Aleksi Tani

Tapahtumien ja tilausten suorituskyvyltään nopeampi yhdistäminen hajautetussa mikropalveluarkkitehtuurissa

Tietotekniikan pro gradu -tutkielma

24. huhtikuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Aleksi Erkki Oskari Tani

Yhteystiedot: aleksitani@gmail.com

Ohjaajat: Jukka-Pekka Santanen ja Kimmo Löytänä

Työn nimi: Tapahtumien ja tilausten suorituskyvyltään nopeampi yhdistäminen hajautetussa mikropalveluarkkitehtuurissa

Title in English: Improving Subscription Event Matching in Distributed Microservice Architecture

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 61

Tiivistelmä: Tapahtumien ja tilauksien yhteensopivuuden tarkistaminen eli yhdistäminen on keskeinen ongelma suuren kokoluokan hajautettujen Publish/Subscribe -kommunikointimalliin perustuvien tietojärjestelmien kokonaissuorituskyvyssä. Tutkielman tavoitteena oli parantaa olemassa olevan mikropalveluarkkitehtuuriin perustuvan tietojärjestelmän suorituskykyä. Tutkielmassa toteutettiin prototyyppi tapahtumien ja tilausten tehokkaammalle yhdistämiselle nykyisen toteutuksen rinnalle. Toteutettujen muutosten onnistumista arvioitiin suorituskykytestauksella. Tärkeimpinä onnistumisen mittareina toimivat suoritinkuorma ja yhdistämisalgoritmin suoritusnopeus. Tutkielmassa kehitetty prototyyppi oli suorituskykytestauksen tulosten perusteella suorituskyvyltään nopeampi, tarkasteltujen mittareiden perusteella. Suoritinkuorman keskiarvo pieneni ja yhdistämisalgoritmin keskinopeudet paranivat suurilta osin.

Avainsanat: Algoritmi, AMQP, Kommunikointimalli, Mikropalveluarkkitehtuuri, Muistinvarainen, Publish/Subscribe, Suorituskykytestaus, Tapahtuma, Tietorakenteet, Viestijono.

Abstract: Event matching is a key issue in the overall performance of large-scale distributed information systems based on Publish/Subscribe communication paradigm. The aim

of the thesis was to improve the performance of an existing information system based on microservice architecture. In the thesis a prototype was implemented for more efficient event matching alongside the current implementation. The success of implemented changes was evaluated by performance testing. The most important indicators of success used were CPU load and the execution speed of aggregation algorithm. Prototype implemented in this thesis was faster based on performance test results. The average CPU load was smaller, and the execution speed of aggregation algorithm was faster for the most parts.

Keywords: Algorithms, AMQP, Communication paradigm, Data models, Event, In-memory, Message Queue, Microservice architecture, Performance testing, Publish/Subscribe.

Termiluettelo

AMQP	on avoimen standardin sovelluskerroksen protokolla sanomakeskeiselle väliohjelmistolle.
Asynkroninen	on ajasta ja paikasta riippumaton kommunikointitapa.
Gatling	on avoimen lähdekoodin suorituskykytestauskehys tietojärjestelmien suorituskyvyn testaamiseen.
Hazelcast	on Java-pohjainen avoimen lähdekoodin hajautettu muistinvarainen tietovarasto.
Julkaisija	on Publish/Subscribe -kommunikointimalliin perustuva tapahtumien tai viestien julkaisija.
Käyttäjä	on tietojärjestelmän loppukäyttäjä.
Mikropalveluarkkitehtuuri	on arkkitehtoninen tapa hajauttaa tietojärjestelmän sovelluskomponentit omiksi itsenäisiksi palveluiksi.
Muistinvarainen	tarkoittaa tiedon tallennusta ja laskentaa, jossa käytetään tietokoneen keskusmuistia kiintolevyn sijasta.
Mikropalvelu	on itsenäinen palvelukomponentti mikropalveluarkkitehtuuriin perustuvassa tietojärjestelmässä.
Publish/Subscribe	on kommunikointimalli, joka perustuu tilaajiin ja julkaisijoihin.
RabbitMQ	on avoimen lähdekoodin viestinvälittäjäohjelmisto, joka toteuttaa muun muuassa AMQP-protokollan.
Sanomakeskeinen väliohjelmisto	on ohjelmisto- tai laitteistoinfrastruktuuri, joka tukee viestien lähettämistä ja vastaanottamista hajautettujen järjestelmien välillä.

Suorituskyky	tarkoittaa ohjelmiston suorituksen ajallista kestoa verrattuna sille asetettuihin tavoitteisiin.
Testauskehys	on joukko ohjeita tai sääntöjä, joita käytetään testitapausten luomiseen ja suunnitteluun.
Tietovarasto	on tietokanta, tietokokoelmien jatkuvaa tallentamista ja hallintaa varten.
Tilaaja	on Publish/Subscribe -kommunikointimalliin perustuva tapahtumien tai viestien tilaaja.
Viestijono	on asynkronisten palvelujen sekä prosessien välinen kommunikaatiotapa, jota käytetään muun muuassa mikropalveluarkkitehtuureissa.
Viestinvälittäjäohjelmisto	on ohjelmisto, joka muuntaa viestin lähettäjän muodollisesta viestintäprotokollasta vastaanottajan muodolliseen viestiprotokollaan. Yleensä osa sanomakeskeistä väliohjelmistoa.
Välimuisti	on pieni ja nopea muisti, johon tallennetaan yleensä tietoa, johon viitataan pian tai usein. Välimuistilla pyritään nopeuttamaan toimintoja.

Kuviot

Kuvio 1. Monoliittisen arkkitehtuurin ja mikropalveluarkkitehtuurin ero. 8
Kuvio 2. Viestinvälittäjä. 10
Kuvio 3. Mikropalveluiden välinen tapahtumien kuuntelu. 26
Kuvio 4. Pelkistetty yleiskuva järjestelmän korkean tason arkkitehtuurista. 27
Kuvio 5. Testin aikana lähetetyt tapahtumat testikäyttäjille nykyisessä toteutuksessa. 40
Kuvio 6. Testin aikana lähetetyt tapahtumat testikäyttäjille uudessa prototyypissä. 40
Kuvio 7. Suoritinkuormien keskiarvo palvelusolmukohtaisesti. 41
Kuvio 8. Suoritinkuormien maksimiarvot palvelusolmukohtaisesti. 42
Kuvio 9. APN-2-palvelusolmun yhdistämisalgoritmin keskinopeudet toteutusten välillä. 45
Kuvio 10. APN-2-palvelusolmun yhdistämisalgoritmin maksimiarvot toteutusten välillä. 45

Taulukot

Taulukko 1. Istunnon sisältämän tietorakenteet. 24
Taulukko 2. Tilaus (engl. *subscription*). 24
Taulukko 3. Yhdistämisalgoritmin suoritusnopeuksien keskiarvot eri toteutusten välillä. 43
Taulukko 4. Yhdistämisalgoritmin suoritusnopeuksien maksimiarvot eri toteutusten välillä. 44

Sisältö

1 JOHDANTO ... 1

2 TAUSTAT, TARPEET JA TAVOITTEET ... 4
2.1 Tietojärjestelmän taustat ... 4
2.2 Tutkielman tarpeet ja tavoitteet ... 5
2.3 Tutkimusmenetelmä ... 6

3 MIKROPALVELUARKKITEHTUURI ... 7
3.1 Mikropalveluarkkitehtuuri ja mikropalvelu ... 7
3.2 Mikropalveluiden välinen tapahtumien kuuntelu ja kommunikointi ... 9
3.3 Advanced Message Queuing Protocol (AMQP) ... 11
3.4 RabbitMQ-viestinvälittäjä ... 11

4 PUBLISH/SUBSCRIBE -TOIMINTAPERIAATTEET ... 13
4.1 Publish/Subscribe kommunikointimallina ... 13
4.2 Aiheperusteinen Publish/Subscribe ... 14
4.3 Sisältöperusteinen Publish/Subscribe ... 15
4.4 Suodattimet Publish/Subscribe -järjestelmässä ... 15

5 AIEMPI TUTKIMUS ... 17
5.1 Publish/Subscribe -järjestelmät ... 17
5.2 Tapahtumien ja tilausten yhdistäminen ... 18

6 SUORITUSKYKYTESTAUS ... 20
6.1 Suorituskykytestauksen tyyppejä ... 20
6.2 Suorituskykytestaus yleisesti ... 21
6.3 Gatling-suorituskykytestauskehys ... 21

7 TIETOJÄRJESTELMÄN NYKYINEN TOTEUTUSRATKAISU ... 23
7.1 Tilaajat ja tilaukset ... 23
7.2 Tietojärjestelmän tapahtumat ... 25
7.3 Tietojärjestelmän mikropalvelut ... 26
7.4 Tilauspalvelun toteutusratkaisu ... 28
7.5 Suorituskykytestit ... 31

8 PROTOTYYPIN TOTEUTUSRATKAISUT ... 32
8.1 Prototyyppi – Tapahtumien ja tilausten yhdistäminen ... 32
8.2 Aiheperusteisten tilausten käsittely ... 33
8.3 Suodattimien käsittely ... 34
8.4 Tulosten analyysi ... 37

9 SUORITUSKYKYTESTAUSKERTOJEN TULOKSET JA NIIDEN ANALYYSI 39

9.1 Suorituskykytestauksen tulokset 39
9.2 Generoitujen tapahtumien lukumäärä 39
9.3 Tilauspalvelun suoritinkuorma 41
9.4 Tapahtumien ja tilausten yhdistämisalgoritmin suoritusnopeus 42

10 YHTEENVETO 47

LÄHTEET 49

1 Johdanto

Tämän tutkielman tarkoituksena on parantaa Patria Systemsin kehittämän tietojärjestelmän sisältämän, **tilauspalvelu**-sovelluskomponentin **suorituskykyä** hajautetussa palveluperustaisessa **mikropalveluarkkitehtuurissa**. Tilauspalvelun (engl. *subscription management*) vastuulla on järjestelmän generoimien **tapahtumien** (engl. *event*) ja käyttäjien luomien **tilausten** (engl. *subscription*) **yhdistäminen**. Tutkielman tavoitteena on parantaa tapahtumien ja tilausten yhdistämistä eli yhteensopivuuden tarkistamista (engl. *event matching*), käyttäen apuna lähdekirjallisuutta tapahtumien tarkistukseen liittyen. Tutkielmassa kehitettävän prototyypin on tarkoitus tarjota nopeampi ratkaisu tapahtumien sekä tilausten käsittelyyn ja näin pyrkiä parantamaan tietojärjestelmän kokonaissuorituskykyä.

