilallisen palvelun, kuten tietokannan, skaalaamiseen yleensä vaaditaan, että palvelun yksittäiset yksiköt voivat kommunikoida keskenään, ja joko koko tietokanta on jaettu kaikille identtisesti tai tallennettu data on jaettu osiin ja eri yksiköille erityisten sääntöjen mukaan.

Tilallisen palvelun skaalaaminen on siis paljon tilatonta haastavampaa, ja käytännön toteutus on riippuvainen käytetystä teknologiasta ja ohjelmistosta. Myöskään

kuormantasausta ei voida käyttää tilallisille palveluille, sillä tilallisen palvelun yksiköt eivät toimi keskenään identtisesti tai sisällä samoja tietoja toistensa kanssa. Tilallisen palvelun yksiköt on siis mahdollista erottaa toisistaan palvelun ulkopuolelta.

4.4.1. Skaalautuvuus

Järjestelmä on suunniteltu horisontaalisesti skaalautuvaksi hajautetuksi järjestelmäksi. Koska tavoitteena on pystyä käsittelemään mahdollisimman suuri määrä pyyntöjä, järjestelmän osia on kyettävä skaalaamaan tarvittaessa suorituskyvyn parantamiseksi. Tilallisten palveluiden horisontaaliseen skaalaamiseen on yleensä kaksi tapaa: replikointi ja jakaminen. Replikoinnissa jokainen yksikkö sisältää samat tiedot ja toimii identtisesti. Replikoinnissa suorituskyky lukuoperaatioissa kasvaa, kun yksiköitä lisätään, mutta suorituskyky kirjoitusoperaatioissa ei kasva ja voi jopa laskea. Yleensä replikoinnin tarkoituksena on maksimoida järjestelmän vikasietoisuus ja saavutettavuus, sillä järjestelmä voi jatkaa normaalia toimintaansa siitä huolimatta, että yksiköitä, eli replikoita, on välillä pois käytöstä.

Jakamisessa jokainen yksikkö sisältää vain osan palvelun resursseista, ja jakaminen tapahtuu erityisten sääntöjen perusteella. Resurssien jakamisesta päättää joko palvelu itsenäisesti omien sisäisten sääntöjensä mukaan tai palvelua käyttävät ohjelmat sisältävät itse säännöt, joiden perusteella valitaan mihin yksikköön pyyntö lähetetään. Jakamisen avulla lukuoperaatioiden suorituskyvyn lisäksi voidaan kasvattaa myös kirjoitusoperaatioiden suorituskykyä.

Jakamisella toteutetun skaalaamisen ongelmaksi muodostuu sellaiset pyynnöt, joihin liittyvä tieto sijaitsee useammassa yksikössä. Tällainen on erityisesti ongelma tietokannoissa, koska tietokantahaut voivat olla erityisen monimutkaisia, ja tietojen hakeminen useammalta yksiköltä voi olla hankalaa ja hidasta. Sen sijaan esimerkiksi välimuistipalvelun ja jonopalvelun jakaminen on yleensä helppoa, sillä pyyntö voi kohdistua vain yhteen resurssiin kerrallaan, eikä yksiköiden siten tarvitse kommunikoida keskenään.

4.4.2. Rajapintapalvelut

Rajapintapalvelu, joka tässä järjestelmässä on toteutettu käyttäen Django REST framework -ohjelmistokehystä, tarjoaa REST-rajapinnan, jonka avulla järjestelmän ulkopuoliset toimijat voivat hakea ja muokata järjestelmän resursseja. Jokainen REST-rajapinnan HTTP-pyyntö on itsenäinen kokonaisuus, eikä rajapintaa käyttävälle ulkopuoliselle toimijalle ole merkitystä sillä, mikä yksikkö pyynnön käsittelee. Rajapintapalvelu tarjoaa myös WebSocket-rajapinnan, jonka avulla järjestelmän ulkopuoliset toimijat voivat tilata järjestelmän resursseja ja vastaanottaa ilmoituksia tilattujen resurssien tilamuutoksista.

WebSocket-rajapinnassa avoimeen WebSocket-yhteyteen liittyy tila, joka on saatavilla vain siinä yksikössä, jonne yhteys on alun perin muodostettu. Tämä ei kuitenkaan ole ongelma, sillä sama yhteys pysyy auki koko istunnon ajan, eikä ulkopuolisen toimijan näkökulmasta ole merkitystä mille yksikölle yhteys avataan. Jos kyseinen yksikkö poistetaan käytöstä, avoin yhteys katkaistaan ja asiakasohjelman on

muodostettava uusi yhteys. Uuden yhteyden muodostamisen jälkeen asiakasohjelman on tilattava haluamansa resurssit uudelleen, mutta tämän jälkeen toiminta jatkuu normaalisti.

Rajapintapalvelun yksiköiden ei tarvitse kommunikoida muille yksiköille, vaan yhteinen tila on jaettu käyttäen tietokantaa ja välimuistipalvelua. Mikäli rajapintapalvelun täytyy pystyä vastaanottamaan reaaliaikaisia komentoja, voidaan se toteuttaa viestipalvelun avulla, jota rajapintapalvelu käyttää jo muutenkin WebSocket-viestien välittämiseen. Koska rajapintapalvelu on tilaton palvelu, se skaalautuu yksinkertaisesti muuttamalla palveluyksiköiden lukumäärää. Kuormantasaimen ei tarvitse pystyä erottamaan yksittäisiä palveluyksiköitä, vaan se voi käyttää kaikkia palveluyksiköitä mielivaltaisesti sekä REST- että WebSocket-rajapintoja käyttävien pyyntöjen ohjaamiseen.

4.4.3. Tietokannat

Järjestelmän tietoja hallinnoi relaatiotietokannan hallintajärjestelmä PostgreSQL. Tietokanta on tilallinen palvelu, jota käyttävät järjestelmässä ainoastaan rajapintapalvelu ja taustatyöpalvelu. Koska tietokannan käyttö järjestelmässä on sen suunnittelufilosofian seurauksena kirjoitusintensiivistä, tietokannan replikointi ei ole järkevää. Tietokannan skaalaaminen jakamalla voisi lisätä suorituskykyä kirjoitusoperaatioiden suhteen, mutta sen käyttöönotto lisäisi rajapintapalvelun toteutuksen monimutkaisuutta niin paljon, että se on toistaiseksi jätetty toteuttamatta. Mikäli tietokanta muodostuu pullonkaulaksi järjestelmän suorituskyvyn kannalta, voidaan tietokannan suorituskykyä optimoida esimerkiksi lisäämällä välimuistin käyttöä, parantamalla tietokantayhteyksien hallintaa, vähentämällä tehtyjen kyselyiden määrää sekä parantamalla tietokannan rakennetta ja tietokantaindeksien käyttöä.

4.4.4. Viestipalvelut

Viestipalvelu on vastuussa järjestelmän sisäisestä viestien välityksestä. Viestipalvelua käyttävät rajapintapalvelu sekä taustatyöpalvelu, välittäessään WebSocket-viestejä asiakasohjelmille. Kun käyttäjälle halutaan lähettää reaaliaikainen viesti, viestin lähettäjä voi olla eri palveluyksikkö kuin se, jolle kohteena olevan asiakasohjelman WebSocket-yhteys on avattu. Tällöin tarvitaan mekanismi, jolla viesti saadaan välitettyä oikealle palveluyksikölle ja sitä kautta asiakasohjelmalle. Viestipalvelun avulla rajapintapalvelun yksiköille voidaan myös välittää niiden toimintaan vaikuttavia komentoja reaaliaikaisesti.

Viestipalvelun toteuttaa Redis, jonka tarjoaman julkaisu–tilaus-toiminnallisuuden (engl. *Publish–Subscribe*) avulla voidaan muodostaa viestikanavia, joita rajapintapalvelu ja taustatyöpalvelu tilaavat tarpeidensa mukaan. Viestipalvelu on skaalattu jakamalla, eivätkä yksiköt kommunikoi keskenään. Viestikanavat on jaettu viestipalveluyksiköiden välillä tiettyjen sääntöjen mukaan, ja viestipalvelua käyttävät palvelut käyttävät näitä sääntöjä määrittämään, miltä yksiköltä tilata mikäkin kanava.

4.4.5. Välimuistipalvelu

Välimuistipalvelu on toteutettu samalla Redis-ohjelmistolla kuin viestipalvelu, mutta se voidaan katsoa loogisesti eri palveluksi. Välimuistipalvelun tarkoitus on tallentaa muistiin avain-arvo-pareja, joita luetaan ja kirjoitetaan annetun avaimen perusteella, ja jotka ovat saatavilla kaikkialla järjestelmän sisällä. Välimuistipalvelua käytetään myös taustatyöpalvelun töiden tilojen ja lopputulosten tallentamiseen, jotta tiedot ovat saatavilla taustatyöpalvelua käyttävälle palvelulle työn valmistuttua. Välimuistipalvelu on periaatteessa tilaton palvelu, mutta koska sitä käyttävien palvelujen on pystyttävä erottamaan yksiköt toisistaan, sitä käytetään järjestelmässä tilallisena palveluna.

Luku- sekä kirjoitusoperaatiot ovat välimuistipalvelussa erittäin kevyitä, ja tietokantaan verrattuna niitä voidaan suorittaa huomattavasti enemmän. Välimuistipalvelun tiedot eivät ole pysyvässä muistissa, eli tietoja menetetään palvelun sammuttamisen yhteydessä, eikä välimuistipalvelua tule käyttää tiedon ensisijaisena tallennuspaikkana. Välimuistipalvelu kuitenkin voidaan konfiguroida tallentamaan tiedot levylle aika ajoin, jolloin suurin osa tiedoista selviää kaatumisesta tai uudelleenkäynnistämisestä.

Jokainen Redis-yksikkö hoitaa sekä välimuistipalvelun että viestipalvelun rooleja. Jakamalla skaalatun välimuistipalvelun yksiköt toimivat samalla tavalla kuin jakamalla skaalatun viestipalvelun yksiköt, ja kanavan nimen sijasta käytetään avainta määrittämään se välimuistipalvelun yksikkö, josta haluttu tieto löytyy. Aivan kuten viestipalvelussakin, välimuistipalvelun yksiköt eivät kommunikoi keskenään.

Välimuistipalvelun tarkoitus on nopeuttaa pyyntöjen käsittelyä tallentamalla yhteiseen muistiin useasti käytettyjä tietoja tai sellaisia tietoja, joiden laskeminen on raskasta. Tällä tavoin voidaan vähentää muun muassa tietokantaan kohdistuvien toistuvien pyyntöjen määrää ja siten vähentää tietokannan käyttöä. Lisäksi REST-rajapinnan tilattomuuden vuoksi on mahdollista tallentaa REST-hakujen vastauksia sellaisenaan välimuistiin, jolloin tällaisten hakujen käsittelyä voidaan keventää ja nopeuttaa huomattavasti.

Yleensä välimuistiin tallennettuihin tietoihin sidotaan vanhenemisaika, jonka jälkeen tieto poistetaan välimuistista automaattisesti. Jos välimuistista ei löydy pyydettyä tietoa, lasketaan tieto uudelleen ja tallennetaan se välimuistiin uudella vanhenemisajalla. Tällä tavoin voidaan rajoittaa sitä, kuinka vanhaa tietoa välimuistissa voi olla. Mikäli samaa tietoa käytetään usein, voi vanhenemisaika olla hyvin lyhytkin edelleen saavuttaen merkittävää tehokkuushyötyä.

4.4.6. Muut palvelut

Muita palveluita järjestelmässä ovat jonopalvelu sekä taustatyöpalvelu. Jonopalvelu on toteutettu RabbitMQ-ohjelmistolla, ja sen tarkoitus on tallentaa työkuvauksia jonoihin. Jonopalvelu takaa, että jonoon lisätty työkuvaus käsitellään heti, kun sen käsittelyyn on saatavilla käsittelijä, ja että se käsitellään onnistuneesti ja vain kerran. Työkuvaus voi sisältää myös ajan, jolloin työ on suoritettava.

Jonopalvelu on tilallinen palvelu, sillä se ylläpitää tietoja jonojen työkuvauksista ja niiden käsittelystä ilman, että tietoja menetetään kaatumisen tai uudelleenkäynnistyksen yhteydessä. Jonopalvelu voidaan skaalata horisontaalisesti

jakamalla jonot palveluyksiköiden välillä samaan tapaan kuin viestipalvelulla. Palvelun käyttö järjestelmässä on kuitenkin niin vähäistä, että jonopalvelua ei ole toistaiseksi skaalattu ollenkaan.

Taustatyöpalvelu on Python-pohjainen palvelu, joka suorittaa jonopalvelun jonoista löytyviä työkuvauksia. Se on teknisesti lähes identtinen Django-rajapintapalvelun kanssa, mutta käyttää Celery-ohjelmistoa työjonon käsittelyyn ja Django-resursseja muun muassa tietokantaan ja välimuistipalveluun liittyviin operaatioihin. Käsiteltyään työkuvauksen, taustatyöpalvelu tallentaa työn tilan ja lopputuloksen välimuistipalveluun, ja ilmoittaa jonopalvelulle työn valmistumisesta.

Taustatyöpalvelu on tilaton palvelu, ja sen skaalaaminen on äärimmäisen helppoa. Se ei tarvitse kuormantasainta, sillä taustatyöpalvelun yksiköiden kanssa ei voi kommunikoida suoraan millään tavalla, vaan yksiköt vain odottavat uusia jonoon tulevia työkuvauksia. Järjestelmässä taustatyöpalvelua ei kuitenkaan skaalata erityisen suureksi, sillä työkuvauksia tulee jonoon verrattain harvoin.