Tietojärjestelmä perustuu hajautettuun **mikropalveluarkkitehtuuriin**, jossa eri sovelluslogiikasta vastaavat sovelluskomponentit ovat itsenäisiä sekä löyhästi liitettyjä osia järjestelmän kokonaisarkkitehtuurissa. Järjestelmän ohjelmointirajapinnat ovat pääosin toteutettu HTTP-protokollaan perustuvalla REST-arkkitehtuurimallilla.

Suuri osa nykypäivän sovelluksista ovat hajautettuja järjestelmiä [Salah et al. 2016]. **Mikropalveluihin** perustuvasta **arkkitehtuurista** on tullut uusi tapa hajauttaa sovelluksia pienempiin sekä ylläpidettävimpiin palveluihin, jotka toimivat itsenäisesti ja kommunikoivat keskenään [Aderaldo et al. 2017]. Mikropalveluiden hyötyjä verrattuna perinteiseen **monoliittiseen** arkkitehtuuriin ovat joustavuus, ketteryys sekä hajautettavuus. Mikropalveluarkkitehtuurin haasteita taas on esimerkiksi järjestelmän kokonaisarkkitehtuurin monimutkaisuus ja sen ymmärtäminen, kun järjestelmä kasvaa tarpeeksi isoksi. Yksittäisen mikropalvelun kasvaessa liian isoksi se alkaa käyttäytyä kuin monoliittinen järjestelmä. Kun taas mikropalvelun ollessa liian pieni, kasvaa alun perin yksinkertaisen tehtävän suorittamisen monimutkaisuus sekä käsittely [Salah et al. 2016].

Tutkielman kohteena olevan tietojärjestelmän **mikropalveluiden** välinen **tapahtumien** kuuntelu on toteutettu **RabbitMQ**-viestinvälitystekniikan avulla, joka pohjautuu **AMQP**-protokollaan. Advanced Message Queuing Protocol (AMQP) mahdollistaa sovellusten

lähettää ja vastaanottaa viestejä sekä se sallii määrittää mitä ja miten viestejä välitetään [Fernandes et al. 2013].

Viestinvälitystekniikan **kommunikointimallina** on Publish/Subscribe -tyylinen malli. Siinä **tilaajat** (engl. *subscribers*) ilmaisevat kiinnostuksena saada ilmoituksia tietyistä **tapahtumista** (engl. *event*), joita eri mikropalvelut **tuottavat** tai **julkaisevat** (engl. *publish*) [Setty 2015, s. 1]. **Mikropalveluarkkitehtuurissa** tapahtumien tuottamisesta vastaavat yleensä kyseisestä sovelluslogiikan alueesta vastaavat mikropalvelut. Publish/Subscribe -järjestelmissä tilaajina toimivat yleensä **asiakasohjelmistot**. Asiakasohjelmisto voi olla esimerkiksi Windows-sovellus tai internet-selain. Publish/Subscribe -järjestelmissä tilauksien ja tapahtumien yhteensopivuuden tarkistaminen (engl. *event matching*) on kriittinen osa tietojärjestelmän kokonaissuorituskykyä [Yang, Fan, ja Jiang 2016].

Tutkielman **tutkimusmenetelmä** on osin **suunnittelutieteellistä** tutkimusta sekä osin **konstruktiivista** tutkimusta. Suunnittelutieteellinen tutkimus tukee tutkielmaa, koska se keskittyy muun muuassa suunniteltujen asioiden kehittämiseen ja suorituskykyyn nimenomaisella tarkoituksella parantaa tietyn artefaktin toiminallista **suorituskykyä**. Konstruktiivisessa tutkimuksessa taas luodaan alkuperäiseen ongelmaan ratkaisu eli konstruktio sekä testataan luodun konstruktion toimivuus.

Tutkielman tuloksissa tarkastellaan ensisijaisesti tietojärjestelmän **suorituskykyä** suurilla käyttäjämäärillä. Tutkielman teoriaosio keskittyy **mikropalveluarkkitehtuuriin** sekä viestinvälitykseen mikropalveluiden välillä. Tämän lisäksi teoriaosiossa keskitytään Publish/Subscribe -kommunikointimalliin, joka on myös keskeisessä osassa tutkimuksen kohteena olevassa tietojärjestelmässä sekä suorituskykytestaukseen yleisellä tasolla. Näiden lisäksi teoriaosiossa käydään läpi lähdemateriaaleihin perustuvaa aikaisempaa tutkimustietoa liittyen **tapahtumien** ja **tilausten** yhdistämiseen tapahtumapohjaisissa hajautetuissa järjestelmissä.

Tämä tutkielma jakautuu seuraavasti. Luvussa 2 esitellään tutkielman taustat ja tarpeet, tavoitteet ja tutkimusmenetelmä. Luvussa 3 esitellään mikropalveluarkkitehtuuri sekä mikropalveluiden välinen viestinvälitystekniikka. Luvussa 4 esitellään Publish/Subscribe -kommunikointimalli sekä tapoja toteuttaa se. Luvussa 5 perehdytään tutkielmassa käytet-

tyyn aineistoon ja kirjallisuuteen sekä tarkastellaan aiempaa tutkimusta liittyen tapahtumien ja tilausten yhdistämiseen. Luvussa 6 esitellään suorituskykytestausta yleisesti sekä tutkielmassa hyödynnettävien testien suorituskykytestauskehys. Luvussa 7 esitellään tietojärjestelmän nykyinen toteutus yleisellä tasolla, siihen liittyvät tekniset haasteet sekä nykyisen tietojärjestelmän olemassa olevia suorituskykytestejä. Luvussa 8 esitellään tutkielmassa kehitettävä prototyyppi ja sen tuomat muutokset sekä tulosten analysointi. Luvussa 9 esitellään tutkimuksen tulokset ja niiden analysointi. Luvussa 10 esitellään tutkielman yhteenveto, pohditaan tavoitteiden onnistumista sekä mahdollista jatkokehitystä.

2 Taustat, tarpeet ja tavoitteet

Tässä luvussa esitellään tutkielman kohteena olevan tietojärjestelmän taustat ja tarpeet, tutkielman tavoitteet sekä tutkimusmenetelmä.

2.1 Tietojärjestelmän taustat

Tämän tutkielman kohteena on tietojärjestelmän **tilauspalvelu**-sovelluskomponentti, joka toimii järjestelmässä itsenäisenä mikropalveluna. Tilauspalvelun tarkoituksena on kuunnella muiden mikropalveluiden **tapahtumia** ja yhdistää saatu data sekä lähettää se eteenpäin siitä kiinnostuneille käyttäjille. Tapahtumalla tarkoitetaan järjestelmän jonkin **mikropalvelun** lähettämää dataa jostain muutoksesta. Muutos voi olla esimerkiksi jonkin järjestelmän entiteetin luonti- tai muokkausoperaatio.

Järjestelmään kirjautuneilla loppukäyttäjillä voi olla yhtäaikaisesti useita eri aktiivisia **istuntoja** (engl. *session*). Aktiivisella istunnolla tarkoitetaan käyttäjän kirjautumista asiakasohjelmistoon, joka voi olla esimerkiksi internet-selain tai Windows-sovellus. **Tilauspalvelu** pitää kirjaa kaikista istunnoista sekä vastaa **mikropalveluista** tulleen **tapahtuman** edelleen lähettämisestä istuntokohtaisesti. Tilauspalvelun lähettämät tapahtumat määräytyvät istuntokohtaisten **suodattimien** ja **tilausten** avulla.

Käyttäjä voi aktiivisessa **istunnossaan** seurata jonkin tietyn entiteetin muutoksia ja tilauspalvelun vastuulla on välittää kyseiset muutokset istunnolle. Entiteetin muutoksien seuraaminen tapahtuu **tilausten** ja **suodattimien** avulla. Suodattimet toimivat järjestelmässä niin sanottuina epäsuorina tilauksina, joilla järjestelmä lähettää tiedon tapahtumasta, jos se sopii käyttäjän suodattimeen. Suodattimista voi helposti kehkeytyä pullonkaula suurien käyttäjämäärien luodessa yhä enemmän monimutkaisia suodattimia [Xie, Navathe ja Prasad 2005].

Tilauspalvelu toimii keskitetysti niin sanottuna **väliohjelmistona** tapahtumien edelleen lähettämisessä ja vastaa **tapahtumien** sekä **tilausten** yhteensopivuudesta ja niiden yhdistämisestä. Kuten mikä tahansa keskitetty sovelluskomponentti tai palvelu, niin tilauspalve-

lusta muodostuu tietojärjestelmän suorituskyvyn kannalta pullonkaula [P. R. Pietzuch ja Bacon 2002].

2.2 Tutkielman tarpeet ja tavoitteet

Tilauspalvelusta muodostuu helposti pullonkaula järjestelmän **suorituskyvyn** kannalta. Tietojärjestelmää varten on kehitetty **suorituskykytestit**, joilla saadaan ajantasainen kuva järjestelmän eri komponenttien suorituskyvystä. Suorituskykytestit kuvataan tarkemmin luvussa 7.5. Näillä testeillä on kuitenkin jo todettu, että suurilla istuntomäärillä tilauspalvelusta muodostuu **pullonkaula** järjestelmän kokonaissuorituskyvyn kannalta. Suorituskyvyn nopeuden lisäksi järjestelmässä on todettu muitakin pullonkauloja, mutta tässä tutkielmassa keskitytään nimenomaan suorituskyvyn nopeuden parantamiseen.

Vaikka kyseessä on **hajautettu** mikropalveluarkkitehtuuriin perustava **tietojärjestelmä**, niin **suorituskykyongelmat** keskittyvät suurimmilta osin tilauspalvelu-sovelluskomponenttiin. Hajautetusta arkkitehtuurista huolimatta on järjestelmän tarkkojen vaatimusten takia päädytty keskittämään **tilausten** sekä **suodattimien** hallinnointi **tilauspalveluun**, mikä sitoo kyseisen mikropalvelun tiukasti muuhun tietojärjestelmään.

Tietojärjestelmän nykyinen toteutus ei ole tarpeeksi tehokas suurien istuntomäärien kanssa. Nykyinen toteutus käy läpi kaikki aktiiviset **istunnot** löytääkseen kaikki muutoksista kiinnostuneet istunnot. Istuntojen määrän sekä palveluiden tuottaman datamäärän kasvaessa, **tilauspalveluun** kohdistuu huomattava rasitus. Tutkielman **tavoitteet** ovatkin seuraavat:

- Parantaa tilauspalvelun nykyistä toteutusta uudelleen suunnittelemalla suodattimien sekä tilausten tietorakenteet sekä niiden läpikäynti.