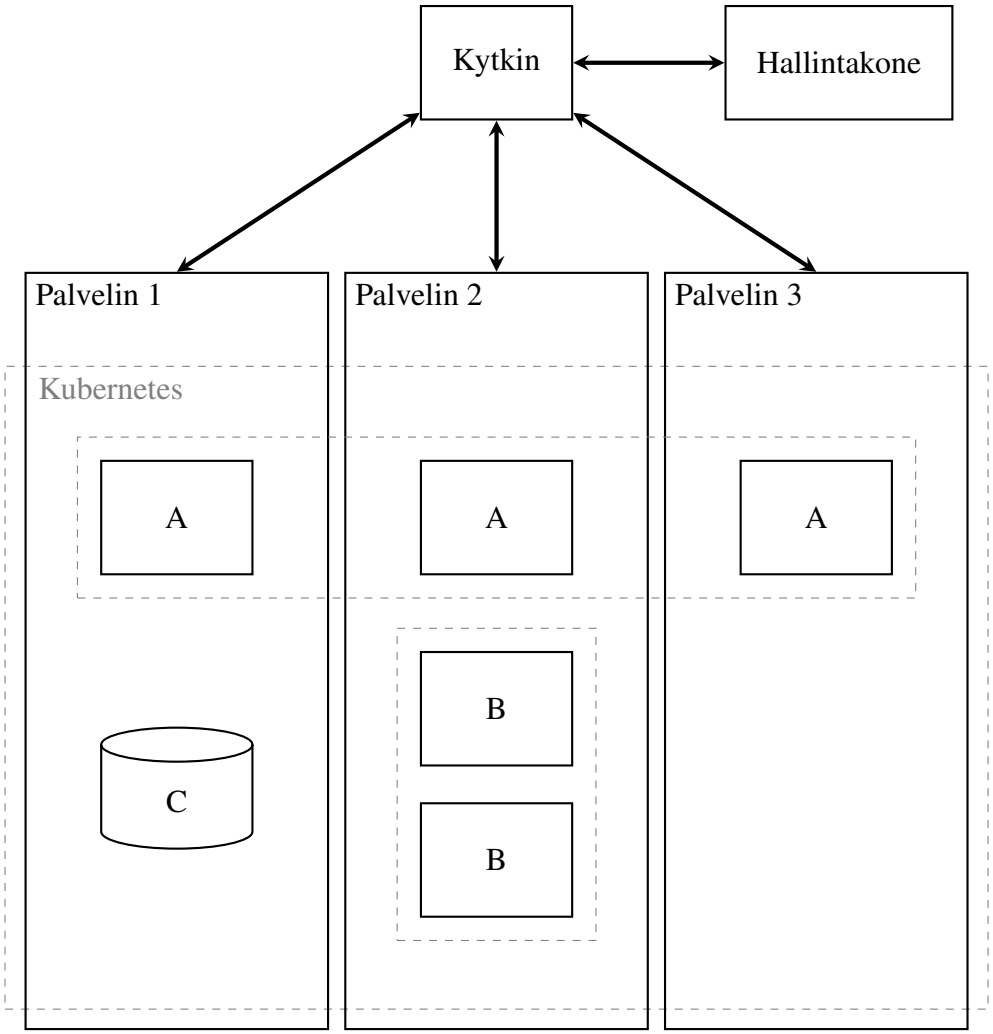
4.4.7. Infrastruktuuri

Järjestelmä on rakennettu toimimaan Kubernetes-klusterissa. Kubernetes on Googlen kehittämä avoimen lähdekoodin konttien hallintajärjestelmä, joka sisältää konttien orkestrointiin ja hajautetun järjestelmän hallintaan tarvittavia työkaluja. Se tarjoaa muun muassa sisäisiä verkkoja palveluiden väliseen kommunikointiin, kuormantasaimia skaalattujen palveluiden käyttöön sekä mahdollistaa useamman fyysisen tietokoneen samanaikaisen käytön. [45]

Tässä työssä järjestelmää suoritetaan kolmen identtisen palvelintietokoneen muodostamassa klusterissa, ja Kubernetes on valittu järjestelmän orkestraattoriksi, koska se yhdistää tietokoneet yhdeksi loogiseksi kokonaisuudeksi. Järjestelmän kehitys helpottuu, sillä palvelut eivät näe eroa sillä, onko tietokoneita yksi vai useampi. Kolme palvelinta on käytössä suorituskyvyn maksimoimiseksi sekä järjestelmän vertikaalisen skaalautumisen ja hajautettavuuden demonstroimiseksi.

Käytössä olevat palvelimet ovat Dell PowerEdge R410 -tietokoneita, joissa on jokaisessa kaksi kappaletta 6-ytimisiä Intel Xeon -suorittimia, 12 gigatavua DDR3 RAM-muistia ja 128 gigatavua SSD-tallennustilaa. Intel Hyper-threading-tekniikka tuplaa loogisten ytimien määrän verrattuna fyysisiin ytimiin, joita on yhteensä 36. Klusterilla on siis kokonaisuudessaan käytössä 72 loogista CPU-ydintä ja 36 gigatavua RAM-muistia.

Kuvassa 2 on esitetty järjestelmän palvelininfrastruktuuri. Verkkokerroksessa kolme palvelinta kommunikoi toisilleen kytkimen kautta. Kubernetes-kerros luo palveluiden käyttöön koneiden välisen yhteisen verkon, jossa järjestelmän sisäinen kommunikaatio tapahtuu. Palvelut eivät näe kytkintä tai muuta fyysisen verkon topologiaa, palvelinten määrää eikä sitä, millä palvelimella eri palvelut sijaitsevat.



Kuva 2. Järjestelmän palvelininfrastruktuuri ja Kubernetes-kerros esimerkkipalveluin.

Esimerkkipalvelu A koostuu kolmesta yksiköstä, joista jokainen sijaitsee eri palvelimella. Orkestraattori on mahdollista määrittää jakamaan palvelun yksiköt eri palvelimille halutulla tavalla, joko tasaisesti kuten palvelu A tai samaan paikkaan kuten esimerkkipalvelu B. Mikäli kyseessä on tilaton palvelu, kuten A ja B tässä esimerkissä, palveluista muodostuu yksi kokonaisuus, jolla on oma kuormantasaimensa. Mikäli A-palvelu tekisi pyynnön B-palvelulle, päätyisi pyyntö satunnaisesti jommallekummalle B-palvelun yksikölle. Sama pätee myös toisin päin, vaikka A-palvelun yksiköt sijaitsevatkin eri palvelimilla.

Tietokantaa esittävä esimerkkipalvelu C on tilallinen palvelu, eikä sitä ole esimerkissä skaalattu. Tietokanta on kuitenkin samalla tavalla saatavilla kaikille esimerkin palveluyksiköille. Mikäli tietokanta kuormittaisi palvelinta huomattavasti, olisi muiden palveluyksiköiden sijoittaminen osin muille palvelimille järkevää. Orkestraattori voidaan konfiguroida tasapainottamaan palvelinten välistä kuormaa siirtelemällä tilattomien palveluiden palveluyksiköitä palvelimelta toiselle, mikäli palvelinten kuormitus on riittävän erilainen.

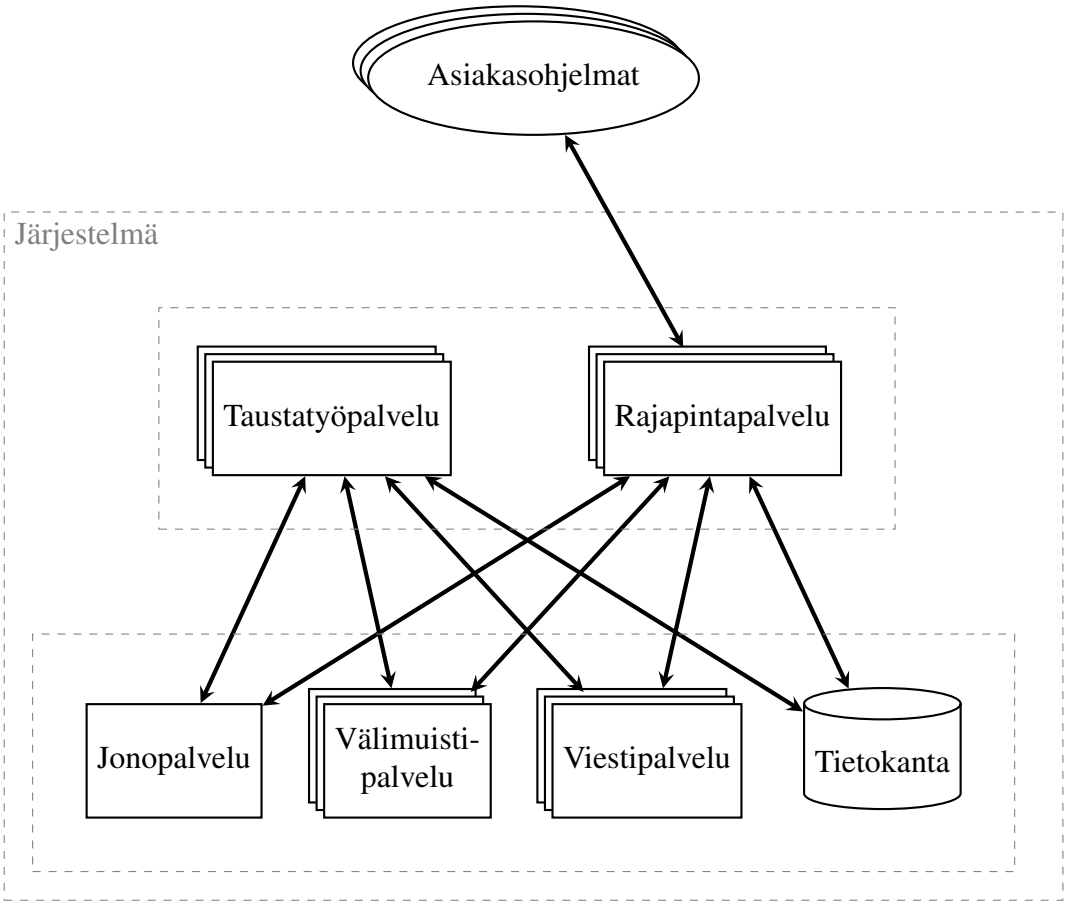
Kubernetes-klusteria ja sen sisältämää kokonaisuutta hallitaan järjestelmän ulkopuoliselta hallintakoneelta. Hallintakoneella on YAML-muotoinen (engl. *YAML*

Ain't Markup Language) kuvaus Kubernetes-klusterin tilasta ja rakenteesta, ja tarvittava ohjelmisto päivittämään klusteri vastaamaan kuvausta, mikäli siihen tehdään muutoksia. Hallintakonetta tarvitaan kuitenkin vain asennus- ja muutosoperaatioihin, ja näiden operaatioiden ulkopuolella järjestelmä kykenee toimimaan täysin ilman hallintakonetta.

4.4.8. Kokonaisuus

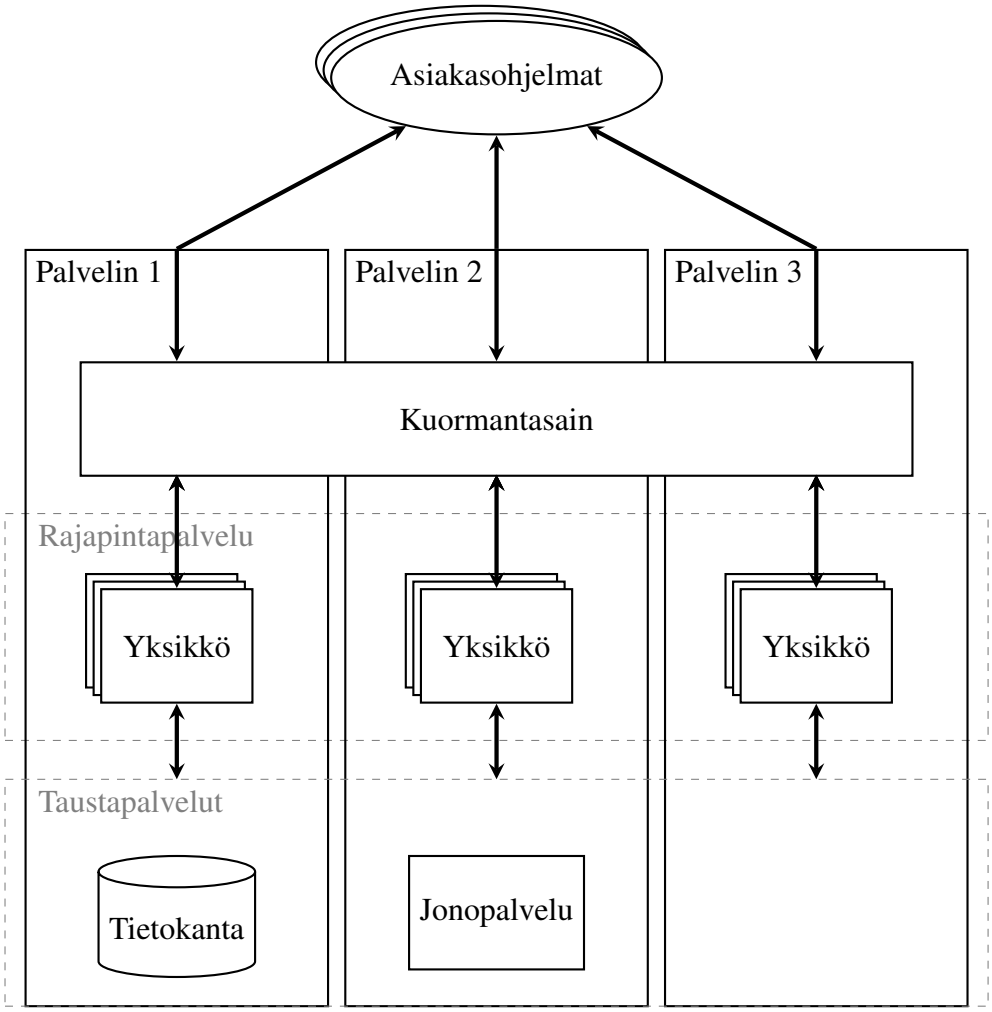
Järjestelmä koostuu kuudesta palvelusta, jotka muodostavat kaksi kerrosta. Ylempi kerros, logiikkakerros, koostuu tilattomista palveluista, eli rajapinta- ja taustatyöpalvelusta. Alempi kerros, taustapalvelukerros, koostuu välimuistipalvelusta, jonopalvelusta, viestipalvelusta sekä tietokannasta. Taustapalvelukerroksen palvelut ovat kaikki tilallisia, ja niistä vain Redis, joka kattaa viesti- ja välimuistipalvelun, on skaalattu.

Kuvassa 3 esitellään järjestelmän arkkitehtuuri, palvelukerrokset sekä palveluiden väliset yhteydet. Järjestelmän ulkopuolella ovat asiakasohjelmat, jotka kommunikoivat rajapintapalvelun kanssa. Rajapintapalvelu on järjestelmän ainoa palvelu, joka on saatavilla järjestelmän ulkopuolelta.



Kuva 3. Järjestelmän arkkitehtuuri ja palveluiden väliset yhteydet. Arkkitehtuuri voidaan jakaa logiikka- ja taustapalvelukerroksiin (ylempi ja alempi).

Kuva 4 esittää järjestelmän palveluiden jakautumisen palvelinten välille. Kuvassa näkyvä kuormantasain esittää rajapintapalvelun edessä olevaa kuormantasainta, jonka läpi asiakasohjelmilta saapuvat pyynnöt kulkevat. Jokainen palvelin avaa ulospäin näkyvän portin verkkoon, josta rajapintapalvelua on mahdollista kutsua. Kaikki tähän porttiin saapuvat pyynnöt, riippumatta palvelimesta, käsitellään samalla kuormantasaimella, ja pyyntö voi päätyä käsiteltäväksi millä tahansa palvelimella sijaitsevalle rajapintapalvelun yksikölle.



Kuva 4. Järjestelmän palveluiden jakautuminen palvelimille sekä viestien välittyminen asiakasohjelmien, kuormantasaimen ja taustapalveluiden välillä.

Kuvassa taustapalvelut on kuvattu yhtenä kokonaisuutena, jossa viestipalvelu ja välimuistipalvelu on jätetty kokonaan piirtämättä. Jokaisella palvelimella on yksi viesti- ja yksi välimuistipalveluyksikkö. Myöskään taustatyöpalvelua ei ole piirretty tähän kuvaan. Tietokantapalvelu ei ole skaalattu, ja ainoa tietokantayksikkö sijaitsee palvelimella 1. Sama pätee myös jonopalveluun, jonka ainoa yksikkö sijaitsee palvelimella 2.

Rajapintapalvelun yksiköiden lukumäärä riippuu järjestelmän kuormituksesta, asiakasohjelmien lukumäärästä ja niiden käyttäytymisestä, eikä parasta lukumäärää voi yksiselitteisesti määrittää. Orkestraattori voidaan asettaa automaattisesti muuttamaan

rajapintayksiköiden lukumäärä järjestelmän käytön perusteella. Paras yksiköiden lukumäärä tullaan määrittämään testausvaiheessa, jossa järjestelmän suorituskykyä mitataan eri määrillä rajapintayksiköitä.

5. TOTEUTUS

Palvelinjärjestelmän suunnittelun jälkeen voidaan siirtyä järjestelmän toteutukseen. Toteutuksessa järjestelmän vaatimukset pyritään täyttämään valittujen teknologioiden avulla rakentaen järjestelmän eri osat, ja yhdistäen ne toimivaksi kokonaisuudeksi arkkitehtuurin mukaisesti. Toteutuksen toimivuuden ja vaatimusten täyttymisen varmistamiseksi järjestelmälle rakennetaan myös testit sekä testien suorittamiseen tarkoitettu testiympäristö.

5.1. Teknologiat

Palvelinjärjestelmän kaikki ohjelmoitavat osat ja suuri osa järjestelmän ulkopuolisista ohjelmistoriippuvuuksista on toteutettu käyttäen Pythonia. Ohjelmoitavan logiikan toteuttamisen helpottamiseksi järjestelmässä käytetään Django-ohjelmistokehystä, joka tarjoaa valmiita ominaisuuksia yleisimpien web-ympäristön ominaisuuksien toteuttamiseen. Järjestelmän tiedon tallentamiseen käytetään PostgreSQL:n avulla toteutettua relaatiotietokantaa.

5.1.1. Python

Python on ohjelmointikieli, jota Guido van Rossum lähti kehittämään ABC-ohjelmointikielen seuraajaksi 80-luvun lopussa. Python suunniteltiin C-ohjelmoijien tueksi sellaisiin tehtäviin, joihin C-ohjelman kirjoitus ei ollut tehokasta esimerkiksi koodin kertakäyttöisyyden vuoksi ja joihin nopeaa shell-skriptausta ei voitu helposti soveltaa abstraktien tietotyyppien puutteen vuoksi [46]. Nykyään, noin 30 vuoden kehitystyön jälkeen, Python nähdään erittäin monipuolisena ja helppokäyttöisenä kielenä. Python soveltuu hyvin ensimmäiseksi ohjelmointikieleksi pääosin sen helpon syntaksin takia. Helpolla syntaksilla tarkoitetaan sitä, että Python ei vaadi yhtä paljon koodin kirjoittamista kuin muut kielet, ja sen syntaksi muistuttaa enemmän englannin kieltä kuin muut. [47]

Kielenä Python on korkeatasoinen, yleiskäyttöinen, interaktiivinen sekä olio-ohjelmointiin suuntautunut, ja kääntämisen sijaan sitä suoritetaan tulkin avulla. Pythonin yksinkertainen syntaksi, dynaaminen tyypitys ja laaja sisäänrakennettu kirjasto korkeatasoisia tietorakenteita ja moduuleita tekee ohjelmointityöstä nopeaa ja tehokasta. Usein sitä kehutaankin juuri paremman tuottavuuden, laadun sekä ylläpidettävyyden mahdollistamisesta ohjelmistolle. Sen käyttö on ilmaista myös kaupallisiin tarkoituksiin, ja sitä voidaan hyödyntää oikeastaan jokaisella nykyaikaisella tietokoneella [48]. Pythonin käyttö on laajentunut voimakkaasti ja se on yksi laajakäyttöisimmistä ohjelmointikielistä. [49]

Pythonia käytetään maailmanlaajuisesti erilaisten ohjelmistojen lisäksi koneoppimisessa, teollisuudessa, tieteellisessä tutkimuksessa ja koulutustarkoituksessa. Vaikka Python on laajakäyttöinen, sitä ei kuitenkaan usein käytetä mobiilisovelluksissa. Tämä johtuu siitä, että tulkki vaatii paljon muistia ja suoritinaikaa, ja mobiilisovelluksissa pyritään mahdollisimman vähäiseen resurssien käyttöön. [49]

5.1.2. Django

Django on verkossa toimivien web-sovellusten toteuttamiseen suunniteltu ohjelmistokehys. Se on Python-pohjainen ilmainen avoimen lähdekoodin projekti, joka on yli 30 vuoden kehityksen aikana kasvanut maailman käytetyimpien ohjelmistokehysten joukkoon. Django sisältää erilaisia järjestelmän kehittämistä helpottavia ominaisuuksia, kuten muun muassa tietokannan käytön olio-ohjelmoinnin kaltaisesti mahdollistavan ORM-kerroksen (engl. *Object-Relational Mapping*), automaattiset tietokantamigraatiot, tuen kirjautumiselle ja käyttöoikeuksille, käyttövalmiita tietoturvaominaisuuksia, sekä sisäänrakennetun admin-käyttöliittymän, joka mahdollistaa järjestelmän tietojen hallinnan selaimessa. [50]

Sisäinen rakenne Djangossa on tehty helposti muokattavaksi ja laajennettavaksi, joten sen avulla voidaan helposti luoda laajasti erilaisia vaatimuksia täyttäviä sovelluksia. Djangoon on saatavilla valikoima erilaisia lisäosia, joilla voidaan laajentaa tai muuttaa Djangon toimintaa tarpeen mukaan. Lisäksi Djangoon voidaan yhdistää, suoraan tai lisäosien avulla, tietokantojen lisäksi muitakin taustapalveluita, kuten vaikkapa tehtäväjonoja, tehtävien ajastusta, välimuisteja ja tiedostojen tallennusta. [50]

Djangon voidaan katsoa noudattavan MVC-arkkitehtuurimallia (engl. *Model-View-Controller*), joka jakaa sovelluslogiikan kolmeen osaan; malliin, näkymään ja käsittelijään. Malli edustaa dataa ja tietokantaa, näkymä määrittelee, miten tietoja esitetään ja käsittelijä ohjaa pyynnöt oikeaan näkymään. Oletuksena Django-näkymät ovat sapluunoita HTML-sivuille, mutta näkymiä on mahdollista muokata palauttamaan dataa myös muussa muodossa. [51]

Järjestelmä käyttää Djangon lisäosaa, Django REST framework:ia, jonka avulla Djangon näkymistä voidaan luoda REST-rajapinta. Tilattoman REST-rajapinnan toteuttava Django-sovellus on helposti horisontaalisesti skaalattava, ja Djangon integrointi välimuistipalveluun mahdollistaa helpon tiedon jakamisen Django-yksiköiden välillä [52]. Järjestelmässä on käytössä myös Djangoon saatavilla oleva virallinen Channels-lisäosa, jonka avulla Djangon saa toimimaan asynkronisesti ja tukemaan WebSocket-rajapintojen toteutusta. [53]

5.1.3. Tietokanta ja ORM

PostgreSQL on ilmainen avoimen lähdekoodin relaatiotietokannan hallintajärjestelmä, jossa painotetaan luotettavuutta, SQL-standardin noudattamista ja laajennettavuutta. Se soveltuu myös hajautettujen järjestelmien tietokannaksi kehittyneiden skaalautuvuusominaisuuksiensa ansiosta. [54]

Sen sijaan, että järjestelmä manipuloisi relaatiotietokantaa perinteiseen tapaan SQL-kyselyiden avulla, järjestelmä käyttää Djangon sisältämää ORM-tekniikkaa. ORM-tekniikalla yhdistetään tietokantarakenteet ohjelmallisesti määritettyihin abstrakteihin tietorakenteisiin eli malliluokkiin. Tällainen abstraktio mahdollistaa tietokannan resurssien käyttämisen olioina sen sijaan, että tauluja käsiteltäisiin riveinä ja sarakkeina. Malliluokilla voidaan kuvata esimerkiksi eri mallien väliset relaatiot, jolloin ORM pystyy olion tietoja haettaessa automaattisesti tekemään taulujen välisen liitoksen (engl. *join*) ja relaatio näyttäytyy olion yhtenä ominaisuutena. [55]

ORM-tekniikan avulla SQL-kerros voidaan pitkälti piilottaa tietokannan rakenteita kuvaavien malliluokkien taakse, tehostaen ohjelmistokehitystä, kun SQL-kerroksen ylläpitoon ei kulu enää aikaa [55]. Django-ohjelmistokehys sisältää kehittyneen ORM-toteutuksen, jonka taustalla voidaan käyttää useita erilaisia tietokantoja. Järjestelmässä Django-ORM on konfiguroitu käyttämään PostgreSQL-tietokantaa.

5.2. Rajapinnat

Järjestelmä sisältää kaksi erilaista rajapintaa. Näistä ensimmäinen, Djangolla toteutettu REST-rajapinta, on pääasiassa resurssien hakuun, luontiin, muokkaamiseen ja poistoon tarkoitettu rajapinta. REST-rajapinnan kautta on myös mahdollista suorittaa resursseihin liittymättömiä toimintoja, kuten käyttäjän kirjautuminen ja erilaisten käskyjen antaminen.

Toinen rajapinta on WebSocket-protokollan avulla toteutettu kaksisuuntainen asynkroninen viestintärajapinta, joka mahdollistaa reaaliaikaisen tiedonvaihdon asiakasohjelman ja palvelimen välillä. Asiakasohjelma voi tilata tiettyihin resursseihin liittyviä kanavia ja vastaanottaa niiden tilaan liittyviä muutoksia heti, kun ne tapahtuvat järjestelmässä. WebSocket-rajapinta välittää myös muita yleisiä järjestelmään ja käyttäjään liittyviä reaaliaikaisia muutoksia.

Käyttäjän kirjautuessa järjestelmään ja aloittaessa uuden istunnon, asiakasohjelma avaa yhteyden WebSocket-rajapintaan. Tämän jälkeen se voi käyttää REST-rajapintaa resurssien hakemiseen ja manipuloimiseen, samalla tilaten WebSocket-rajapinnasta reaaliaikaisia päivityksiä niistä resursseista, joiden tilan seuraaminen on asiakasohjelman toiminnan kannalta hyödyllistä. Esimerkkejä tällaisista ovat resurssit, joiden tiedot ovat käyttöliittymässä suoraan käyttäjän näkyvillä tai joiden tila vaikuttaa olennaisesti siihen, mitä käyttäjä voi tehdä tai nähdä käyttöliittymässä.

5.2.1. Kanavat

Koska WebSocket-yhteys on tilallinen ja yhteys pysyy auki samalle rajapintayksikölle koko istunnon ajan, tietyn asiakasohjelman kaikki kanavatilaukset ovat yhden rajapintayksikön hallussa. Kun asiakasohjelma tilaa kanavan tai kanavan tilaus aiheutuu sivuvaikutuksena jostain muusta toiminnosta, tulee tilauskäsky välittää rajapintayksikölle, johon asiakasohjelma on avannut WebSocket-yhteytensä. Kyseisen rajapintayksikön löytäminen ei kuitenkaan ole yksinkertaista.

Ongelma ilmenee, mikäli kanava halutaan tilata sivuvaikutuksena esimerkiksi resurssien hakemisen yhteydessä. Jos haku tehdään REST-rajapinnan kautta ja käytössä on satunnaisuuteen perustuva kuormantasaus, saapuva pyyntö voi päätyä mille tahansa useasta palveluyksiköstä. Tällöin rajapintakutsu saatetaan käsitellä eri yksikössä, kuin missä asiakasohjelma on avannut WebSocket-yhteytensä, eikä kanavan tilausta voida tehdä suoraan.

Yksinkertainen ratkaisu tähän ongelmaan on se, että asiakasohjelma tilaa kanavan itse käyttäen aina WebSocket-yhteyttä. Tällöin tilauskäsky välittyy suoraan asiakasohjelmalta oikealle rajapintayksikölle. Tämä kuitenkin vaikeuttaa kanavien hallintaa asiakasohjelmassa ja saattaa vaarantaa resurssin tilan synkronoinnin,

koska kanavan tilaus ja resurssin haku tapahtuvat eri ajanhetkillä. Myös resurssin käyttöoikeudet täytyy tarkistaa kahdesti, koska haku ja tilaus eivät tässä tapauksessa liity toisiinsa. Toisaalta tämä ratkaisu on hyvä, jos järjestelmää käyttävät ulkopuoliset integraatiot tai muut asiakasohjelmat, jotka ovat kiinnostuneita vain resurssin hausta ja muutospyynnöistä reaaliaikaisen tilan sijaan.

Toinen mahdollinen ratkaisu olisi käyttää avointa WebSocket-yhteyttä kaikkien pyyntöjen tekemiseen. Tällöin kanavatilausten tekeminen olisi helppoa, sillä jokainen pyyntö käsitellään jo valmiiksi oikeassa rajapintayksikössä. Pyyntöperusteinen HTTP-protokolla helpottaa hakujen tekemistä, koska pyyntöön saadaan aina vastaus. REST-rajapinnan kaltainen rajapinta olisi mahdollista toteuttaa WebSocket-yhteyden avulla, mutta se olisi huomattavasti monimutkaisempi kehittää ja hankalampi käyttää, koska WebSocket-yhteyden päälle pitäisi rakentaa logiikkaa vastausten ja virheiden käsittelyyn sekä rinnakkaisuuden simuloimiseen. Myöskään REST-kehityksen tehostamiseksi kehitettyjen työkalujen käyttö ei olisi enää mahdollista.

Kolmas vaihtoehto olisi käyttää välimuistipalvelua oikean rajapintayksikön löytämiseen. Välimuistipalveluun voitaisiin tallentaa se rajapintayksikkö, johon asiakasohjelma on avannut WebSocket-yhteytensä. Rajapintapalvelulla on pyynnön tapahtuessa käytössään vain kaksi tietoa asiakasohjelman yksilöimiseksi: autentikoinnin kautta saatava käyttäjätunnus ja asiakasohjelman IP-osoite. Jos käyttäjällä on useampi istunto auki samanaikaisesti ja kuormantasaus on valinnut eri palveluyksiköt tarjoamaan WebSocket-yhteyksiä, oikean palveluyksikön löytäminen ei ole mahdollista, sillä sama käyttäjätunnus olisi sidottu useampaan eri palveluyksikköön. Edes IP-osoitteen käyttö ei riitä, koska useampi asiakasohjelma voisi käyttää samaa IP-osoitetta.