- Kehittää tapahtumien ja tilausten yhdistämiseen tehokkaampi toteutusratkaisu.

- Arvioida toteutettujen muutoksien onnistuminen suorituskykytestien avulla.

Jokaisella tietojärjestelmällä on omat vaatimukset **toiminnallisuuden** ja **arkkitehtuurin** näkökulmasta. Koska tässä tutkielmassa on kyse laajasta jo olemassa olevasta tietojärjestelmästä, on muutosten toteuttaminen pelkästään aikaisemman **lähdekirjallisuuteen** perus-

tuvan tutkimuksen perusteella haastavaa. Tätä tutkielmaa varten kerätyistä lähdekirjallisuuden tutkimuksista voidaan kuitenkin soveltaa tutkitusti tehokkaita **tietorakenteita** sekä **algoritmeja** tapahtumien ja tilausten **yhdistämiseen**.

2.3 Tutkimusmenetelmä

Tutkielman **tutkimusmenetelmä** perustuu osin suunnittelutieteelliseen tutkimukseen sekä osin konstruktiiviseen tutkimukseen, koska tutkielmassa on tarkoitus parantaa olemassa olevaan suorituskykyongelmaan ratkaisu. **Suunnittelutieteellinen** tutkimus keskittyy muun muuassa suunniteltujen asioiden kehittämiseen ja suorituskykyyn nimenomaisella tarkoituksella parantaa tietyn artefaktin toiminallista suorituskykyä. **Konstruktiivisen** tutkimuksen periaatteiden mukaan, tutkielmassa kehitetään olemassa olevaan ongelmaan ratkaisu sekä testataan luodun ratkaisun eli konstruktion toimivuus. Molempien tutkimusmenetelmien tavoitteena on luoda toimiva ratkaisu ongelmaan sekä soveltaa aiemmin tunnettuja ratkaisuja uudessa ympäristössä.

Design science eli **suunnittelutieteellinen** tutkimus on ongelmanratkaisuun perustuva tutkimusmenetelmä. Blogikirjoituksessaan [Jokinen 2021] kuvaa suunnittelutiedettä IT-alalla yleisesti hyödynnetyksi tutkimusotteeksi. Hän myös korostaa, että kuten tässä tutkielmassakin, niin konstruktiivisessa tutkimuksessa työ perustuu teoreettiseen perustaan, johon tukeutuen tutkielmassa toteutettava ratkaisu kehitetään. Vaikka suunnittelutieteellinen tutkimus perustuu erilaisten ratkaisujen suunnitteluun ja kokeiluun, niin tämän tutkielman puitteissa keskitytään vain yhden ratkaisun kokeiluun.

3 Mikropalveluarkkitehtuuri

Tässä luvussa käsitellään mikropalveluarkkitehtuurin yleistymistä, toimintaperiaatetta sekä mikropalveluiden välistä kommunikointia. Luvussa käsitellään myös AMQP-protokollaa ja siihen pohjautuvaa RabbitMQ-viestinvälittäjää.

3.1 Mikropalveluarkkitehtuuri ja mikropalvelu

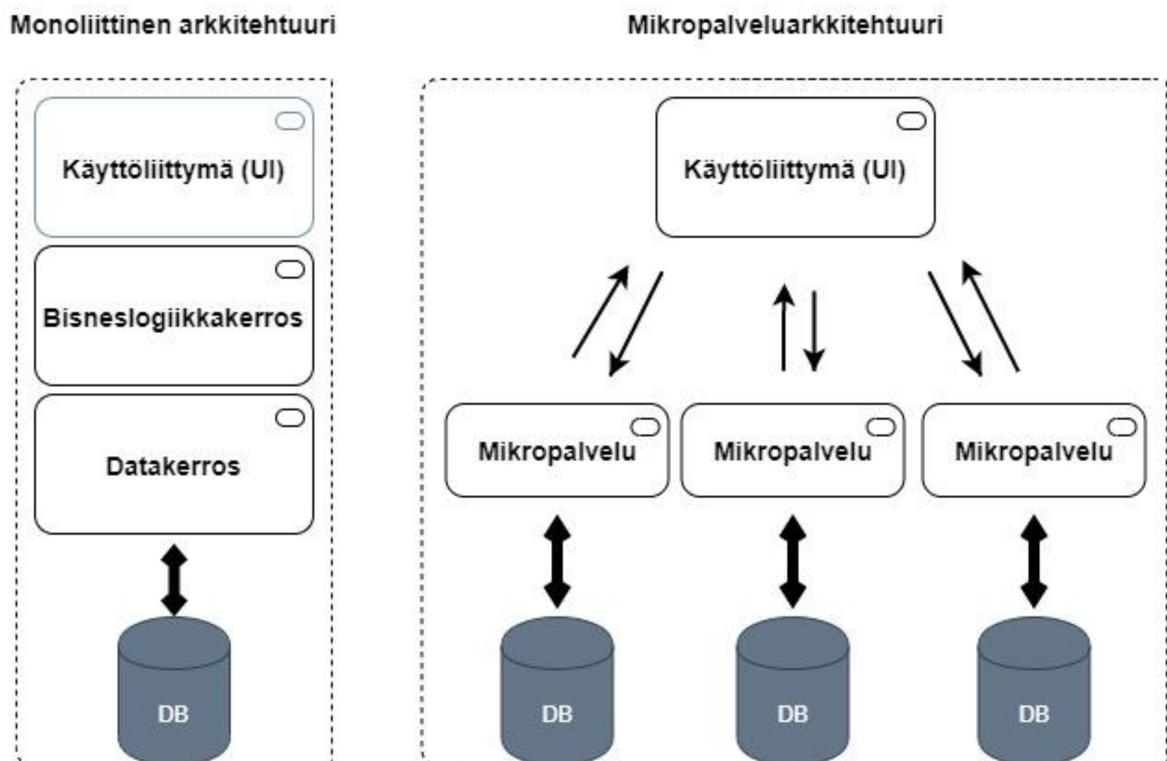
Mikropalveluarkkitehtuuri on lähteen [Di Francesco 2017] mukaan yleistynyt sekä akateemisessa tutkimuksessa että tietojärjestelmien kehityksessä. Vaikka mikropalveluarkkitehtuuri on saanut huomattavaa huomiota tutkimuksissa, niin tutkijoille ja arkkitehtuurin toteuttajille on edelleen vaikeaa saada selkeä näkemys olemassa olevista tutkimusratkaisuista arkkitehtuuria varten [Di Francesco, Lago, ja Malavolta 2019].

Iso osa nykypäivän moderneista sovelluksista ovat **hajautettuja** järjestelmiä. Hajautettujen järjestelmien yleistyminen alkoi nousta voimakkaasti, kun tuli tarve kehittää palveluita yhä useammalle ihmiselle reaaliaikaisesti verkon yli [Salah et al. 2016]. **Mikropalveluarkkitehtuurista** on tullut hallitseva arkkitehtoninen ratkaisu hajautettujen sekä **palveluperustaisten** järjestelmien toteutuksessa [Alshuqayran, Ali, ja Evans 2016]. Vaikka mikropalveluarkkitehtuurin suunnitteluperiaatteet ovat laajalti tunnistettu, niin monet osa-alueet ovat edelleen epäselviä tai tutkimattomia [Di Francesco 2017].

Mikropalveluarkkitehtuuri perustuu sovelluskomponenttien hajauttamiseen omiksi itsenäisiksi toteutuksikseen. Mikropalveluarkkitehtuuri keskittyy tiettyihin toteutusratkaisuihin kuten, pieniin itsenäisiin palveluihin, ketterien käytäntöjen soveltaminen kehittämiseen sekä käyttöönottoon, hajautettuun tiedonhallintaan ja palveluiden hajautettuun hallintaan [Di Francesco 2017]. Konferenssijulkaisussaan [Alshuqayran, Ali ja Evans 2016] nostavat myös esille, että mikropalveluarkkitehtuuriin perustuva lähestymistapa mahdollistaa, että organisaatiot sekä sovelluskehitystiimit ovat yleensä tuottavampia järjestelmän kehitysvaiheessa. Muutettaessa yksittäisen mikropalvelun sisäistä toteutusratkaisua, sillä ei ole vaikutusta muihin sitä kutsuviin palveluihin. Tämä johtuu siitä, että kommunikointi mikropalve-

luiden välillä tapahtuu esimerkiksi viestijonojen tai julkisten REST-rajapintojen kautta, joka mahdollistaa palveluiden löyhän liitoksen [Aderaldo et al. 2017].

Ennen suuri osa tietojärjestelmistä oli monoliittisia sovelluksia. **Monoliittinen järjestelmä** koostuu yhdestä laajasta koodipohjasta, joka tarjoaa kaikki sovelluksen tukemat palvelut ja ominaisuudet [Goel ja Nayak 2019]. Kuviossa 1 on esitetty monoliittisen arkkitehtuurin ja **mikropalveluarkkitehtuurin** olennaisin ero. Monoliittisessa järjestelmässä sovelluskehitys koskee aina koko koodipohjaa ja sovellusta. Jos sovelluskomponenttiin tehdään muutoksia, täytyy koko sovellus kääntää ja uudelleen käyttöönottaa, mikä tekee jatkuvasta käyttöönotosta (engl. *continuous deployment*) haastavaa. Tämä aiheuttaa sen, että organisaatioiden ohjelmistoversioiden julkistamiskierrokset pitenevät sekä iteratiivinen, että inkrementaalinen kehitys hidastuu [Goel ja Nayak 2019].



Kuvio 1. Monoliittisen arkkitehtuurin ja mikropalveluarkkitehtuurin ero.

Mikropalveluarkkitehtuuri tarjoaa myös ratkaisun itsenäiseen horisontaaliseen skaalautumiseen palvelimien infrastruktuurin tehostamisessa. Koska mikropalvelut ovat itsenäisiä osia järjestelmässä, voidaan järjestelmän kokonaiskuorma jakaa usean pienemmän palvelimen kesken. Perinteisessä **monoliittisessa** järjestelmässä skaalaaminen on hyvin vaikeaa, sillä yksittäisen palvelun kuorman kasvaessa joudutaan palvelimien skaalautuvuus hoitamaan vertikaalisesti koskemaan koko järjestelmää. Tämä voi aiheuttaa palvelimien resurssien tarpeetonta kasvattamista. [Goel ja Nayak 2019]

Hajautetussa **mikropalveluarkkitehtuurissa** yksittäinen **mikropalvelu** voi pienimmillään sisältää vain yhden funktion, operaation tai tietyn palvelun, josta se vastaa. Joissain tietojärjestelmissä mikropalvelu voi riippua toisen mikropalvelun tilasta tai olla täysin itsenäinen muista mikropalveluista. Mikropalvelu kasvaa ajan kanssa, kun uusia ominaisuuksia lisätään. Täysin itsenäisen mikropalvelun kasvaessa se voidaan helposti jakaa pienempiin palveluihin [Goel ja Nayak 2019]. Konferenssijulkaisussaan [Kookarinrat ja Temtanapat 2016] kuitenkin ilmaisevat, että mikropalveluiden hajauttamiseen ja niiden kokoon ei ole mitään selkeää käytännettä.