Ongelma voidaan ratkaista lähettämällä rajapintakutsujen mukana WebSocket-yhteyteen liitetty yksilöity istuntotunniste, jonka avulla avoimen yhteyden sisältämä palveluyksikkö voidaan yksiselitteisesti löytää. Tällainen lähestymistapa kuitenkin lisää viestipalvelun kuormitusta, koska istuntojen ylläpitämisen lisäksi välimuistipalvelussa täytyy lähettää jokaisen tilauksen aiheuttavan pyynnön sivuvaikutuksena uusi tilauskäsky WebSocket-yhteyden sisältävälle palveluyksikölle, sillä ei voida tietää, onko tilausta vielä tehty vai ei. Tämän voisi toki ratkaista pitämällä välimuistipalvelussa kirjaa myös jokaisen istunnon tilauksista, mutta välimuistipalvelun kuorma ja järjestelmän monimutkaisuus kasvaisivat entisestään. Toisaalta, jos järjestelmää käytetään suunnittelufilosofian mukaisesti, ainoa syy tehdä resurssin haku olisi nimenomaan se, että asiakasohjelma haluaa tilata resurssin ja tarvitsee tilauksen alkamisen yhteydessä resurssin lähtötilan. Tällöin resurssin hakemisen yhteydessä tilaus tehtäisiin joka kerta muutenkin.

Jos kuormantasaus toteutetaan tahmaisesti IP-osoitteen perusteella, ongelma voidaan ratkaista ilman istuntotunnistetta, sillä HTTP-pyynnöt päätyisivät aina samalle palveluyksikölle kuin asiakasohjelman avaama WebSocket-yhteys. Kuitenkin, jos käyttäjällä on samasta IP-osoitteesta useita istuntoja auki, oikea WebSocket-yhteys ei löydy yksiselitteisesti ilman istuntotunnistetta. Toisaalta tällaisessa tilanteessa muutama ylimääräinen tilauskäsky ei olisi ongelma. Sen sijaan tahmainen kuormantasaus vaikeuttaa rajapintayksiköiden skaalausta, sillä lukumäärän muuttaminen muuttaa uusien yhteyksien kohdetta. Kun avoimia yhteyksiä ei ole järkevää katkaista skaalauksen yhteydessä, HTTP-pyynnöt päätyisivät skaalauksen muutoksesta eri rajapintayksikölle kuin jo avattu WebSocket-yhteys.

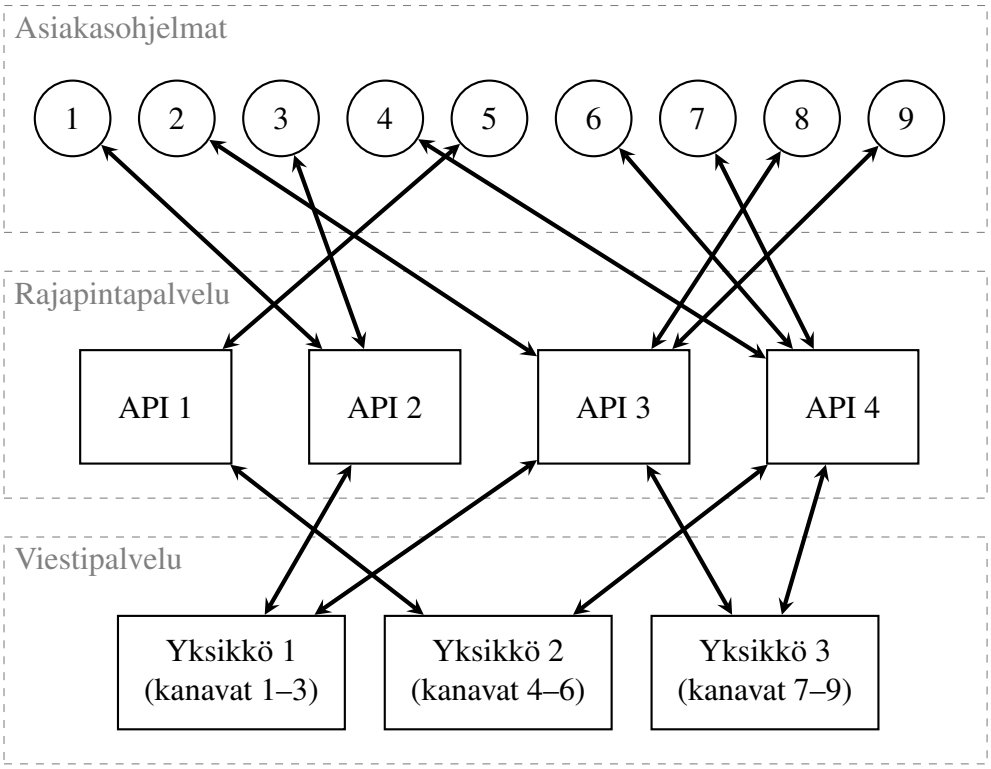
Järjestelmässä käytetään yksinkertaisinta ratkaisua, eli kanavatilausten ulkoistaminen asiakasohjelman vastuulle niin, että tilaus tapahtuu WebSocket-yhteyteen lähetetyllä tilauspyynnöllä. Resurssin tilan synkronointiongelma voidaan minimoida tilaamalla kanava ensin ja vasta tilauksen kuittauksen saamisen jälkeen hakemalla itse resurssi. Tämä hieman pidentää resurssin hakuaikaa, mutta vain niissä tapauksissa, joissa resurssin kanavan tilaus on tarpeen. Lisäksi tilauspyyntö on hyvin yksinkertainen ja nopea operaatio, eikä sen aiheuttama viive ole merkittävä. Mikäli nopeasti muuttuvien resurssien kanssa ongelmaksi muodostuu vanhentunut alkutila tai tilan muutos ennen kuin tilaus on ehditty tehdä, vaihtoehtona on ottaa käyttöön istuntotunnistetta käyttävä ratkaisu synkronoinnin varmistamiseksi.

5.2.2. Viestien välitys

Kun asiakasohjelma tilaa kanavan, rajapintayksikkö, jolla kyseisen asiakasohjelman WebSocket-yhteys on avoinna, tilaa kyseisen kanavan viestipalvelusta. Vastaavasti, kun kanavan tilaus lopetetaan tai kun WebSocket-yhteys sulkeutuu, rajapintayksikkö peruu kyseisen kanavan tilauksen viestipalvelusta. Näin ollen jokaisella rajapintapalveluyksiköllä on avoin yhteys vain niihin viestipalvelun yksiköihin, joissa tilatut kanavat sijaitsevat. Viestit välittyvät järjestelmässä avoimien yhteyksien avulla kaikille niistä kiinnostuneille rajapintayksiköille, ja vain niille. Lisäksi rajapintayksiköt eivät avaa viestipalveluyksikölle yhteyttä, ellei kyseisessä viestipalveluyksikössä ole jotain rajapintayksikköä kiinnostavaa kanavaa. Mikäli rajapintayksikkö tilaa useampaa kanavaa samalta viestipalveluyksiköltä samanaikaisesti, käytössä on silti vain yksi yhteys, jonka läpi viestit välitetään.

Avoimet yhteydet, sekä asiakasohjelman että rajapintayksikön puolelta avattuina, on tarkoitettu tilattujen kanavien saapuvien viestien vastaanottamiseen. Jos halutaan lähettää viesti, siihen ei tarvita avointa yhteyttä, vaan asiakasohjelma tekee tällöin HTTP-pyynnön REST-rajapintaan ja vastaavasti rajapintayksikkö tekee viestipyynnön viestipalveluun. HTTP-pyyntö päätyy satunnaiselle rajapintayksikölle, mutta viestipyyntö tulee lähettää sille viestipalveluyksikölle, jossa kohdekanava sijaitsee. Viestin lähettämiseksi rajapintayksikön ei kuitenkaan tarvitse itse olla tilannut kyseistä kanavaa.

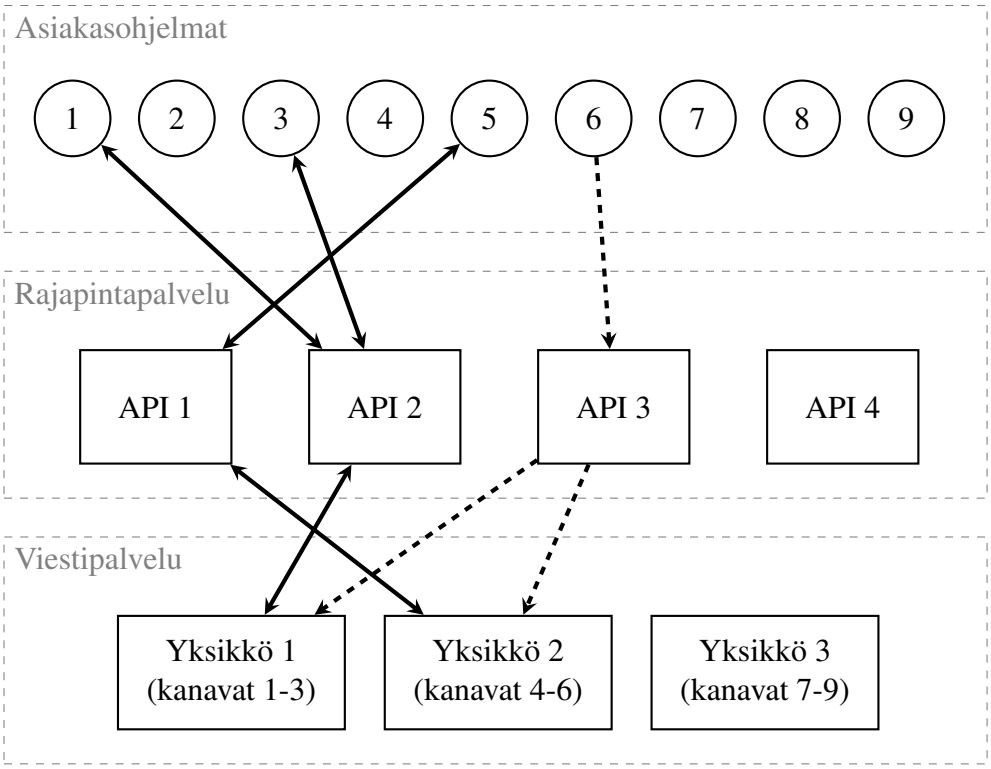
Kuvassa 5 on esimerkkitilanne, jossa järjestelmää käyttää yhdeksän asiakasohjelmaa, numeroituina 1–9. Esimerkkitilanteen rajapintapalvelu koostuu neljästä rajapintapalveluyksiköstä, ja viestipalvelu on jaettu kolmeen yksikköön siten, että jokainen viestipalveluyksikkö vastaa kolmesta viestikanavasta. Yksinkertaisuuden vuoksi kanavat on numeroitu asiakasohjelmien numeroiden mukaan, vaikka todellisessa käytössä kanavat yleensä edustavat yhtä tiettyä resurssia.



Kuva 5. Viestin välitys järjestelmässä esimerkkitilanteessa. Avoimia yhteyksiä kuvataan nuolilla yksiköiden välillä.

Kuvan tilanteessa jokainen asiakasohjelma on avannut yhden WebSocket-yhteyden ja on valmis vastaanottamaan viestejä. Jokainen asiakasohjelma on tilannut vain omaa numeroaan vastaavan kanavan, ja yhteydet rajapintayksiköiltä viestipalveluyksiköille on avattu näiden tilausten perusteella. Asiakasohjelma ei voi valita käyttämäänsä rajapintayksikköä, vaan valinta tapahtuu asiakasohjelmien ja rajapintayksiköiden välissä sijaitsevalla kuormantasaimella, jota ei ole piirretty kuvaan. Sen sijaan viestipalvelun edessä ei ole kuormantasainta, sillä rajapintapalvelun yksiköiden on itse valittava oikea viestipalvelun yksikkö käytetyn kanavan perusteella.

Kuvassa 6 on esitetty viestin välitys järjestelmässä HTTP-pyynnön seurauksena. Asiakasohjelma 6 tekee HTTP-pyynnön järjestelmään. Pyyntö käsitellään rajapintayksiköllä numero 3 (API 3), ja pyynnön seurauksena tapahtuu muutos, josta on ilmoitettava asiakasohjelmille 1, 3 ja 5. Rajapintayksikkö lähettää ilmoituksen kanaville 1, 3 ja 5 eli viestipalvelun yksiköille 1 ja 2. Viesti välittyy avoimia yhteyksiä pitkin kohteena oleville asiakasohjelmille.



Kuva 6. Viestin välitys järjestelmässä HTTP-pyynnön seurauksena. Tilanne on sama kuin kuvassa 5, mutta pyynnön kannalta epäolennaiset yhteydet on piilotettu. Avoimia yhteyksiä kuvataan nuolilla ja pyyntöjä katkoviivoin.

Mikäli kanavia käytettäisiin esimerkissä resurssipohjaisesti, lähettäisi rajapintayksikkö ilmoituksen muuttuneen resurssin kanavalle tietämättä sen vastaanottavista asiakasohjelmista. Resurssista kiinnostuneet asiakasohjelmat olisivat tilanneet resurssiin liittyvän kanavan ja saisivat resurssiin liittyvät ilmoitukset viestipalvelun kautta samalla tavalla kuin esimerkissä. Jos muutos koskisi useampaa eri resurssia, lähetettäisiin viesti vastaavasti jokaisen resurssin kanavalle.

Järjestelmässä viestipalvelu on skaalattu kolmeen yksikköön. Sääntönä yksikön määrittämiseen käytetään algoritmia, jossa kanavan nimestä lasketaan tiivistealgoritmilla luku, joka jaetaan yksiköiden lukumäärällä, eli kolmella, ja laskun jakojäännös määrittää kanavan yksikön. Koska yksiköiden lukumäärä on olennainen osa algoritmia, yksiköiden lukumäärän muuttaminen vaatii muutoksen kaikkiin sitä käyttävien palvelujen toimintaan. Myös välimuistipalvelun kanssa käytetään samaa mekanismia yksikön valintaan.