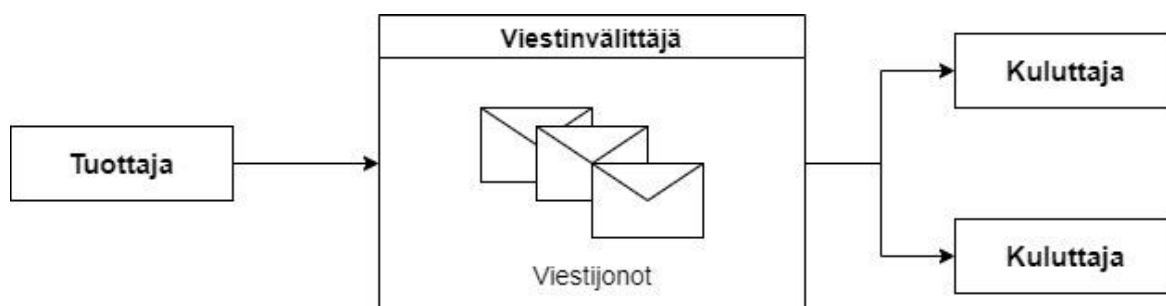
3.2 Mikropalveluiden välinen tapahtumien kuuntelu ja kommunikointi

Modernit **hajautetut** tietojärjestelmät voivat koostua sadoista tai jopa tuhansista eri komponenteista sekä sovelluksista, jotka tarjoavat toisilleen erilaisia palveluita ja funktionaalisia ominaisuuksia. Tällaisissa hajautetuissa arkkitehtuureissa on useita erilaisia haasteita liittyen komponenttien tiukkaan suhteeseen kommunikoinnissa [Magnoni 2015]. Tätä ongelmaa on yrittänyt poistaa esimerkiksi konferenssijulkaisussaan [Kookanrinrat ja Temtanapat 2016], esittelemällä hajautetun viestiväylän palveluiden väliseen kommunikointiin.

Koska mikropalveluiden palvelut sijaitsevat eri palvelimilla, voidaan **kommunikointiin** käyttää suoria HTTP-protokollaan perustuvia julkisia REST-rajapintakutsuja. Tässä tapauksessa pyynnöt ovat synkronisia etäpyyntöjä toiseen mikropalveluun, ja tämä johtaa riippuvuuksiin mikropalveluiden välillä. Toinen yleinen tapa mikropalveluiden väliseen kommunikointiin onkin kevytrakenteiset **viestijonot**.

Viestijonot voivat vapauttaa eri mikropalvelut ylimääräisistä riippuvuuksista ja löyhentää liitosta mikropalveluiden välillä [Kookarinrat ja Temtanapat 2016]. Viestijonot ovat eräänlainen puskuri tai postilaatikko lähetettäville viesteille ja ne mahdollistavat **asynkronisen** kommunikoinnin, jossa osapuolten ei tarvitse olla vuorovaikutuksessa viestijonon kanssa samanaikaisesti. Asynkronisella kommunikaatiolla eli ei-reaaliaikaisella kommunikoinnilla tarkoitetaan **kommunikointitapaa**, jossa kommunikoinnin osapuolet eivät ole ajallisesti toisistaan riippuvaisia.

Viestijärjestelmiin viitataan yleensä **viestinvälittäjinä** (engl. *message broker*) ja ne ovat vastuussa tiedonsiirrosta sovellusten sekä palveluiden välillä. Tyypillisin käyttötapaus viestinvälityksessä on, että jokin palvelu tai sovellus toimii **tuottajana** (engl. *producer*) viestille ja toinen taas sen **kuluttajana** (engl. *consumer*). Viestinvälittäjän ansiosta tuottajat ja kuluttajat voivat keskittyä jaettaviin tietoihin kuin siihen, kuinka niitä voidaan jakaa. Viestinvälittäjiä on laajasti käytetty viime vuosina kommunikoinnin ja integraatioiden toteuttamiseen hajautetuissa järjestelmissä [Magnoni 2015]. Kuvio 2 on havainnollistava kuva viestinvälittäjästä.



Kuvio 2. Viestinvälittäjä.

Viestijärjestelmät tukevat useita erilaisia **kommunikointimalleja** tuottajien ja kuluttajien väliseen tiedon jakamiseen. Viestijärjestelmien yleisimmät kommunikointimallit ovat **jonot** (engl. *queue*) tai **aiheet** (engl. *topic*). Jono mahdollistaa tiedon pitämisen viestikanavassa myöhempää lähetystä varten, jos tiedolle ei ole kuluttajaa. Aiheet taas ovat yleisiä

Publish/Subscribe -kommunikointimallissa, jossa usean kuluttajan tapauksessa, viestijärjestelmä lähettää sen kaikille haluavilla kuluttajille. Monimutkaisempia lähetystapoja tarjoaa protokollatasolla esimerkiksi **AMQP**, joka perustuu solmuihin ja viestiväyliin [Magnoni 2015]. AMQP:sta kerrotaan enemmän tutkielman luvussa 3.3. Publish/Subscribe -kommunikointimalli on yleinen **asynkroniseen** kommunikointiin tarkoitettu tapa julkaista tietoa usealle eri vastaanottajalle. Malli mahdollistaa sen, että tuottajien (engl. *producers*) ei tarvitse tietää keitä vastaanottajat eli kuluttajat (engl. *consumers*) ovat [Kookarinrat ja Temtanapat 2016]. Publish/Subscribe -kommunikointimallista kerrotaan enemmän tutkielman luvussa 4.

3.3 Advanced Message Queuing Protocol (AMQP)

Advanced Message Queuing Protocol eli **AMQP** on 2006 vuonna kehitetty avoimen standardin protokolla alun perin viestikeskeisten väliohjelmistojen kommunikointiin (Fernandes et al. 2013). AMQP on internetprotokolla, joka on tarkoitettu sovellusten tai organisaatioiden väliseen kommunikointiin. AMQP:n kehittäjien mukaan sen tunnusmaisimmat ominaisuudet ovat tietoturvallisuus, luotettavuus, yhteensopivuus, norminmukaisuus sekä avoimuus [AMQP 2021]. AMQP-protokollaan perustuva kommunikointitapa on yleinen tapa toteuttaa kommunikointi mikropalveluarkkitehtuurissa, jossa jokainen mikropalvelu vastaa tietystä sovelluslogiikan alueesta.

AMQP yksinkertaistettuna mahdollistaa sovellusten lähettää ja vastaanottaa viestejä. AMQP:n standardien mukaan viestijärjestelmät, jotka ovat kirjoitettu eri ohjelmointikielillä ja eri alustoille, voivat lähettää ja vastaanottaa viestejä toistensa kanssa. Yhteensopivuus eri sovellusten välillä on mahdollista, koska AMQP on alustariippumaton sekä se on niin sanottu alatason protokolla [AMQP 2021]. Alatason protokollalla tarkoitetaan protokollaa, joka tarjoaa tavan toteuttaa sovelluskerroksen tekniikat.

3.4 RabbitMQ-viestinvälittäjä

RabbitMQ on yksi suosituimmista avoimeen lähdekoodiin perustuvista viestinvälittäjistä (engl. *message broker*). RabbitMQ:n tekniikka perustuu jonoihin, vaihteisiin, tuottajiin

sekä kuluttajiin. Yksittäinen palvelu voi olla yhtäaikaisesti sekä **tuottaja** (engl. *producer*), että **kuluttaja** (engl. *consumer*). **Jonolla** tarkoitetaan RabbitMQ:n sisällä sijaitsevaa viestipuskuria eli postilaatikkoa viesteille. Tuottajien viestit päätyvät jonoihin, joista kuluttajat vastaanottavat niitä. Tätä on havainnollistettu Kuvio 2. **Vaihde** taas toimii viestinvälittäjässä välikappaleena tuottajan ja viestijonon välissä. Vaihteen tehtävänä on päättää, lähetetäänkö tuottajan viesti yhteen jonoon, useampaan jonoon vai hylätäänkö se kokonaan. [RabbitMQ 2021]

RabbitMQ:n palvelin on kirjoitettu Erlang-ohjelmointikielellä ja palvelimen eräitä tärkeimpiä ohjelmaosioita ovat RabbitMQ-vaihdepalvelin, yhdyskäytävä HTTP-protokollalle sekä AMQP-asiakaskirjastoja muun muuassa Javalle, .NET kehykselle ja Erlangille [Fernandes et al. 2013]. RabbitMQ mahdollistaa Publish/Subscribe -kommunikointimallin toteuttamisen, jossa viesti voidaan lähettää usealle kuluttajalle yhtäaikaisesti [RabbitMQ 2021]. RabbitMQ:n arkkitehtuuri tukee pääasiassa AMQP- ja STOMP-protokollia, mutta muitakin protokollia on mahdollista asentaa lisäosina. STOMP-protokolla eli *Simpe Text Oriented Messaging Protocol* on eräs kommunikointiprotokolla viestinvälittäjille AMQP:n ohella [Magnoni 2015].

4 Publish/Subscribe -toimintaperiaatteet

Tässä luvussa esitellään Publish/Subscribe -kommunikointimalli, sen toimintaperiaatteet sekä eri tapoja toteuttaa se.

4.1 Publish/Subscribe kommunikointimallina

Publish/Subscribe on suosittu kommunikointimalli laajojen **hajautettujen** järjestelmien toteutuksessa. Publish/Subscribe on **kommunikointimalli**, jossa **tilaajat** (engl. *subscribers*) ilmaisevat kiinnostuksena saada ilmoituksia tietyistä **tapahtumista** (engl. *events*) [Setty 2015, s. 1]. Publish/Subscribe -malli on käytössä suuren kokoluokan tietojärjestelmissä laajalti. Näitä ovat esimerkiksi finanssialan datan levitys, RSS-syötteen hajautus ja tunnettu Spotify-musiikkisovellus [Setty et al. 2013]. Publish/Subscribe -järjestelmissä **tilaukset** yhdistetään niitä vastaaviin **tapahtumiin** eli julkaisuihin, joita **julkaisijat** (engl. *publishers*) julkaisevat. Tilaaja esittää kiinnostuksensa tiettyihin tapahtumiin ja jos julkaistu tapahtuma on yhteensopiva tilauksen kanssa, ilmoitetaan siitä tilaajalle [Ashayer, Leung ja Jacobsen 2002].