5.3. Testaus

Tässä työssä esitetyn järjestelmän suorituskykyä mitataan kuormittamalla järjestelmää mitaten samalla järjestelmään tehtyjen pyyntöjen laatua ja määrää, sekä viivettä, joka kuluu pyyntöjen käsittelemiseen. Locust on avoimen lähdekoodin Python-pohjainen kuormantestausohjelmisto, jota järjestelmässä käytetään järjestelmän kuormitukseen ja sen suorituskyvyn mittaamiseen. Sen avulla voidaan määritellä testikäyttäjiä

ja niiden käyttäytymistä, ja näin simuloida käyttäjien toimintaa järjestelmässä. Testikäyttäjiä voidaan suorittaa rinnakkain haluttu määrä, ja näin simuloida järjestelmän kuormitusta. [56]

Kuormatestauksen testikäyttäjät kutsuvat järjestelmän suunnittelufilosofian puitteissa rakennettua testisovellusta. Testisovellus sisältää tietokantarakenteita, niiden päälle rakennettuja ORM-malleja sekä malleja kuvaavan REST-rajapinnan. REST-rajapinnan kautta tapahtuvista resurssien muutoksista lähetetään päivitysviestejä viestipalvelun kautta kaikille niille käyttäjille, jotka ovat tilanneet kyseisen resurssin kanavan. Näin kutsut testirajapintaan simuloivat tehokkuuden kannalta kriittiset komponentit: rajapinnan, tietokannan ja viestipalvelun.

Testauksen tarkoituksena on määrittää suurin mahdollinen lähetettyjen viestien määrä tietyllä realistiseksi katsotulla viestien tyyppijakaumalla. Yksi testikerta sisältää jatkuvaa testikäyttäjämäärän kasvattamista, kunnes järjestelmä ei enää kykene käsittelemään pyyntöjä. Ongelmat käsittelyssä ilmenevät joko kasvavana virhevastauksien osuuksina tai kasvavana viiveenä pyyntöihin vastaamisessa. Testi suoritetaan useita kertoja ja pyritään löytämään sellainen skaalattujen palveluiden yksiköiden lukumäärä, jolla saavutetaan maksimaalinen suorituskyky. Kaikkien palveluiden yksiköiden lukumäärä pidetään kolmella jaollisena, jotta jokaisella palvelintietokoneella on suorituksessa toistensa kanssa saman verran yksiköitä.

5.3.1. Testisovellus

Testisovellus koostuu pohjimmiltaan kahdesta osasta: testitietokannasta ja testirajapinnasta. Testisovelluksen tietokantaa mallinnetaan kolmella ORM-mallilla, jotka esittävät käyttäjä-, keskusteluryhmä- ja viestiresursseja. Keskusteluryhmät koostuvat jostakin joukosta käyttäjiä, johon ryhmän jäsenet voivat halutessaan lähettää viestejä tai lisätä uusia jäseniä. Käyttäjät voivat myös luoda uusia keskusteluryhmiä.

Testirajapinta koostuu resurssien hakuun ja manipulointiin tarkoitetusta REST-rajapinnasta sekä WebSocket-rajapinnasta, josta käyttäjät voivat tilata ja vastaanottaa resurssien päivitysviestejä. REST-rajapinta sisältää listaus-, haku-, lisäys-, muokkaus- ja poistokutsut kaikille kolmelle resurssille. Lisäksi on erilliset kutsut keskusteluryhmän viestien ja jäsenten listaamiseen, yksittäisen käyttäjän keskusteluryhmien listaamiseen sekä käyttäjän lisäämiseen ja poistamiseen ryhmästä.

5.3.2. Testikäyttäjä

Locust-testikäyttäjän määrittäminen tapahtuu luomalla erilaisia toimintoja, joille asetetaan painotus sekä toimintoon liitetty logiikka. Testikäyttäjä suorittaa yhden toiminnon oman ajastuksensa perusteella, esimerkiksi joka sekunti. Tässä työssä testikäyttäjien ajastus on määritetty olemaan satunnainen aika 0,1 ja 1,9 sekunnin välillä, tehden siis keskimäärin yhden toiminnon sekunnissa.

Testikäyttäjä alustetaan sen luontivaiheessa luomalla sen käyttöön käyttäjä, ja avaamalla uusi WebSocket-yhteys käyttäen luotua käyttäjää. Myös kaikki käyttäjän tekemät REST-rajapintakutsut tehdään käyttäen samaa luotua käyttäjää. Alustuksen jälkeen testikäyttäjä käynnistyy, ja suorittaa toimintojaan ajastuksensa mukaisesti,

kunnes testi päättyy tai lopetetaan. Testikäyttäjän suorittama toiminto valitaan painotuksista lasketun todennäköisyyden perusteella satunnaisesti.

Testikäyttäjällä on omassa muistissaan istuntoon sidottuja tietoja. Tällaisia tietoja on käytössä oleva käyttäjä, lista tunnetuista käyttäjistä ja lista tunnetuista keskusteluryhmistä. Mikäli käyttäjä hakee käyttäjiä tai keskusteluryhmiä, tallennetaan hakuun saadut tiedot muistiin. Kun käyttäjä tekee johonkin tiettyyn keskusteluryhmään tai käyttäjään kohdistuvia toimintoja, kohde valitaan satunnaisesti tunnettujen resurssien listasta.

Taulukko 1 esittää testikäyttäjän eri toiminnot sekä niiden painotukset. Siinä ei kuitenkaan ole kaikki toiminnot, joita testikäyttäjä tekee, sillä käyttäjä reagoi vastaanottamiinsa WebSocket-rajapinnasta lähetettyihin viesteihin. Tällaisia viestejä ovat viestin vastaanottaminen johonkin omista keskusteluryhmistä, sekä se, että on tullut lisätyksi johonkin uuteen keskusteluryhmään. Tieto uudesta jäsenyydestä lähetetään sekä lisätylle käyttäjälle, että myös kyseiseen keskusteluryhmään, josta se päätyy kaikille sen jäsenille.

Painotus	Toiminnon kuvaus
1 (5 %)	Hakee yhden sivun keskusteluryhmien listauksesta, eli 100 kappaletta, ja tallentaa tunnettujen keskusteluryhmien listalle. Hakee jokaisella kerralla seuraavan hakemattoman sivun, kunnes kaikki sivut on käyty läpi. Jos kaikki saatavilla olevat sivut on jo haettu, hakee satunnaisen sivun.
1 (5 %)	Hakee satunnaisen yksittäisen keskusteluryhmän tunnetuista keskusteluryhmistä.
1 (5 %)	Luo uuden keskusteluryhmän ja lisää sen tunnettujen keskusteluryhmien listaan. Keskusteluryhmän luonut käyttäjä on automaattisesti sen jäsen.
1 (5 %)	Sama hakulogiikka kuin keskusteluryhmien haulle, mutta hakee käyttäjiä.
1 (5 %)	Sama logiikka kuin yksittäisen keskusteluryhmän haulle, mutta hakee käyttäjän.
1 (5 %)	Lisää satunnaisen käyttäjän tunnetuista käyttäjistä johonkin omista keskusteluryhmistä.
14 (70 %)	Lähettää viestin johonkin omista keskusteluryhmistä.

Taulukko 1. Testikäyttäjän toimintojen painotukset ja kuvaukset. Painotukset annettu myös prosentteina kuvastaen toiminnon tapahtumisen todennäköisyyttä.

Keskusteluryhmään lähetetyn viestin vastaanottaminen ei aiheuta lisätoimintojen suorittamista, mutta lisätyksi tuleminen aiheuttaa kyseisen keskusteluryhmän lisäämisen tunnettujen keskusteluryhmien listaan. Vastaavasti tieto uudesta jäsenestä jollain omalla keskusteluryhmällä lisää kyseisen käyttäjän tunnettujen käyttäjien listaan. Vastaanotetut viestit myös tilastoidaan testauksessa, mutta näille viesteille ei ole mahdollista määrittää pyynnön kaltaista viivettä, sillä ne ovat yksisuuntaisia.

Sen sijaan viesteihin sidotaan aikaleima, josta voidaan laskea kuinka kauan viestillä kesti päätyä vastaanottajalle. Aikaleiman ajanhetki on sen pyynnön käsittelyn

alkamisaika, jonka seurauksena viesti on luotu. Viestien tilastoitu viive on siis tämän aikaleiman ja viestin vastaanottamisen välillä kulunut aika.

Koska asiakasohjelma on itse vastuussa kanavien tilauksesta, kulkee WebSocket-rajapinnasta myös testikäyttäjän lähettämiä viestejä. Ainoa viestityyppi on kanavan tilauspyyntö, ja vaikka tilausviestiin vastataan rajapinnasta kuittauksella, on viiveen laskeminen lähetetyille viesteille vaikeaa. Koska tilauksia tehdään verrattain harvoin, viiveen merkitseminen näille viesteille nollaksi ei juuri muuta viiveen mittaustuloksia.

5.3.3. Testiympäristö

Locust sisältää ominaisuuden hajautetulle suorittamiselle. Tällaisessa konfiguraatiossa Locust sisältää pääohjelman, sekä pääohjelmaan yhteyden ottavia työntekijäohjelmia. Pääohjelma jakaa työntekijöille käyttäjiä, ja työntekijät raportoivat tehdyistä pyynnöistä ja niihin liittyvistä tiedoista. Pääohjelma kerää ja yhdistää tiedot, sekä esittää ja raportoi testin tulokset. Locust-pääohjelmaa suoritetaan järjestelmän ulkopuolisella tietokoneella.

Järjestelmä noudattaa pitkälti arkkitehtuurikuvauksessa esitettyä arkkitehtuuria palvelimineen. Locust-työntekijäohjelma kuitenkin on lisätty järjestelmään tilattomana palveluna. Rajapintapalvelun yksiköiden lukumäärän lisäksi myös Locust-palvelun yksiköiden lukumäärää vaihdellaan testien suorituksen aikana parhaimman lukumäärän löytämiseksi.

Locust-palvelua suoritetaan samassa ympäristössä kuin muitakin järjestelmän komponentteja. Testikäyttäjien suorittaminen on raskasta niiden suuren määrän vuoksi, jolloin ne kuluttavat osan palvelinklusterin käytössä olevista laskentaresursseista. Kuormituksen rajoittamiseksi testikäyttäjä tekee yhden toiminnon sekunnissa, mikä on merkittävästi enemmän kuin mitä todellinen käyttäjä tekisi, ja näin voidaan vähentää tarvittavien testikäyttäjien määrää.

Ylimääräinen kuormitus voi vaikuttaa testien lopputuloksiin tuloksia laskevasti. Toisaalta itse testijärjestelmä on loogisesti melko vaatimaton, jolloin testeissä tehdyt pyyntömäärät eivät ole vertailukelpoisia todellisen järjestelmän pyyntökapasiteettiin verrattuna. Lisäksi tällaisen käyttäjämäärän suorittaminen järjestelmän ulkopuoliselta tietokoneelta on vaikeaa, sillä käytössä olevat palvelimet ovat varsin tehokkaita.

5.3.4. Testivalmistelut

Ennen testien suorituksen aloittamista järjestelmää kokeiltiin kuormittamalla sitä lyhyesti erilaisilla määrillä käyttäjiä ja erilaisilla määrillä rajapintayksiköitä. Näin arvioitiin rajapintayksiköiden optimaalisen lukumäärän olevan yli 30. Kokeilu myös paljasti järjestelmästä virheitä ja muita ongelmia, jotka korjattiin ennen varsinaisten testien suorittamista.

Ongelmaksi muodostui tietokantaan avatut samanaikaiset avoimet yhteydet, jotka voimakkaasti rajoittavat rajapintapalvelun yksiköiden suurinta lukumäärää. Ongelma korjattiin ottamalla käyttöön tietokannan yhteyksien hallintaan tarkoitettu kirjasto, joka kierrättää samoja, aiemmin avattuja tietokantayhteyksiä eri toimintoihin rajapinnan sisällä. Tämä lisäsi suorituskykyä jonkin verran, mutta tietokanta oli

vielä tämänkin jälkeen järjestelmän pullonkaula, sillä rajapintakutsut tekivät liikaa tietokantakyselyitä.

Koska suurin osa testirajapintaan tehdyistä pyynnöistä koski uuden viestin luontia, täytyi tietokannan suorituskyvyn parantamiseksi viestin luontia optimoida. Viesti luotiin tilapäisesti välimuistiin, ja hetken kuluttua viesti siirrettiin pysyvään tietokantaan yhdessä muiden tällä aikaa saapuneiden viestien kanssa. Luomalla tietokantaan monta viestiä yhdellä pyynnöllä, voitiin merkittävästi parantaa tietokannan suorituskykyä.

Kokeilun aikana huomattiin myös, että mikäli Locust-yksiköitä on liian vähän, tulee yhden yksikön vastuulle niin monta testikäyttäjää, että yksittäisen yksikön suorittimen käyttörajat tulevat vastaan. Locust-yksiköiden määrän kasvattaminen tarpeellisesta määrästä ylöspäin ei kuitenkaan parantanut järjestelmän suorituskykyä. Kaikissa testeissä käytetty Locust-yksiköiden lukumäärä on 9, joka riittää testeissä tarvitun määrän suorittamiseen.

5.3.5. Testit

Testit suoritettiin järjestelmässä, jossa palvelut olivat käyttövalmiita ja tilaltaan käyttämättömiä. Tietokanta ja välimuistipalvelu nollattiin jokaisen testin jälkeen. Ennen testin aloittamista rajapintapalvelu skaalattiin vastaamaan haluttua yksikkömäärää ja Locust-pääohjelma käynnistettiin.

Testin alussa käyttäjiä ei vielä ollut, ja testikäyttäjiä luotiin lisää testin edetessä yksi kappale sekunnissa. Kun käyttäjiä oli 1000, käyttäjien luonti pysäytettiin ja järjestelmän tilaa seurattiin muutaman minuutin ajan. Tämän jälkeen käyttäjämäärää nostettiin 25 käyttäjän ryhmissä niin, että yksi uusi käyttäjä luotiin kolmen sekunnin välein. Ryhmien lisäämisien välissä järjestelmän tilaa seurattiin myös muutaman minuutin ajan, ja ryhmiä lisättiin niin kauan, kunnes testin lopetusehdot täyttyivät. Yksittäisen testin kesto oli noin 20 minuuttia tuhannelle käyttäjälle, ja nousi sen jälkeen muutamia minuutteja per 25 käyttäjää.

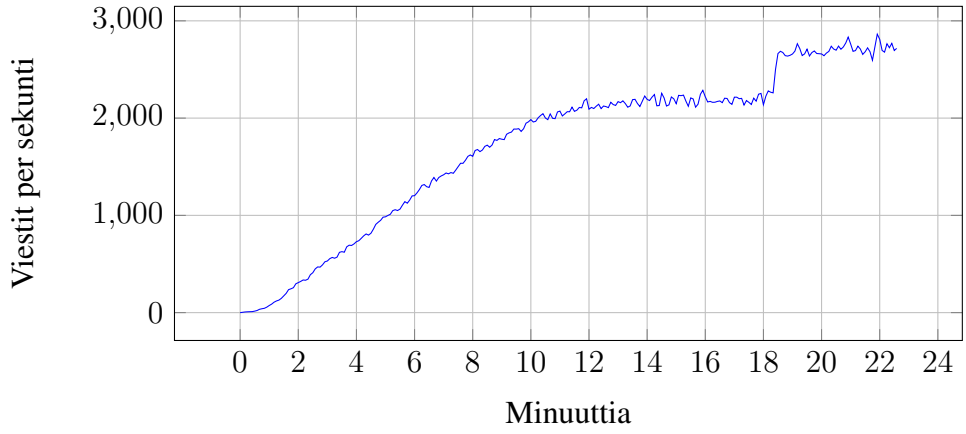
Testi lopetettiin, kun joko pyyntöjen 95. persentiilin pyyntöaika ylitti kaksi sekuntia tai kun virheiden osuus pyynnöistä ylitti yhden prosentin. 95. persentiilin pyyntöaika tarkoittaa sitä, että 95 prosenttia pyynnöistä on suoritettu nopeammin kuin kyseinen aika. Viiveet ja virheiden osuus pyynnöistä laskettiin 5 sekunnin välein samankokoisille ikkunoille.

Testejä suoritettiin aloittaen 30 rajapintayksiköstä, ja nostaen yksiköiden lukumäärää 3 yksiköllä kerrallaan. Testien suorittaminen voitiin lopettaa, kun voitiin selvästi havaita, ettei järjestelmän suorituskykyä voitu enää parantaa lisäämällä yksiköiden lukumäärää. Testin loputtua testin tulokseksi kirjataan paras saavutettu viestien käsittelymäärä sekunnissa sekä testikäyttäjien lukumäärä kyseisellä hetkellä.

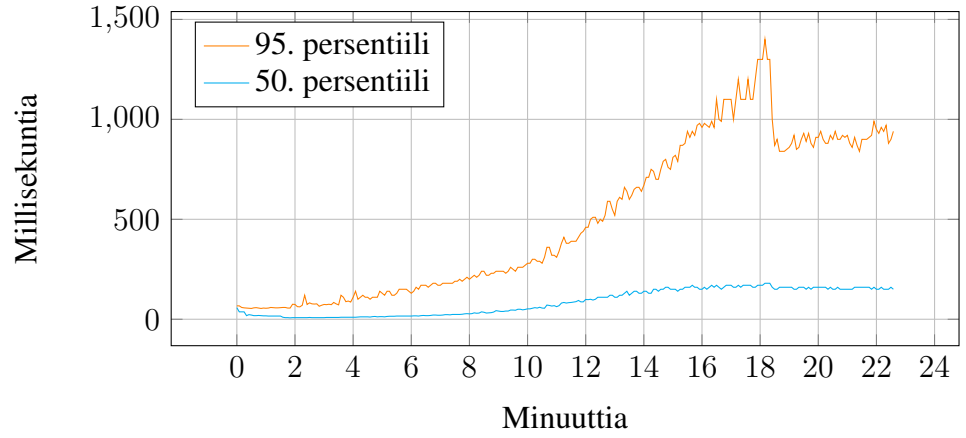
Ennen testien aloittamista suoritettiin esimerkkitesti, jolla testattiin järjestelmän toimintaa 30 rajapintayksiköllä ja 1100 testikäyttäjällä. Esimerkkitestin tarkoituksena oli varmistaa, että testin suoritus onnistuu ja testitulokset saadaan kerättyä halutussa muodossa. Myös järjestelmän palauttaminen alkutilaan testien jälkeen testattiin toimivaksi.

Kuva 7 näyttää viestien käsittelymäärän sekunnissa testin edetessä. Kuva 8 näyttää viestien viiveet eri persentiileillä. Mitatut persentiilit ovat 50. ja 95. persentiili,

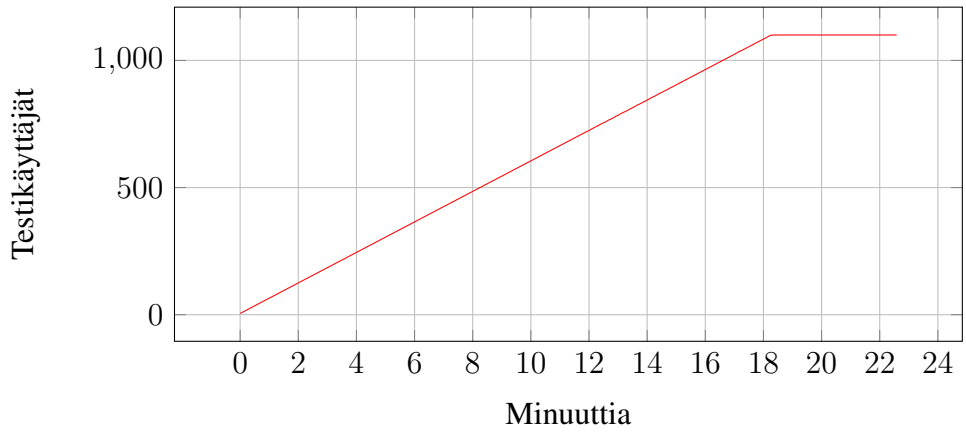
jotka on esitetty vaaleansinisellä ja oranssilla viivalla. Kuva 9 näyttää testikäyttäjien lukumäärän testin edetessä. Testikäyttäjien määrää kasvatettiin tasaisesti yksi yksikkö sekunnissa aina 1100 käyttäjään asti.



Kuva 7. Esimerkkitestin viestien käsittelymäärä per sekunti suhteessa aikaan.



Kuva 8. Esimerkkitestin viestien viiveet eri persentiileillä suhteessa aikaan.



Kuva 9. Esimerkkitestin testikäyttäjien lukumäärä testin edetessä.

Esimerkkitestissä käytettyjen Locust-yksikköjen määrä oli vain kolme, joka oli liian vähän suorittamaan yli tuhatta testikäyttäjää samanaikaisesti. Ongelma johtui Locust-yksikön suoritinresurssien loppumisesta, mikä ilmeni viiveiden nousuna ja viestien käsittelynopeuden nousun pysähtymisenä uusia käyttäjiä luodessa. Minuutin 18 jälkeen, kun uusien käyttäjien luonti loppui, viiveet laskivat äkisti ja viestien käsittelynopeus nousi noin 2800 viestiin sekunnissa. Ennen varsinaisten testien aloittamista Locust-yksiköiden määrä nostettiin 9 yksikköön, ja sen arvioitiin riittävän ainakin 2000 käyttäjän samanaikaiseen suorittamiseen. Esimerkkitestin seurauksena myös käyttäjien luontitapaa paranneltiin, jotta minimoidaan käyttäjien luomisen vaikutus viiveisiin suurilla käyttäjämäärillä.

Taulukossa 2 on esimerkkitestin aikana kaikki testikäyttäjien käsittelemien viestien lukumäärät viestityypeittäin. *POST* ja *GET* ovat HTTP-pyyntöjen tyyppejä, ja *WSR* tarkoittaa vastaanotettua WebSocket-viestiä ja vastaavasti *WSS* on lähetetty WebSocket-viesti. Viestien tyypit ja kuvaukset on esitetty testikäyttäjän näkökulmasta.

Viestin tyyppi	Kuvaus	Lukumäärä (osuus)
WSR add-chat-for-user	Ilmoitus keskusteluryhmään lisätystä uudesta käyttäjästä.	54 676 (2,3 %)
POST add-user-to-chat	Pyyntö lisätä käyttäjä keskusteluryhmään.	54 695 (2,3 %)
WSR add-user-to-chat	Ilmoitus, että on itse tullut lisätyksi keskusteluryhmään.	110 056 (4,6 %)
POST create-chat	Pyyntö luoda uusi keskusteluryhmä.	27 745 (1,2 %)
POST create-user	Pyyntö luoda uusi käyttäjä.	1 100 (0,046 %)
GET get-chat	Pyyntö hakea keskusteluryhmän tiedot.	27 140 (1,1 %)
GET get-user	Pyyntö hakea käyttäjän tiedot.	27 255 (1,1 %)
GET get-user-chats	Pyyntö hakea käyttäjän keskusteluryhmät.	27 586 (1,2 %)
GET get-users	Pyyntö hakea käyttäjälistauksen sivu.	27 770 (1,2 %)
WSR message-create	Ilmoitus uudesta viestistä keskusteluryhmässä.	1 537 934 (64 %)
POST send-message	Pyyntö lähettää viesti keskusteluryhmään.	380 907 (16 %)
WSR subscribe	Pyyntö tilata kanava.	54 417 (2,3 %)
WSS subscribe	Kuittaus kanavan tilauksen onnistumisesta.	55 517 (2,3 %)
Yhteensä		**2 386 798**

Taulukko 2. Esimerkkitestin viestien tyyppien ja niiden lukumäärien jakautuminen.

Esimerkkitestin suorituksen aikana tuli yhteensä 24 virhettä, joista 19 oli pyyntöjä lisätä käyttäjä keskusteluryhmään, ja loput 5 oli pyyntöjä lähettää uusi viesti.

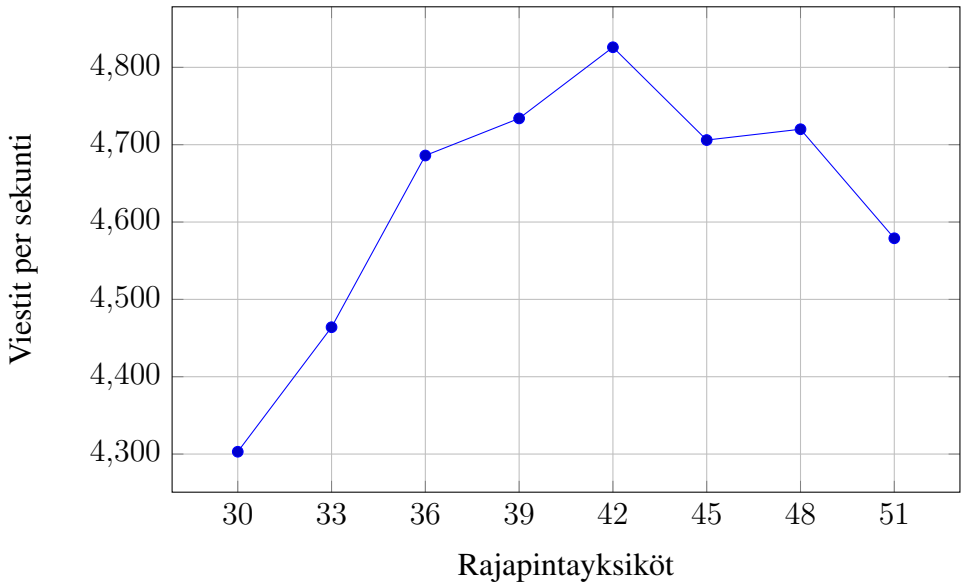
Jokainen virhe johtui siitä, että tietokanta kieltäytyi avaamasta uutta yhteyttä ja rajapinta palautti virheen käyttäjälle. Virheet selittävät eron keskusteluryhmään lisäämispyynnön ja lisätyksi tulemisen ilmoituksen lukumäärien välillä. Kanavan tilauspyyntöjen ja tilauksen kuittauksien välinen ero selittyy sillä, että jokainen käyttäjä saa automaattisesti tilauksen käyttäjäresurssiaan koskevalle kanavalle avatessaan WebSocket-yhteyden.

6. TULOSTEN ARVIOINTI

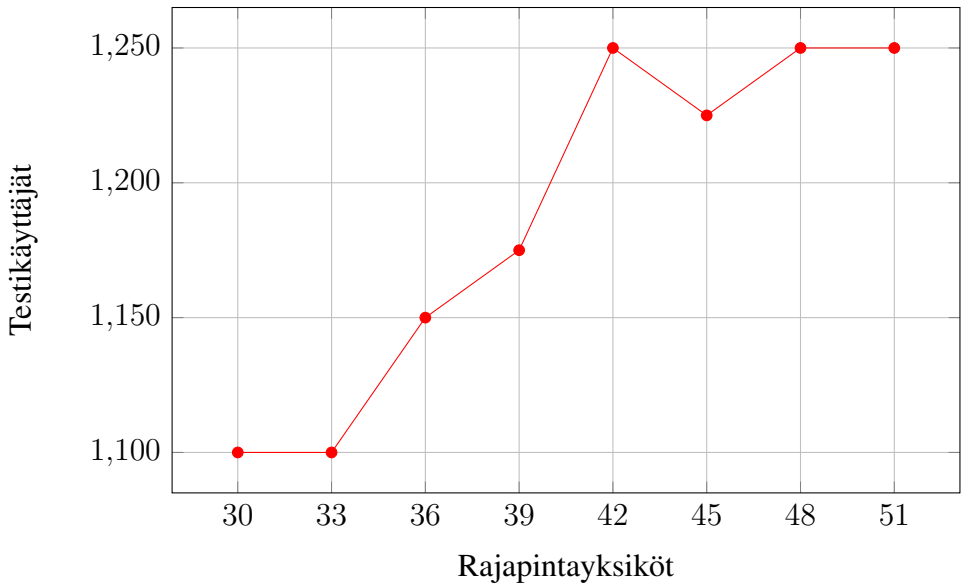
Testejä suoritettiin yhteensä 7 kappaletta. Rajapintayksiköiden lukumäärää nostettiin aina 51 yksikköön asti, kunnes oli ilmeistä, että järjestelmän suorituskyky ei noussut yksiköitä lisäämällä. Testikäyttäjien maksimimäärä nousi rajapintayksiköiden lukumäärän noustessa tuhannesta aina 1250 kappaleeseen asti.

Kuva 10 kuvaa rajapintayksiköiden vaikutusta suurimpaan saavutettuun viestien käsittelymäärään. Kuvasta on selkeästi nähtävissä suorituskyvyn huippukohta, kun rajapintayksiköitä on 42 kappaletta, jossa testikäyttäjät kykenivät käsittelemään runsaat 4800 viestiä sekunnissa. Rajapintayksiköiden kasvattaminen tästä eteenpäin aiheutti suorituskyvyn laskemista.

Kuva 11 kuvaa testikäyttäjien lukumäärää, sillä hetkellä, kun suurin viestien käsittelymäärä saavutettiin. Kuvaan piirretty testikäyttäjämäärä oli myös kyseisen testin suurin käytetty käyttäjämäärä, ja paras suorituskyky saavutettiin aina juuri ennen testin loppumisehdon toteutumista. Järjestelmän suorituskykymaksimi ei ilmennyt pelkästään testikäyttäjien lukumäärää tarkastelemalla, sillä testikäyttäjien lisääminen muuttui helpommaksi rajapintayksiköiden lisääntyessä.



Kuva 10. Rajapintayksiköiden lukumäärän vaikutus suurimpaan mahdolliseen viestien käsittelymäärään testikäyttäjillä.



Kuva 11. Testikäyttäjien suurin lukumäärä eri rajapintayksiköiden lukumäärillä.

Kaikkien suoritettujen testien käsitellyt viestimäärät noudattivat varsin tarkasti samaa jakaumaa taulukossa 2 esitetyn esimerkkitestin viestijakauman kanssa. Tämä oli odotettavaa, sillä käsitellyt viestit olivat suoraa seurausta testikäyttäjän tekemien pyyntöjen jakaumasta, joka oli jokaisessa testissä täsmälleen sama. Virheiden määrissä oli huomattavia eroja testien välillä, mutta virheet kuitenkin koskivat samoja rajapintakutsuja kuin esimerkkitestissä.