Publish/Subscribe mahdollistaa minkä tahansa määrän julkaisijoita kommunikoida minkä tahansa määrän tilaajia kanssa, **asynkronisesti** ja tuntemattomasti. Tuntemattomuudella tarkoitetaan, että julkaisijoiden ei tarvitse tietää, ketkä kyseisistä julkaisuista on kiinnostuneita. **Kommunikointimallin** tärkein vaatimus on tapahtumien ja viestien välitys luotettavasti julkaisijalta kaikille kiinnostuneille tilaajille [Turau ja Siegemund 2017]. Konferenssijulkaisussa [Ashayer, Leung ja Jacobsen 2002] esittävät, että Publish/Subscribe -järjestelmien **yhteensopivuusalgoritmien** eli tapahtumien ja tilausten yhdistämisen kaksi tärkeintä vaihetta ovat tilauspredikaattien tarkistus sekä niiden perusteella tilausten lähetys tilaajille.

Publish/Subscribe -järjestelmissä julkaisijoiden **tapahtumien** tuottaminen voi yhtäaikaisesti jatkua itsenäisesti sekä riippumattomasti samalla kun tilaajat vastaanottavat ja käsittelevät niitä. Tämä löyhä liitos asiakasohjelmistojen välillä vähentää riippuvuuksia niiden väliltä ja näin ollen soveltuu hyvin suurien tietojärjestelmien kommunikointimalliksi [Peter

R. Pietzuch 2004, s. 36]. Kaksi tunnetuinta Publish/Subscribe -skeemaa eli rakennetta ovat **aiheperusteinen** sekä **sisältöperusteinen** rakenne [Setty 2015, s. 19]. Publish/Subscribe -järjestelmät ovat perinteisesti olleet rakenteeltaan aiheperusteisia (engl. *topic-based*), missä tilaaja voi osoittaa kiinnostuksensa ainoastaan joukkoon ennaltamääriteltyjä aiheita. Toisaalta sisältöperusteiset (engl. *content-based*) järjestelmät, missä tilaajat osoittavat kiinnostuksensa perustuen ominaisuuksien arvoihin ovat kasvattaneet suosiotaan viime vuosina [Yang, Fan ja Jiang 2016].

4.2 Aiheperusteinen Publish/Subscribe

Aiheperusteinen eli *Topic-based* **Publish/Subscribe** on toinen käytetyimmistä skeemoista toteuttaa Publish/Subscribe -kommunikointimalli. Aiheperusteisessa järjestelmässä tilaukset muodostetaan ennaltamääritellyillä **aiheilla** (engl. *topic*). Jokainen **tapahtuma** (engl. *event*) sisältään tiedon mihin ennalta määriteltyyn aiheeseen se liittyy. Tapahtumien julkaisijat ovat näin ollen vastuussa, että julkaistut tapahtumat vastaavat määriteltyjä aiheita [Setty 2015, s. 2].

Artikkelissaan [Setty et al. 2013] esittävät aiheperustaisen tilauksen merkkijonoparina, joka koostuu tilaajan käyttäjätunnuksesta sekä aiheen nimestä. Heidän tutkimansa Spotify-musiikkisovelluksen tapauksessa käyttäjä voi esimerkiksi tehdä tilauksen tiettyyn soittolistaan (engl. *playlist*) eli kokoelmaan kappaleita, missä soittolistan nimi toimii uniikkina aiheen tunnisteena. Kyseinen tilaus olisi muotoa `username.playlistname`. Käytännössä tämä tarkoittaisi, että kun kokoelmaan lisätään uusi kappale, ilmoitetaan tästä lisäyksestä kaikille tilauksen tehneille käyttäjille.

Tilaajat saavat siis tiedon **tapahtumista** kaikista niistä aiheista, joihin he ovat tilauksensa tehneet. Jos useampi tilaaja on tehnyt tilauksensa samaan **aiheeseen**, saavat he kaikki tiedon samasta aiheen tapahtumasta [Setty 2015, s. 20]. **Aiheperustaisessa** mallissa jokainen aihe pitää mukanaan uniikin tunnisteen. Tilaajat ilmaisevat kiinnostuksena tiettyyn aiheeseen tekemällä tilauksensa kyseisellä uniikilla tunnisteella [Turau ja Siegemund 2017]. Tavalliset aiheperusteiset Publish/Subscribe -arkkitehtuurit sisältävät välittäjäpalvelimen

aiheiden hallinnoimiseen. Välittäjä kerää kaikki julkaistut tapahtumat tai viestit ja välittää ne niistä kiinnostuneille tilaajille [Banno et al. 2014].

4.3 Sisältöperusteinen Publish/Subscribe

Toinen suosittu skeema toteuttaa Publish/Subscribe -kommunikointimalli on **sisältöperusteinen** (engl. *content-based*) Publish/Subscribe. Sisältöperustaisessa järjestelmässä tilaukset muodostetaan **totuusarvolausekkeilla** perustuen ominaisuuksiin ja arvoihin, esimerkiksi (`”Osake = ’Apple’, arvo > 95 ja arvo < 98”`) [Setty 2015, s. 22]. Aiheperusteisiin järjestelmiin verrattuna sisältöpohjaiset järjestelmät tarjoavat hienojakoisempia tilauksia, mutta aiheuttaa samalla ylimääräistä monimutkaisuutta tilauksien tarkistuksessa [Yang, Fan ja Jiang 2016]. Sisältöperusteinen kommunikointimalli on joustava monenvälinen malli (engl. *many-to-many*), joka soveltuu useisiin moderneihin hajautettuihin järjestelmiin, esimerkiksi tiedon **suodatuksen** ja sijaintiin liittyvien palveluiden kautta [Qian et al. 2014].

Konferenssijulkaisussaan [Fabret et al. 2001] kuvaavat **sisältöperustaista** tilausta parina tapahtuman kanssa. He esittävät, että yksittäinen tilaus voi olla esimerkiksi predikaatti, joka muodostuu ominaisuudesta, sen arvosta ja vertailuoperaattorista. **Julkaisijan** tai järjestelmän generoima **tapahtuma** taas pitää sisällään tiedon ominaisuudesta sekä sen arvosta, joiden perusteella yhteensopivuus voidaan laskea. Esimerkiksi tapahtuma `(hinta, $8)` sopii yhteen tilauspredikaatin `(hinta, $10, <=)` kanssa. Molemmat sisältävät saman ominaisuuden ja tilauspredikaatissa olevan vertailuoperaattorin perusteella kahdeksan dollarin arvo tapahtumassa on yhteensopiva tilauksen määrittelemän arvon kanssa.

4.4 Suodattimet Publish/Subscribe -järjestelmässä

Suodattimet (engl. *filters*) ovat suosittu lähestymistapa vähentää lähetettävän tiedon määrää **tilauspohjaisissa** järjestelmissä. Suodattimet ovat eräs **sisältöperustaisen tilauksen** toteutus. Suodattimet mukautuvat käyttäjän tarpeisiin sekä tarjoavat dynaamisemman tavan tehdä tilauksia. Yleensä **tilaaja** lähettää omiin tarpeisiinsa kustomoidun suodattimen järjestelmän **tilauspalveluun** ja tilauspalvelun vastuulla on suodattaa vain haluttu tieto takaisin

tilaajalle. Tilauspalvelun vastaanottaessa **tapahtuman** joltain julkaisijalta, käy se läpi kaikki tilaajat, jotka ovat kiinnostuneita tästä tapahtumasta. Päivitysten ja tilaajalistojen muuttuessa jatkuvasti täytyy myös palvelun mukautua dynaamisesti [Xie, Navathe ja Prasad 2005].

Tiedon suodatus on erityisen tärkeää, kun **tilaaja** käyttää tilauspohjaista järjestelmää mobiililaitteella, koska se auttaa vähentämään liikennettä järjestelmän ja laitteen välillä [Xie, Navathe ja Prasad 2005]. Käyttäjän manuaalisesti lisäämien **tilauksien** lisäksi **suodattimet** aktivoivat järjestelmässä automaattisia tilauksia perustuen käyttäjän kiinnostuksiin [Leontiadis 2007]. Tiedon automaattinen suodatus sekä lähetys mahdollistavat tiedon lähetyksen ilman, että käyttäjän tarvitsee erikseen sitä pyytää. Tämä on niin sanottu *push*-malli eli palvelimen aktivoima tiedon lähetys [Yu et al. 2014].

Tilaaja voi esimerkiksi osoittaa kiinnostuksena tiettyyn **aiheeseen** suodattimen perusteella. Kun järjestelmään julkaistaan suodattimeen osuva aihe, luo järjestelmä aiheesta automaattisesti **tilauksen** ja julkaisee **tapahtuman** tilaajalle [Zaarour ja Curry 2019]. Tutkimuksessaan [Yu et al. 2014] kuvaavat sijaintiin liittyvää sovellusta, missä esimerkiksi mainostajat voivat julkaista suodattimeen perustuvan mainoksen ”`meriruokaa, Manhattan`” tietojärjestelmään. Järjestelmä automaattisesti työntää (engl. *push*) eli lähettää tiedon mainoksesta mobiilikäyttäjille perustuen mobiilikäyttäjien sijainteihin ja verkkoselaamisen sisältöön. Mobiilikäyttäjät, jotka ovat kyseisessä sijainnissa ja ovat etsineet meriruokaa verkosta ovat tässä tapauksessa osoittaneet kiinnostuksensa mainostajan luomaan suodattimeen.

5 Aiempi tutkimus

Tässä luvussa esitellään aiempaa tutkimustietoa liittyen Publish/Subscribe -järjestelmiin sekä niihin liittyvästä tapahtumien ja tilausten yhdistämisestä eli yhteensopivuuden tarkistamisesta.

5.1 Publish/Subscribe -järjestelmät

Luvussa 4 esitelty Publish/Subscribe -**kommunikointimalli** on kommunikointimekanismina kehittynyt nopeasti [Lian et al. 2020]. Väitöskirjassaan [Setty 2015, s. 12] esittää, että vaikka Publish/Subscribe -järjestelmien käsittelemä tiedon määrä ja sen analysoiminen, sekä ymmärtäminen ovat kriittisiä tietoja Publish/Subscribe -järjestelmän suunnittelussa, niin aihe on silti todella vähän tutkittu. Hajautettuja tietojärjestelmiä sekä tapahtumaperustaisia väliohjelmistoja on kuitenkin tutkittu jo 2000-luvun alusta asti [P. R. Pietzuch ja Bacon 2002].

Yksi suuri haaste tapahtumapohjaisissa järjestelmissä on **suorituskyvyn** optimointi. Järjestelmän tulisi pystyä palvelemaan useita yhtäaikaisia käyttäjiä ja välittää **tapahtumia** sekä tietoa käyttäjille millisekunneissa [Yu et al. 2014]. Konferenssijulkaisussaan [Setty et al. 2013] toteavatkin, että aina kun tapahtumien tuottamiseen perustuvaa järjestelmää suunnitellaan, täytyy arkkitehtien tehdä perustavanlaatuinen valinta **viiveen** (engl. *latency*) ja **toimintavarmuuden** (engl. *reliability*) välillä. Viive mittaa aikaa, joka kuluu tapahtumien saapumiseen määränpäähänsä verkon yli hajautetussa järjestelmässä. Toimintavarmuus taas kuvaa järjestelmän kykyä toimia ilman vikaa sekä vaaditulla tavalla.

Tapahtumien välittämistä Publish/Subscribe -järjestelmissä on tutkittu eri lähtökohdista lähdekirjallisuudessa. **Aiheperusteiset** ja **sisältöperusteiset** rakenteet tarjoavat erilaiset tavat välittää **tapahtumia** järjestelmässä. Sisältöperusteisissa järjestelmissä tapahtumien välittäminen perustuu predikaatteihin sekä totuusarvomuuttujien vertailuun, ja tarjoaa hienojakoisempia tapoja välittää tapahtumia.

Aiheperusteiset järjestelmät taas keskittyvät enemmänkin **tapahtumien** välittämiseen tietyn loogisen ennalta määrätyn aiheen mukaan. Tämä tarkoittaa sitä, että kaikki tietyn

aiheen tilaajat saavan kyseisen aiheen kaikki tapahtumat. Aiheperusteisten järjestelmien tutkiminen on keskittynyt paljon **vertaisverkkoihin** (engl. *peer-to-peer*) tapahtumien välittämisessä. Vertaisverkossa jokainen verkkoon kytketty **tilaaja** toimii yhtenä verkon solmuna (engl. *node*), toimien näin sekä tilaajana että **julkaisijana**. Nämä verkot yleensä takaavat tapahtuman välittämisen verkon solmulle keskiarvoltaan alle `log n` askeleella [Turau ja Siegemund 2017]. Väitöskirjassaan [Setty 2015, s. 17] esittää, että vertaisverkot ovat halpa tapa vähentää aiheperusteisen Publish/Subscribe -järjestelmän kuormaa, hajauttamalla osa kuormasta vertaisverkon solmuille. Aiheperusteisessa järjestelmässä vertaisverkon solmuina toimivat järjestelmän tilaajat.

5.2 Tapahtumien ja tilausten yhdistäminen

Tapahtumien ja tilausten yhdistäminen eli yhteensopivuuden tarkistaminen (engl. *event matching*) on prosessi, jossa tarkistetaan suuri määrä tapahtumia suurta määrää tilauksia vasten. Se on keskeinen ongelma suuren kokoluokan hajautettujen Publish/Subscribe -järjestelmien kokonaissuorituskyvyssä [Qian et al. 2014]. Konferenssijulkaisussaan [Yang, Fan ja Jiang 2016] toteavatkin, että suurin osa nykyisistä toteutuksista kärsii huomattavasta **suorituskyvyn** alenemisesta, kun järjestelmässä on suuri määrä tilauksia. Tämä sama toteamus koskee myös tämän tutkielman aiheena olevaa tilauspalvelu-sovelluskomponenttia.

Konferenssijulkaisussa [Setty et.al 2013] esittelevät Spotifyn käyttämän **aiheperusteisen** (engl. *topic-based*) Publish/Subscribe -välittäjän, joka toimii niin sanottuna hajautettuna tiivistettynä tauluna, missä yksittäinen **tilaus** toimii avaimena. Välittäjän vastuulle kuuluu esimerkiksi uusien tilauksien tallennus, **julkaisujen** tarkistus muistissa olevia tilauksia vasten sekä julkaisujen edelleen lähettäminen eteenpäin. Välittäjän vastuulla on myös poistaa tilauksia tietystä **aiheesta** sekä mahdollistaa kuormanjako palvelimien välillä.

Konferenssijulkaisussaan [Lian et.al 2020] taas keskittyvät **tilausten luokittelun** suodatusalgoritmiin, joka vähentää etsittävien tilauksien määrää ennen varsinaista **tapahtumien** tarkistusta tilauksia vasten. Heidän esittelemä **algoritminsa** perustuu siihen, että tilaukset luokitellaan ja luokittelun avulla voidaan poistaa tai suodattaa ne tilaukset, jotka eivät suoraan liity saapuneeseen tapahtumaan. Tämä vähentää varsinaisen tapahtumien tarkistuksen

kuormaa ja näin parantaa järjestelmän **suorituskykyä**. Julkaisun kokeellisessa osuudessa on huomattava ero vastaaviin suodatusalgoritmeihin tilausten vähenemisessä ennen tapahtumien tarkistusta. Oleellisin huomio on, että tilausmäärien kasvaessa, kasvaa myös algoritmin suodatustehokkuus.

Tutkimuksessaan [Xie, Navathe ja Prasad 2005], esittelevät **suodattimien** tarkistukseen perustuvat indeksointiskeemat eli **rakenteet**. He esittelevät neljä erilaista tapaa **indeksoida** suodattimia tehokkaampaa tarkistusta varten. Indeksoinnilla tarkoitetaan tietorakenteiden alkioiden tarkastelua. Näistä *Group-Sort Indexing Scheme* (**GSIS**) on kokeellisen osuuden perusteella tehokkain indeksointitapa suodattimien hakua ja tarkistusta varten. GSIS perusajatuksena on vähentää samojen suodattimien useaa tarkistuskertaa. Jos useammalla **tilaajalla** on sama suodatin, voidaan ne luokitella kuuluvaksi samaan ryhmään, joten suodattimen ja **tapahtuman** yhteensopivuuden tarkistamisessa ei tarvitse käydä jokaisen tilaajan suodatinta erikseen läpi. **Algoritmin** lopputuloksena tietorakenteen jokaisen indeksin arvona on lista saman suodattimen omaavia tilaajia järjestyksessä suodattimen arvon mukaan.

Tutkimuksissaan [Leontiadis 2007] sekä [Leontiadis, Costa ja Mascolo 2009] ovat kehittäneet Publish/Subscribe -**kommunikointimalliin** perustuvat ratkaisut ajoneuvojen tarpeisiin, missä ajoneuvon sijainti on keskeisessä roolissa. He perustavat ratkaisunsa niin sanottuihin kiinnostuksen kohteena oleviin pisteisiin (engl. *Point of Interest*). Järjestelmän **tilaajina** toimivat yksittäiset ajoneuvot ja kuten useissa Publish/Subscribe -järjestelmissä, tilaajat voivat osoittaa kiinnostuksensa joko suoraan tiettyyn **aiheeseen** (engl. *topic*) tai esimerkiksi oman sijaintinsa perusteella perustuen **suodattimiin** (engl. *filters*).

6 Suorituskykytestaus

Tässä luvussa esitellään suorituskykytestauksen tyyppejä, suorituskykytestaus yleisesti, sekä tutkielmassa hyödynnettävä Gatling-suorituskykytestauskehys.

6.1 Suorituskykytestauksen tyyppejä

Suorituskykytestaus koostuu erilaisista testaustyypeistä, jotka määräytyvät testattavien järjestelmien suorituskykyvaatimusten mukaan. Alla on esitelty joitain blogikirjoituksessa [Jones 2015] esittelemiä suorituskykytestaustyyppejä:

- **Rasitustestaus** (engl. *load testing*): Tarkastelee järjestelmän suorituskykyä ennalta määritellyn kuormituksen mukaan. Tavoitteena on yleensä tarkistaa täyttääkö järjestelmä sille asetetut suorituskykyvaatimukset.
- **Huippurasitustestaus** (engl. *peak load testing*): Tarkastelee järjestelmän tai yksittäisen sovelluskomponentin suorituskykyä ennalta määritellyn kuormituksen huipun mukaan. Kuten rasitustestaus, huippurasitustestauksen päämääränä on tarkistaa täyttääkö järjestelmä sille asetetut suorituskykyvaatimukset, mutta odotettujen kuormituspiikkien mukaan.
- **Stressitesti** (engl. *stress test*): Tarkastelee järjestelmän suorituskyvyn mittaamista määritettyjen suorituskykykriteerien perusteella, mutta ylittää suunnitellun kuormitusrajan. Eräs tapa saavuttaa tämä on lisätä kuormaa asteittain, hallitusti, kunnes järjestelmän heikoin lenkki löytyy.
- **Liotustesti / Kestävyystesti** (engl. *soak test*): Tarkastelee järjestelmän pitkäaikaista käyttöä hallitulla kuormituksella. Tavoitteena on seurata järjestelmän käyttäytymistä pitkällä aikavälillä. Tähän kuuluu esimerkiksi resurssien kulutus ja vapautus, jotka voidaan havaita vain ajan myötä. Hyvänä esimerkkinä on asteittainen muistivuoto.

6.2 Suorituskykytestaus yleisesti

Suorituskykytestaus on tietojärjestelmän kapasiteetin analysointia, jossa mittaustietoja käytetään ennustamaan milloin kuormitustasot käyttävät järjestelmän resurssit loppuun [Wu ja Wang 2010]. Se on prosessi, jossa järjestelmää esimerkiksi kuormitetaan emuloimalla oikeita käyttäjiä järjestelmän **pullonkaulojen** löytämiseksi. Suorituskykytestausta kutsutaan myös usein kuormitustestaukseksi [Sarojadevi 2011].

Järjestelmän **pullonkaulojen** identifioinnin lisäksi **suorituskykytestaus** auttaa määrittämään tarvittavan ajan järjestelmän tehtävien suorittamiseen sekä antaa kuvan kuinka vakaa järjestelmä on kuormituksen alla [Netto et al. 2011]. Järjestelmän resurssien näkökulmasta suorituskykytestauksessa nousee esiin esimerkiksi suorittimien käyttö, muistin käyttö sekä verkon kaistanleveyden (engl. *network bandwith*) käyttö [Sarojadevi 2011]. **Vasteaika** on myös usein käytetty mittari suorituskykytestauksessa. Vasteaika kuvaa tyypillisesti viivettä pyynnön (engl. *request*) ja toimenpiteen valmistumisen välillä [Wu ja Wang 2010].