Viestimäärät testien edetessä kasvoivat samaan tapaan kuin esimerkkitestissä esitettynä kuvassa 7, mutta johtuen varovaisemmasta käyttäjien lisäämisestä 1000 käyttäjän jälkeen, ei viestien käsittelynopeus heitellyt niin paljon kuin esimerkkitestissä. Ensimmäisen 15 minuutin aikana käsittelynopeus kasvoi varsin tasaisesti, mutta alkoi hidastua käyttäjämäärän maksimia lähestyttäessä. Testeissä viestimäärän maksimi saavutettiin runsaan 30 minuutin kohdalla.

6.1. Johtopäätökset

Rajapintapalvelun yksiköiden määrän noustessa viiveet pysyivät paremmin hallinnassa, mutta järjestelmän suorituskyvyn maksimia lähestyttäessä pyynnöt alkoivat päättyä virheisiin useammin. Tämä johtui siitä, että kun rajapintayksiköitä oli tarpeeksi monta, tietokannan yhteyspaikat loppuivat kesken riippumatta siitä, että niitä hallittiin ja uudelleenkäytettiin rajapintayksikön sisällä. Ongelma olisi ratkaistavissa hallitsemalla tietokannan yhteyksiä järjestelmän tasolla suorittamalla kaikille rajapintayksiköille yhteistä tietokantayhteyksien hallintasovellusta. Näin rajapintayksiköt voisivat jakaa tietokantayhteyksiä keskenään, ja rajapintayksiköiden suurin mahdollinen lukumäärä olisi merkittävästi nykyistä suurempi.

Viiveiden nousua havaittiin kaikissa testeissä, mutta tarpeeksi suurella määrällä rajapintayksiköitä virheiden osuus ylitti rajan viiveitä nopeammin aiheuttaen testin loppumisen. Viiveet kasvoivat etenkin keskusteluryhmän uuden viestin ilmoitukseen käytetyissä viesteissä, eli palvelimelta asiakasohjelman suuntaan lähetettyjen

WebSocket-viestin osalta. Tämä selittyy sillä, että viestipalvelu ei pystynyt välittämään viestejä tarpeeksi nopeasti aiheuttaen jonoutumista ja viivettä viestin välitykseen.

Järjestelmän suorituskyvyn parantamiseksi luultavasti suurin merkitys olisi tietokannan jakamisesta useammalle tietokantayksikölle. Jakaminen pitäisi tehdä niin, että käyttäjiä ei olisi jaettu, vaan ne olisivat replikoitu kaikille yksiköille, sillä käyttäjiä manipuloidaan verrattain harvoin. Sen sijaan keskusteluryhmät olisi jaettu tasaisesti jokaiselle yksikölle niin, että keskusteluryhmän jäsenyystiedot sekä viestit olisivat myös kyseisellä yksiköllä. Keskusteluryhmän tiedot sijaitsisivat siis vain yhdellä tietokantayksiköllä, jonka valinta olisi rajapintapalvelun vastuulla. Valintaan voisi käyttää esimerkiksi keskusteluryhmän tunnistenumeroa tai siitä johdetun tiivisteen jakojäännöstä viestipalvelun valinnan tapaan.

Tietokannan jakaminen voisi ratkaista tietokantapyyntöjen suuresta määrästä johtuvat suorituskykyongelmat ja auttaa tietokannan yhteyspaikkojen loppumiseen, sillä jokaisella tietokantayksiköllä on oma yhteysmääränsä käytettävänä. Koska keskusteluryhmiin lähetetään viestejä hyvin paljon, on todennäköistä, että tietokannan skaalaaminen ei yksin riittäisi ratkaisemaan viestien lähetyksen suorituskykyongelmaa, vaan jonkinlainen välimuistipalvelun käyttö välivarastointiin olisi edelleen tarpeen. Suorituskykyä voisi parantaa myös muiden rajapintakutsujen optimoinnilla vähentäen tarvittavia tietokantapyyntöjä ja lisäämällä välimuistin käyttöä.

Tietokannan jakamista yksinkertaisempi, joskin ei yhtä tehokas tapa, olisi käyttää lukureplikaa. Lukureplika tarkoittaa tietokannan pääyksiköstä tehtyä reaaliaikaista kopiota, jota voi käyttää ainoastaan tietojen lukemiseen. Tällaisen replikan käyttöönotto on hyvin helppoa eikä vaadi juurikaan muutoksia rajapintapalveluun. Lukureplikan avulla tietokannan lukuoperaatiot voitaisiin kokonaan siirtää pois tietokannan pääyksiköltä vapauttaen sen resursseja kirjoitusoperaatioihin. Koska järjestelmä on käyttäytymiseltään kirjotusintensiivinen, voitaisiin tällä saavuttaa vain pieni suorituskykyparannus. Kun pääyksikön kapasiteetti olisi taas käytetty, ei lukureplikoinnilla enää voisi saavuttaa suorituskykyhyötyä.

Viestipalvelun skaalaaminen on helppoa, ja koska palvelu on sama kuin välimuistipalvelu, myös se skaalautuisi samalla. Viesti- ja välimuistipalvelun skaalaamisella voitaisiin saavuttaa suurempi määrä WebSocket-viestien välityksiä sekunnissa. Koska viestipalvelu oli jo nyt lähellä rajaansa, muodostuisi se todennäköisesti seuraavaksi pullonkaulaksi, jos tietokantaa skaalattaisiin.

6.2. Teknologiavalinnat

Python ja Django lisäävät suoritukseen useita kerroksia, ja näillä teknologioilla toteutettu ohjelma suoriutuu pyynnöistä merkittävästi hitaammin kuin esimerkiksi C-ohjelmointikielellä toteutettu vastaava ohjelma. Pullonkaulaksi järjestelmässä ei kuitenkaan muodostunut suoritin tai muisti, vaan rajapintapalvelun ulkopuolisiin palveluihin kohdistuvat toiminnot, kuten tietokantaoperaatiot ja viestipalvelun kapasiteetti välittää viestejä. Siispä ei voida odottaa järjestelmän suorituskyvyn olevan merkittävästi parempi, vaikka rajapintapalvelu toteutettaisiin esimerkiksi C-ohjelmointikielellä. Toisaalta tällaisessa tilanteessa kuitenkin voitaisiin odottaa pienemmän määrän rajapintayksiköitä riittävän saavuttamaan sama suorituskyky.

Toiminnallisuudeltaan samantasoisen sovelluksen kehittäminen vaikkapa C-ohjelmointikielellä on merkittävästi hitaampaa verrattuna Pythonilla ja Djangolla toteutettuun sovellukseen Djangon tarjoamien valmiiden ratkaisujen vuoksi. Siitä huolimatta, että suoraan C:llä, tai vastaavalla ohjelmointikielellä kirjoitettuja ohjelmistokehyksiä on olemassa, on niiden avulla toteutettavan sovelluksen kehittäminen yleensä paljon hitaampaa. Djangon ympärille rakennettu ekosysteemi tarjoaa ratkaisuja moniin yleisiin ongelmiin, eikä kehittäjän tarvitse käyttää aikaa näiden ratkaisujen toteuttamiseen. Tässä työssä Djangon käyttö lyhensi järjestelmän kehitysaikaa merkittävästi, sillä käytössä on REST- ja WebSocket-rajapintojen kehitystyökalujen lisäksi valmiit laajennokset muun muassa tietokanta-, välimuisti- ja viestipalveluiden käyttöön.

6.3. Vaatimusten toteutuminen

Järjestelmän vaatimusten toteutumista arvioidaan testijärjestelmää vasten tehdyn kuormatestauksen avulla. Kuormatestauksessa mitattujen pyyntöjen viiveiden perusteella järjestelmän voidaan katsoa täyttävän reaaliaikaisuusvaatimus. Myös nopeasti muuttuvat resurssit, kuten testijärjestelmän keskusteluryhmän viestit, pystyttiin välittämään järjestelmän käyttäjille reaaliaikaisesti. Koska yhdessä keskusteluryhmässä oli useita käyttäjiä ja viestit välittyivät keskusteluryhmän kaikille jäsenille reaaliaikaisesti, voidaan järjestelmän katsoa täyttävän myös interaktiivisuusvaatimuksen.

Tietokantaa lukuun ottamatta kaikki testissä käytetyt komponentit skaalattiin niin, että skaalaamisesta saatiin erityistä suorituskykyhyötyä. Tietokannan skaalautumiseen esitettiin mahdollisia keinoja, joiden avulla järjestelmän suorituskyky todennäköisesti olisi parantunut. Järjestelmän voidaan siis katsoa toteuttavan skaalautuvuusvaatimuksen.

Parhaimmillaan järjestelmä pystyi käsittelemään lähes 5000 viestiä sekunnissa yli tuhannelta simuloidulta käyttäjältä samanaikaisesti. Koska yksi simuloitu käyttäjä vastaa useampaa oikeaa käyttäjää, voidaan järjestelmän katsoa suurella todennäköisyydellä voivan palvella tuhansia käyttäjiä samanaikaisesti. Järjestelmän voidaan siis katsoa täyttävän suorituskykyvaatimuksen.

6.4. Tutkimuskysymyksiin vastaaminen

Työn ensimmäisenä tutkimuskysymyksenä oli, miten hyvin järjestelmän suunnittelufilosofia toimii käyttötarkoitukseensa. Suunnittelufilosofian seurauksena jonkin verran logiikkaa jouduttiin siirtämään asiakasohjelman vastuulle suorituskyvyn maksimoimiseksi. Tämä monimutkaistaa asiakasohjelman toteutusta ja vaikeuttaa kolmannen osapuolten integrointia järjestelmään. Järjestelmän suunnittelufilosofia toimii siis kokonaisuutena melko hyvin käyttötarkoitukseensa, vaikka lisääkin joidenkin järjestelmän osien monimutkaisuutta.

Toisena tutkimuskysymyksenä tutkittiin, kuinka suorituskykyinen vaatimukset täyttävä järjestelmä saadaan toteutettua. Järjestelmän suunnitteluvaiheessa reaaliaikaisuuden ja interaktiivisuuden ongelmiin löydettiin suorituskykyiset

ratkaisut niin, että suorituskykyä on mahdollista edelleen parantaa horisontaalisella skaalaamisella. Testijärjestelmän avulla pystyttiin vahvistamaan, että arkkitehtuuri mahdollisti vaatimukset täyttävän sovelluksen kehittämisen.

Kolmantena tutkimuskysymyksenä oli arvioida, millaisiin muihin sovelluksiin tällaista järjestelmää voidaan käyttää ja millaisiin ei yhtä hyvin. Järjestelmän todettiin soveltuvan hyvin sellaisiin sovelluksiin, joissa on usealle käyttäjälle samanaikaisesti jaettuja resursseja sekä sellaisiin, joissa asiakasohjelman tilan pitäminen reaaliajassa palvelimen kanssa on erityisen tärkeää. Tällaisia sovelluksia ovat esimerkiksi pelit ja sosiaaliset sovellukset. Sen sijaan järjestelmä ei sovellu sellaisille sovelluksille, joissa resursseista vastaa yleensä vain yksi käyttäjä kerrallaan tai sellaisille, joissa resurssit muuttuvat vain harvoin, kuten esimerkiksi potilastietojärjestelmä tai sähköpostiohjelma.

6.5. Jatkokehitys

Mikäli järjestelmää haluttaisiin käyttää kaupallisiin tarkoituksiin, ei omien palvelinten käyttäminen luultavasti olisi järkevää. Mikäli liiketoiminta ei ole riittävän suurta eikä palvelinkokonaisuus riittävän iso, ei voida hyödyntää kustannussäästöjä, joita konesalien avulla voidaan saavuttaa. Tällöin palvelinkokonaisuuden ylläpito olisi todennäköisesti kalliimpaa kuin vastaavien laskentaresurssien vuokraaminen muualta. Tästä syystä merkittävä osa kaikesta maailman palvelinkapasiteetista sijaitseekin suurissa konesaleissa, joiden resursseja vuokrataan eteenpäin.

Pilvipalvelut ovat kaupallisia verkossa toimivia alustoja, joita pilvipalveluntarjoajat tarjoavat asiakkailleen. Näiden alustojen avulla asiakkaat voivat suorittaa omia ohjelmiaan ja tallentaa tietojaan vuokraamalla laskentaresursseja ja tallennuskapasiteettia palveluntarjoajilta. Pilvipalveluntarjoaja huolehtii tarvittavasta fyysisestä palvelin- ja verkkoinfrastruktuurista, joten järjestelmän kehittäjän tehtäväksi jää keskittyä järjestelmän loogiseen toimintaan. Pilvipalvelualustojen ansiosta saavutetaan mittakaavaetuja, ja asiakas maksaa vain käyttämistään resursseista. Lisäksi pilvipalvelun avulla uuden palvelininfrastruktuurin rakentamiseen liittyvät investoinnit ja riskit voidaan välttää kokonaan. [57]

Koska yhdellä alustalla voi olla suuri määrä asiakkaita ja heidän järjestelmiään, alustat jakavat fyysisiä palvelinresursseja virtuaalisessa muodossa. Tämä tekee hajautetun järjestelmän kehittämisen pilvipalveluympäristössä yksinkertaiseksi ja kustannustehokkaaksi, sillä alusta tarjoaa laskentaresursseja esimerkiksi virtuaalikoneiden ja konttien muodossa [58]. Lisäksi pilvipalvelut tarjoavat usein monipuolisen valikoiman valmiita palveluita, kuten tietokanta-, jono- ja välimuistipalveluita.

Vaikka tässä työssä käsiteltyä hajautettua järjestelmää ei ole toteutettu pilvipalvelualustalle, sen siirtäminen esimerkiksi Amazonin AWS-pilvipalveluun olisi mahdollista varsin pienellä vaivalla. AWS-pilvipalvelu sisältää valmiit resurssit täysin samojen konttien suorittamiseen sekä vastaavaan verkon ja kuormantasainten määrittämiseen [59]. Jopa taustapalveluita on mahdollista käyttää täsmälleen samalla ohjelmistolla, sillä PostgreSQL-tietokanta, Redis-yhteensopiva välimuisti ja RabbitMQ-jono ovat saatavilla valmiina palveluina [59]. Järjestelmän logiikkaa, ohjelmistoja tai edes arkkitehtuuria ei siis tarvitse muuttaa ollenkaan, ja järjestelmän

siirtämiseksi tehtävä työ on ainoastaan palveluiden ja verkkojen määrittelyä ja käyttöönottoa.