6.3 Gatling-suorituskykytestauskehys

Gatling on avoimeen lähdekoodiin perustuva **suorituskykytestauskehys**. Gatling on suunniteltu jatkuvaan suorituskykytestaukseen, joka elää tietojärjestelmän kehityksen mukana niin kehitysympäristöissä kuin tuotannossakin. Gatling-suorituskykytestauksen ideana on parantaa järjestelmän suorituskykyä jatkuvasti kehityksen aikana. Gatling-testauksen muodostama kattava raportti auttaa kehittäjiä löytämään suorituskyvyn ongelmakohdat ja pullonkaulat helposti. [Gatling 2021]

Gatling-pohjaiset suorituskykytestit ovat kehitetty simuloimaan oikeita järjestelmän loppukäyttäjiä. Testikäyttäjien tekemät HTTP-pyynnöt määräytyvät rakennetun simulaation mukaan. Gatling tarjoaa useita erilaisia **skenaarioita** syöttää käyttäjiä järjestelmään. Gatlingin skenaariolla tarkoitetaan luvussa 6.1 esiteltyä testaustyyppiä. Näitä ovat lähteen [Gatling 2021] mukaan esimerkiksi seuraavat skenaariot:

- Kapasiteettitestissä syötetään järjestelmään käyttäjiä pikkuhiljaa koko ajan enemmän, kunnes järjestelmä kaatuu.

- Stressitestillä simuloidaan useiden käyttäjien yhtäaikaista kirjautumista tai tiettyä käyttötapausta.
- Liotustestillä simuloidaan järjestelmän pitkäaikaista käyttöä useiden tuntien tai päivien ajan.

7 Tietojärjestelmän nykyinen toteutusratkaisu

Tässä luvussa esitellään tietojärjestelmän sekä tilauspalvelu-sovelluskomponentin nykyisen toteutuksen yleiskuva.

7.1 Tilaajat ja tilaukset

Tutkielman kohteena olevan tietojärjestelmän loppukäyttäjä voi ilmaista kiinnostuksensa tiettyyn **aiheeseen** (engl. *topic*) tai siihen liittyviin aliaiheisiin tekemällä **tilauksen** (engl. *subscription*) kyseisestä aiheesta. Järjestelmän tilauspalvelu-sovelluskomponentti tallentaa tiedon uudesta tilauksesta käyttäjän aktiivisen **istunnon** yhteyteen. Aktiivisella istunnolla tarkoitetaan käyttäjän kirjautumista asiakasohjelmistoon, joka voi olla esimerkiksi internet-selain tai Windows-sovellus.

Luvussa 4.1 esiteltyyn Publish/Subscribe -kommunikointimalliin perustuen järjestelmän **tilaajina** toimivat käyttäjät ja **tapahtumien julkaisijoina** järjestelmän mikropalvelut. Publish/Subscribe -malliin perustuen tietojärjestelmän **tilaukset** noudattavat luvussa 4.2 esiteltyä **aiheperustaista** kommunikointimallia. Pääentiteetti vastaa tiettyä **aihetta** ja aiheen yksilöivänä tunnuksena toimii kyseisen entiteetin tunniste. Aiheella voi olla useita eri aliaiheita, jotka tilauspalvelu käsittelee saman tilauksen puitteissa.

Tilaukset voivat olla joko luvussa 4.2 esiteltyjä **aiheperusteisia tilauksia** kyseiseen aiheeseen tai koostua erilaisista luvussa 4.4 esitellyistä **suodattimista**, joilla käyttäjä voi osoittaa kiinnostuksensa. Aiheperusteinen tilaus tarkoittaa yksityiskohtaista pyyntöä osoittaa kiinnostus tiettyyn aiheeseen uniikin tunnisteen avulla. Suodatin taas voi perustua esimerkiksi loppukäyttäjän sijaintiin. Esimerkiksi sijaintisuodattimen perusteella järjestelmä lähettää kaikki **tapahtumat**, jotka käsittelevät maantieteellisesti kyseisen suodattimen etäisyysrajauksen sisällä olevaa **aihetta**. Käyttäjä voi lisätä, muuttaa tai poistaa käytöstä järjestelmän tarjoamia suodattimia. Kun käyttäjä tekee aiheperusteisen tilauksen johonkin tietojärjestelmän **aiheeseen**, niin **tilauspalvelu** lähettää automaattisesti tapahtumat myös kaikista siihen liittyvistä aliaiheista sekä niiden muutoksista.

Istuntoon liittyy tieto kaikista **tilauksista**, joista käyttäjä on kiinnostunut, sen omissa tietorakenteissaan. Taulukko 1 kuvaa istunnon sisältämät tämän tutkielman kannalta tärkeimmät tietorakenteet. Istunto sisältää käyttäjän käyttäjätunnuksen `username`, istunnon yksilöivän tunnisteen `sessionId`, listan istunnon tilauksista `subscriptions` sekä listan istunnon sisältämistä suodattimista `filters`.

Session
String userName; String sessionId; List<Subscription> subscriptions; List<Filter> filters;

Taulukko 1. Istunnon sisältämän tietorakenteet.

Tilausten luonnin lisäksi käyttäjä voi lopettaa tilauksen tai päivittää olemassa olevaa tilausta. Taulukko 2 kuvaa aiheperusteisen tilauksen pelkistetyn rakenteen eri HTTP-metodeissa järjestelmän tilauspalvelukomponentin julkisessa REST-rajapinnassa. **Tilauspalvelu** käsittelee huomattavan määrän tietoa jokaiselta käyttäjältä järjestelmän ollessa aktiivisessa käytössä.

HTTP-metodi	URI	REQUEST	RESPONSE
POST	/api/subscriptions	entity.id	sub_id
PUT	/api/subscriptions/sub_id	entity.id	sub_id
DELETE	/api/subscriptions/sub_id	entity.id	sub_id

Taulukko 2. Tilaus (engl. *subscription*).

7.2 Tietojärjestelmän tapahtumat

Tilausten hallinnan lisäksi tilauspalvelukomponentti vastaa myös mikropalveluissa tapahtuvien **tapahtumien** välittämisestä käyttäjälle. Käyttäjien toiminnan seurauksena **mikropalveluissa** syntyy tapahtumia ja **tilauspalvelun** vastuulla on välittää kyseinen tapahtuma käyttäjille. Tapahtuma voi pitää sisällään joko pelkkää dataa tai tietoa jostain käyttäjää kiinnostavasta asiasta.

Tapahtuma koostuu mahdollisen datan lisäksi yksilöivästä tunnisteesta (engl. *identifier*), identifioivasta tietoalkion tyypistä sekä tapahtuman tyypistä. Esimerkiksi, jos kaksi käyttäjää ovat molemmat kiinnostuneet samasta aiheesta, voidaan toisen käyttäjän tehdessä muokkauksia kyseiseen aiheeseen, välittää se päivitetty tieto tapahtumana myös toiselle käyttäjälle.

Tapahtuma välitetään käyttäjän asiakasohjelmistoon JSON-muodossa. Asiakasohjelmisto voi olla esimerkiksi internet-selain tai Windows-sovellus. Alla on havainnollistettu esimerkki JSON-muotoisesta tapahtumasta. Esimerkissä on kyse tietoalkion `location` luontitapahtumasta, joka pitää sisällään `Test_Location` nimisen sijainnin.

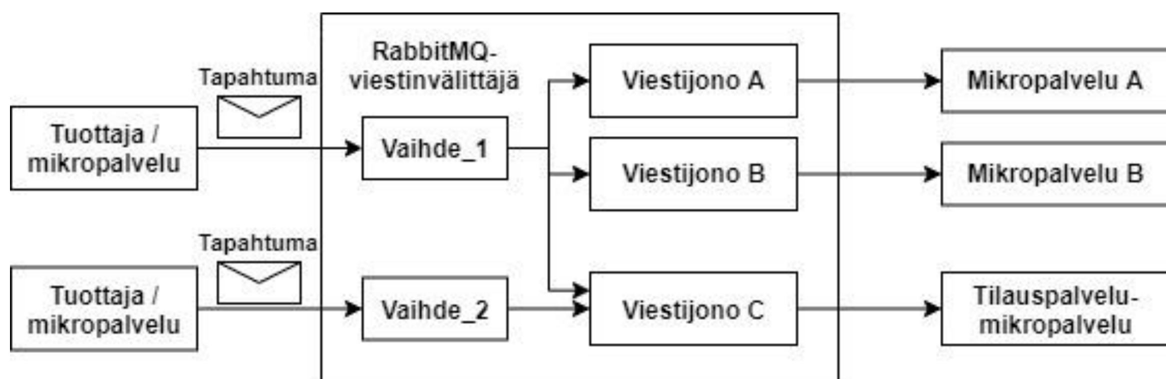

```
{
    "payload" : {
        "id" : "12312",
        "version" : "1",
        "locationX" : "60.12",
        "locationY" : "80.11",
        "displayName" : "Test_Location",
        ”author”  : ”Aleksi Tani”
    },
```

```
    "entityId" : "12312",

    "entityType" : "location",

    "methodType" : "create"

}
```