7. YHTEENVETO

Tässä työssä tutkittiin, miten suorituskykyinen palvelinohjelmisto reaaliaikaiselle web-sovellukselle voitaisiin suunnitella ja toteuttaa. Lisäksi työssä käsiteltiin järjestelmän suunnittelufilosofiaa ja sen vaikutusta järjestelmän arkkitehtuuriin. Järjestelmälle määriteltiin arkkitehtuurillisia vaatimuksia, joiksi valittiin reaaliaikaisuus, interaktiivisuus, skaalautuvuus ja suorituskyky. Lisäksi järjestelmän teknisenä vaatimuksena on kyky toimia web-ympäristössä.

Järjestelmän web-teknologiavaatimusta pohjustettiin avaamalla selaimen toimintaa, vahvuuksia ja rajoituksia. Työssä esiteltiin järjestelmän toteuttamisen mahdollistavat teknologiat, sekä näistä teknologioista rakennetut rajapinnat, joiden avulla järjestelmän käyttäminen on mahdollista. Lisäksi kuvattiin järjestelmän muut osat toteuttavat taustapalvelut, näiden rooli kokonaisuuden toiminnassa sekä niiden tyyppiin liittyviä rajoituksia järjestelmää rakennettaessa. Myös asiakasohjelman rooli, siihen liittyvät vaatimukset ja rajoitukset kuvattiin.

Järjestelmän suunnittelutavoitteiden saavuttamisen arviointia varten rakennettiin testiympäristö, jossa testisovellusta suoritettiin. Eri järjestelmän osien kommunikointia ja toimintaa hallinnoiva orkestraattorijärjestelmä mahdollisti kolmen suorituskykyisen palvelintietokoneen käyttämisen testijärjestelmän kuormituksen testaamisessa. Lisäksi orkestraattori mahdollisti erilaisten palveluiden hyvän skaalauksen hallinnan, ja kaikki palvelut tietokantaa lukuun ottamatta skaalattiin horisontaalisesti koneiden välillä.

Kuormantestaus toteutettiin simuloimalla suuria määriä käyttäjiä samanaikaisesti erityisellä kuormantestausohjelmistolla. Testejä suoritettiin parhaimman rajapintapalvelun skaalaamisen löytämiseksi, jotta järjestelmän suurin mahdollinen suorituskyky voitaisiin saavuttaa. Suorituskykyä mitattiin käsiteltyjen viestien määränä sekunnissa. Sen lisäksi testikäyttäjien määrää seurattiin, vaikkakin arvioitiin ettei testikäyttäjä ole vertailukelpoinen todellisen käyttäjän kanssa.

Testituloksista, sekä ennen testejä tehtyjen kokeilujen avulla, havaittiin järjestelmän suorituskykyyn olennaisesti vaikuttavia seikkoja tietokannan käytössä. Testituloksien perusteella arvioitiin paras rajapintapalvelun skaalaus sekä käsiteltiin eri seurauksia järjestelmässä, kun käyttäjämäärä kasvaa liian suureksi. Tietokannan skaalaamisen tapoja ja sen vaikutuksia suorituskyvyssä arvioitiin, sekä sitä, millaisilla toimilla testijärjestelmän suorituskykyä olisi voitu parantaa.

Testien interaktiivisen luonteen vuoksi, ja koska kuormantestaus onnistui, järjestelmän interaktiivisuusvaatimus katsottiin täyttyneeksi. Testitulosten perustella järjestelmän suorituskyky parani skaalaamisen myötä ja järjestelmä siis toteutti skaalautumisvaatimuksen. Parhaimmillaan järjestelmä pystyi käsittelemään niin suuren määrän pyyntöjä sekunnissa, että myös suorituskykyvaatimus katsottiin täyttyneen. Testijärjestelmän avulla kaikki vaatimukset saatiin siis täytettyä.

Tutkimuskysymyksiin vastaaminen onnistui myös hyvin. Järjestelmän suunnittelufilosofian todettiin sopivan käyttötarkoitukseensa, siitä huolimatta, että se monimutkaistaa asiakasohjelman toimintaa. Arkkitehtuurin todettiin mahdollistavan vaatimusten mukaisen sovelluksen kehittämisen, ja sen todettiin myös pystyvän skaalautumaan suorituskyvyn parantamiseksi. Lisäksi tunnistettiin erilaisia arkkitehtuurin kanssa yhteensopivia ja vähemmän yhteensopivia sovelluksia.

Lopuksi käsiteltiin jatkokehityksen näkymiä järjestelmän pilvipalveluun siirtämisen muodossa. Arvioitiin, että järjestelmän siirtäminen Amazonin AWS-pilvipalveluun

olisi varsin helppoa komponenttien yhteensopivuuden vuoksi. AWS-pilvipalvelu tarjoaa teknisesti hyvin yhtäläisiä palveluita suunnitellun järjestelmän kanssa, ja järjestelmän siirtämiseksi tehtävä työ olisi ainoastaan palveluiden ja verkkojen määrittelyä ja käyttöönottoa.

8. VIITTEET

[1] Client-Server Model (2022). URL: https://www.geeksforgeeks.org/client-server-model/. Haettu 21.5.2023.

[2] Schussel G. (2001), Client/Server: Past, Present and Future. URL: http://ciains.info/elearning/Solutions/Architecture/ClientServer/CS-past,presentFurure.pdf.

[3] Taivalsaari A., Mikkonen T., Ingalls D. & Palacz K. (2008) Web Browser as an Application Platform. Teoksessa: 2008 34th Euromicro Conference Software Engineering and Advanced Applications, ss. 293–302.

[4] Progressive Web Apps (PWA) (2022). URL: https://techradar.softwareag.com/technology/progressive-web-apps/. Haettu 18.12.2022.

[5] Jomppanen J. (2020) Skaalautuva palvelinarkkitehtuuri reaaliaikaiselle selainpelille. kandidaatintyö, Oulun yliopisto.

[6] Tanenbaum A.S. & Van Steen M. (2016) A Brief Introduction to Distributed Systems. Computing 98, ss. 967–1009. DOI: https://doi.org/10.1007/s00607-016-0508-7.

[7] Coulouris G., Dollimore J., Kindberg T. & Blair G. (2011) Distributed Systems: Concepts and Design. Addison-Wesley Publishing Company, USA, 5 p.

[8] Michael M., Moreira J.E., Shiloach D. & Wisniewski R.W. (2007) Scale-up x Scale-out: A Case Study using Nutch/Lucene. Teoksessa: 2007 IEEE International Parallel and Distributed Processing Symposium, ss. 1–8. DOI: https://doi.org/10.1109/IPDPS.2007.370631.

[9] Qu C., Calheiros R.N. & Buyya R. (2018) Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. ACM Computing Surveys 51. URL: https://dl.acm.org/doi/abs/10.1145/3148149.

[10] Cardellini V., Casalicchio E., Colajanni M. & Yu P.S. (2002) The State of the Art in Locally Distributed Web-Server Systems. ACM Computing Surveys 34, ss. 263–311. URL: https://dl.acm.org/doi/abs/10.1145/508352.508355.

[11] Linthicum D.S. (2009) Cloud Computing and SOA Convergence in Your Enterprise: A Step-by-Step Guide. Addison-Wesley Professional, 1 p.

[12] Comparison - Centralized, Decentralized and Distributed Systems (2023). URL: https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems/. Haettu 20.5.2023.

[13] Stephens R. (2015) Beginning Software Engineering. Wrox Press Ltd., GBR, 1 p.

[14] Blinowski G., Ojdowska A. & Przybyłek A. (2022) Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. IEEE Access 10, ss. 20357–20374.

[15] Types of Distributed Systems (2023). URL: https://www.geeksforgeeks.org/types-of-distributed-system/. Haettu 20.5.2023.

[16] Native Apps vs. Web Apps (2021). URL: https://www.lifewire.com/native-apps-vs-web-apps-2373133. Haettu 14.12.2021.

[17] Fielding R.T., Gettys J., Mogul J.C., Nielsen H.F., Masinter L., Leach P.J. & Berners-Lee T. (1999) Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc2616.txt, http://www.rfc-editor.org/rfc/rfc2616.txt.

[18] Long polling (2022). URL: https://javascript.info/long-polling. Haettu 14.12.2021.

[19] Fette I. & Melnikov A. (2011) The WebSocket Protocol. RFC 6455, RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc6455.txt, http://www.rfc-editor.org/rfc/rfc6455.txt.

[20] Stack Overflow Developer Survey (2021). URL: https://insights.stackoverflow.com/survey/2021. Haettu 18.12.2022.

[21] ECMA International (2022) ECMAScript 2022 language specification. ECMA 262, ECMA International. URL: https://www.ecma-international.org/wp-content/uploads/ECMA-262_13th_edition_june_2022.pdf.

[22] TypeScript Documentation (2022). URL: https://www.typescriptlang.org/docs/handbook/. Haettu 18.12.2022.

[23] Rapo J. (2020), TypeScriptin hyödyllisyys JavaScript-ohjelmistokehityksessä. URL: https://helda.helsinki.fi/bitstream/handle/10138/313593/Rapo_Jani_Pro_gradu_2020.pdf.

[24] Modern Web Development with TypeScript, Babel, and Webpack (2021). URL: https://blog.bitsrc.io/modern-web-development-with-typescript-babel-and-webpack-36c5c58b36e. Haettu 18.12.2022.

[25] WebAssembly (2022). URL: https://webassembly.org/. Haettu 18.12.2022.

[26] Bishop M. (2022) HTTP/3. RFC 9114, RFC Editor.

[27] Perna G., Trevisan M., Giordano D. & Drago I. (2022) A first look at HTTP/3 adoption and performance. Computer Communications 187, ss. 115–124. URL: https://www.sciencedirect.com/science/article/pii/S0140366422000421.

[28] Chromium OS (2022). URL: https://www.chromium.org/chromium-os/. Haettu 18.12.2022.

[29] The web is for everyone: Our vision for the evolution of the web (2022). URL: https://blog.mozilla.org/en/mozilla/mozilla-webvision-future-of-web/. Haettu 18.12.2022.

[30] Web Real-Time Communications (WebRTC) transforms the communications landscape (2021). URL: https://www.w3.org/2021/01/pressrelease-webrtc-rec.html.en. Haettu 25.5.2023.

[31] Boström H., Castelli F., Jennings C. & Bruaroey J.I. (2023) WebRTC: Real-Time Communication in Browsers. W3C recommendation, W3C. Https://www.w3.org/TR/webrtc/.

[32] webtorrent/webtorrent - Streaming torrent client for the web (2023). URL: https://github.com/webtorrent/webtorrent. Haettu 26.5.2023.

[33] mappum/webcoin - SPV Bitcoin client for Node.js and the browser (2023). URL: https://github.com/mappum/webcoin. Haettu 26.5.2023.

[34] Microsoft eCDN (2023). URL: https://www.microsoft.com/fi-fi/microsoft-teams/ecdn. Haettu 26.5.2023.

[35] WebRTC NAT Traversal Methods: A Case for Embedded TURN (2022). URL: https://www.liveswitch.io/blog/webrtc-nat-traversal-methods-a-case-for-embedded-turn. Haettu 26.5.2023.

[36] What is a Container? (2022). URL: https://www.docker.com/resources/what-container. Haettu 3.3.2022.

[37] What Is Container Orchestration? (2018). URL: https://blog.newrelic.com/engineering/container-orchestration-explained/. Haettu 3.3.2022.

[38] Configuration as Code: How to Streamline Your Pipeline (2020). URL: https://www.perforce.com/blog/vcs/configuration-as-code/. Haettu 25.1.2023.

[39] What Is Load Balancing? (2015). URL: https://www.nginx.com/resources/glossary/load-balancing/. Haettu 25.1.2023.

[40] Load Balancing, Affinity, Persistence, Sticky Sessions: What You Need to Know (2015). URL: https://www.haproxy.com/blog/load-balancing-affinity-persistence-sticky-sessions-what-you-need-to-know/. Haettu 25.1.2023.

[41] What is REST (2022). URL: https://restfulapi.net/. Haettu 25.1.2023.

[42] Why you don't need Socket.IO (2016). URL: https://codeburst.io/why-you-don-t-need-socket-io-6848f1c871cd. Haettu 25.1.2023.

[43] Thread Pools in NGINX Boost Performance 9x! (2015). URL: https://www.nginx.com/blog/thread-pools-boost-performance-9x/. Haettu 25.1.2023.

[44] API synchronous vs asynchronous in business (2022). URL: https://medium.com/transparent-data-eng/api-synchronous-vs-asynchronous-in-business-87ec9f442ab7. Haettu 25.1.2023.

[45] Overview | Kubernetes (2023). URL: https://kubernetes.io/docs/concepts/overview/. Haettu 16.4.2023.

[46] Python's Design Goals: A Conversation with Guido van Rossum, Part II (2003). URL: https://www.artima.com/articles/pythons-design-goals. Haettu 3.3.2022.

[47] Advantages and Disadvantages of the Python Programming Language (2019). URL: https://learnpython.com/blog/why-is-python-so-popular. Haettu 3.3.2022.

[48] Sanner M.F. et al. (1999) Python: a programming language for software integration and development. Journal of Molecular Graphics and Modelling 17, ss. 57–61.

[49] Advantages and Disadvantages of the Python Programming Language (2019). URL: https://learnpython.com/blog/python-programming-advantages-disadvantages. Haettu 3.3.2022.

[50] 11 Advantages of Django: Why You Should Use It (2022). URL: https://pythonistaplanet.com/advantages-of-django/. Haettu 19.2.2023.

[51] The Django Book: Chapter 5: Models (2009). URL: hhttps://web.archive.org/web/20160902130823/http://www.djangobook.com/en/2.0/chapter05.html#the-mtv-or-mvc-development-pattern. Haettu 19.2.2023.

[52] Why choose Django REST framework for API development (2022). URL: https://p-o.co.uk/tech-articles/why-choose-django-rest-framework-for-api-development/. Haettu 19.2.2023.

[53] Introduction - Channels 4.0.0 Documentation (2022). URL: https://channels.readthedocs.io/en/stable/introduction.html. Haettu 19.2.2023.

[54] PostgreSQL: About (2023). URL: https://www.postgresql.org/about/. Haettu 12.2.2023.

[55] ORM: Rethinking Data as Objects (2018). URL: https://blog.yellowant.com/orm-rethinking-data-as-objects-8ddaa43b1410. Haettu 3.3.2022.

[56] What is Locust? (2023). URL: https://docs.locust.io/en/stable/what-is-locust.html. Haettu 20.4.2023.

[57] What is cloud computing? Everything you need to know about the cloud explained (2022). URL: https://www.zdnet.com/article/what-is-cloud-computing-everything-you-need-to-know-about-the-cloud/. Haettu 17.4.2023.

[58] What Is Distributed Computing: Definition and Examples (2021). URL: https://www.ridge.co/blog/what-is-distributed-computing/. Haettu 17.4.2023.

[59] AWS Cloud Products (2023). URL: https://aws.amazon.com/products/. Haettu 21.5.2023.