7.3 Tietojärjestelmän mikropalvelut

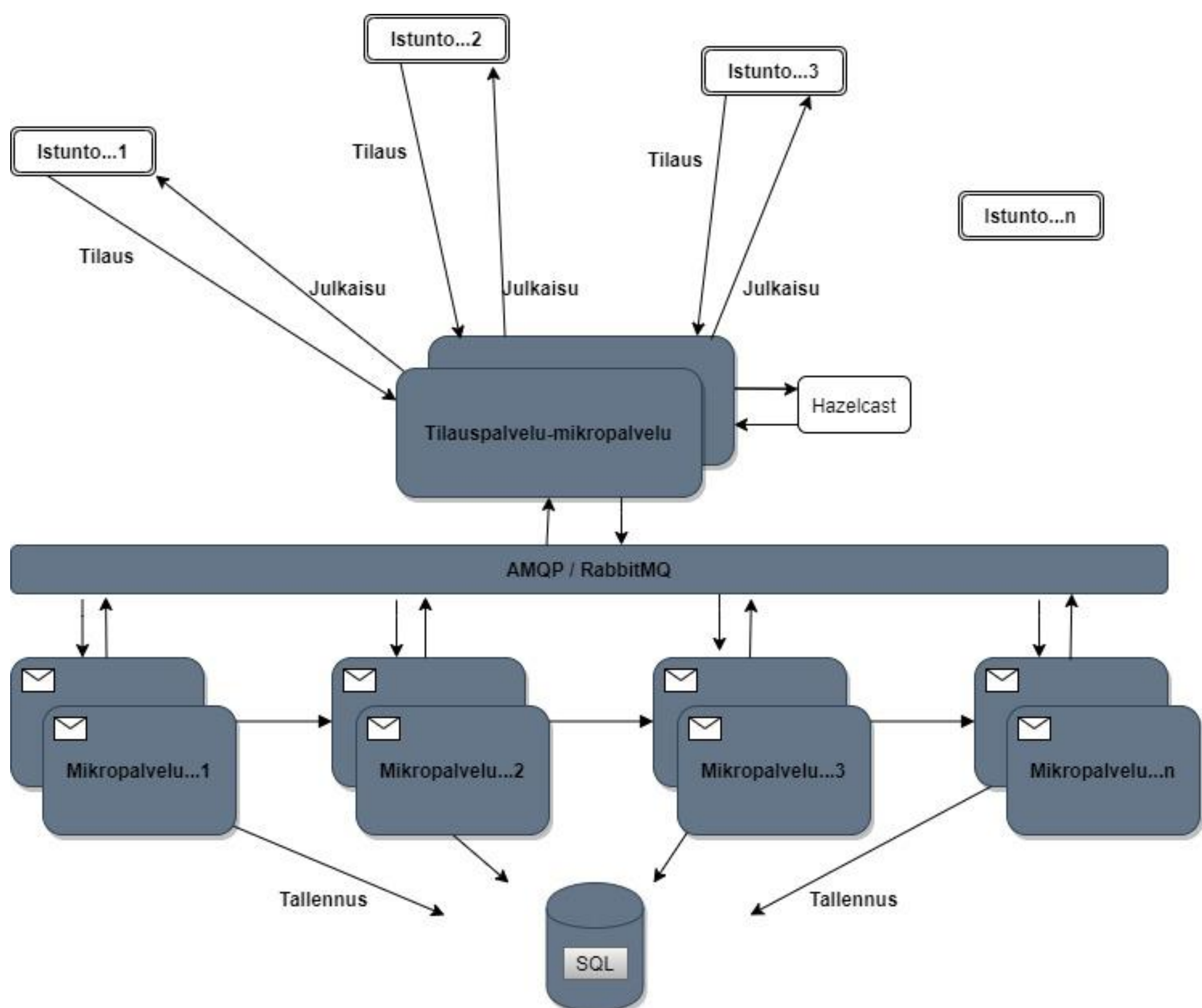
Tietojärjestelmän **mikropalveluiden** välille on kehitetty toimintalogiikkaa, jossa yksittäisen mikropalvelun tuottama **tapahtuma** voi laukaista tietyn prosessin muissa järjestelmän mikropalveluissa. Mikropalveluiden välinen tapahtumien kuuntelu on toteutettu luvussa 3.3 esiteltyyn **AMQP**-protokollaan perustuvalla **viestijonoarkkitehtuurilla**. Mikropalvelut kuuntelevat niiden mikropalveluiden tapahtumia, joista ovat kiinnostuneita.

Myös **mikropalveluiden** välinen kommunikointi perustuu Publish/Subscribe -malliin. Mikropalvelut voivat kuunnella haluamiaan muita mikropalveluita luvussa 3.4 esiteltyjen ennaltamääriteltyjen **vaihteiden** (engl. *exhange*) avulla. Yksittäisen mikropalvelun tuottaessa **tapahtuman**, se välitetään vaihteeseen, josta siitä kiinnostuneet mikropalvelut voivat viestijonojen avulla vastaanottaa sen. Tämän viestinvälityksen hoitaa **AMQP**-protokollaan perustuva **RabbitMQ**-viestinvälittäjä (engl. *message broker*), joka on esitelty luvussa 3.4. **Tilauspalvelu** kuuntelee kaikkia järjestelmän vaihteita. Tätä prosessia on havainnollistettu kuviossa 3.



Kuvio 3. Mikropalveluiden välinen tapahtumien kuuntelu.

Jokainen asynkronisesti välitetty **tapahtuma** kulkee tilauspalvelun kautta istunnoille. Publish/Subscribe -kommunikointimalliin sekä mikropalveluiden löyhään liitokseen perustuen, tapahtuman **julkaisijan** ei tarvitse tietää ketkä siitä ovat kiinnostuneita. Tämän vuoksi **tilauspalvelu**-sovelluskomponentti on ainoa mikropalvelu, jolla on tieto loppukäyttäjien asiakasohjelmistojen **istunnoista**, joille kyseinen tapahtuma täytyy välittää. Kuvio 4 esittää pelkistetyn version tutkimuksen aiheena olevan sovelluskomponentin sijoittumisesta tietojärjestelmän kokonaisarkkitehtuurissa.



Kuvio 4. Pelkistetty yleiskuva järjestelmän korkean tason arkkitehtuurista.

7.4 Tilauspalvelun toteutusratkaisu

Tilauspalvelu etsii kaikista järjestelmän käyttäjien **istunnoista** ne, jotka ovat kiinnostuneita välitettävistä tapahtumista. Tämä prosessi toistetaan jokaisen istunnoille lähetettävän **tapahtuman** kanssa. Jos tietojärjestelmällä on 100 **aktiivista istuntoa**, niin tilauspalvelu käy jokaisen **mikropalveluista** tulleen tapahtuman kohdalla 100 istunnon jokaisen tilauksen ja suodattimen läpi, löytääkseen kaikki kiinnostuneet.

Koska järjestelmä ylläpitää tilaus- sekä suodatintietoja käyttäjien **istunnoissa**, joudutaan jokainen aktiivinen istunto käymään yksitellen läpi. Ylimääräisiä iteraatiokustannuksia läpikäynnin aikana korostaa se, että vaikka ainoat kiinnostuneet istunnot olisivat löydetty jo kymmenen ensimmäisen iteraation aikana, käydään loput 90 istuntoa kuitenkin läpi. Tästä prosessista muodostuu helposti **pullonkaula** järjestelmän **suorituskyvyn** kannalta, varsinkin kun käsiteltäviä tapahtumia, **aiheperusteisia** tilauksia ja **suodattimia** on useita tuhansia. Alla oleva koodiesimerkki kuvaa yleisellä tasolla nykyisen tapahtumien ja tilauksien yhdistämisalgoritmin.

```
void consume(event) {
   forEach (session in Sessions) {
      processEvent(session, event);
   }
} // end of consume


void processEvent(session, event) {
   forEach (topicSubscription in session.topicSubscriptions) {
      if (topicSubscription.id == event.topic) {
         sendEvent(session, event);
         return;
      }
   }
   forEach (filter in session.filters) {
      if (filter.accept(event)) {
         sendEvent(session, event);
         return;
      }
   }
} // end of processEvent
```

Tietojärjestelmän ollessa käynnissä voidaan useampia tilauspalvelun **instansseja** käynnistää yhtäaikaisesti. Instanssilla tarkoitetaan tilauspalvelun tiettyä ilmentymää. Jokainen tilauspalvelu toimii itsenäisenä **palvelusolmuna** mahdollistaen tilauspalvelun kuormanjaon sekä korkean saavutettavuuden. Korkea saavutettavuus (engl. *high availability*) mahdollistaa järjestelmän normaalin käytön, jos johonkin tilauspalvelun ilmentymään tulee häiriö.

Tietojärjestelmän nykyistä toteutusta on optimoitu esimerkiksi käyttämällä hajautettua muistinvaraista tietovarastoa. Jokainen järjestelmän käyttäjä sekä käyttäjän istunnot tallennetaan **Hazelcast-muistitietoverkkoon**, joka on käytössä koko tietojärjestelmän yhteisenä instanssina. Tämä mahdollistaa sen, että jos useampi tilauspalvelun ilmentymä on yhtäaikaisesti käynnissä, niin jokainen **palvelusolmu** pääsee käsiksi samaan tietorakenteeseen.

Suuri kuorma järjestelmässä tulee kuitenkin istuntokohtaisten **suodattimien** käsittelyssä. Loppukäyttäjä voi tehdä useita monimutkaisia suodattimia, joiden perusteella **tilauspalvelu** tekee päätöksen välittää tietty **tapahtuma** eteenpäin. Luvussa 4.4 esitelty suodatin koostuu niin sanotuista avain-arvo-pareista, jotka tallennetaan aktiivisen **istunnon** tietorakenteisiin. **Tilauspalvelun** vastaanottaessa tapahtuman, käy se aiheperusteisten tilausten lisäksi kaikkien istuntojen suodattimet läpi, jos istunnoilta ei löydy tilausta kyseisen tapahtuman aiheeseen.

Näiden kahden eri tilaustyypin jako **tilauspalvelussa** on kriittinen osa tapahtumien tarkastusta. Tapahtumat voidaan **aiheen** mukaan jakaa kahteen eri ryhmään:

- ainoastaan aiheperusteisten tilausten kautta välitettävät,
- sekä aiheperusteisten, että suodattimien kautta välitettävät.

Tietojärjestelmän kaikkien **aliaiheiden tapahtumat** ovat lähtökohtaisesti sidottu **aiheperusteisiin tilauksiin**. Tämän toteamuksen perusteella aliaiheiden tapahtumien tarkistus voidaan irrottaa **suodattimien** tarkistuksesta täysin. Tämä prosessi ja toteutustapa kuvataan tarkemmin luvussa 8.1.

7.5 Suorituskykytestit

Tietojärjestelmässä on jo olemassa **suorituskykytestit** tutkielman tavoitteiden onnistumisen arviointia varten. Järjestelmän suorituskykytestit ovat rakennettu simuloimaan oikeita käyttötapauksia tietojärjestelmässä. Tutkielmassa ei muuteta järjestelmän toiminallisuutta, joten uuden toteutuksen onnistumisen arviointi onnistuu, ilman muutoksia olemassa oleviin suorituskykytesteihin tai simulaatioihin. Samat suorituskykytestit pystytään ajamaan sekä nykyistä, että uutta prototyyppiä vasten ja näin ollen mahdollistaa suorituskyvyn vertailun eri toteutusten välillä. Testit on laadittu luvussa 6.3 esitellyn **Gatling**-suorituskykytestauskehyksen avulla.

Suorituskykytestien toiminnalliset **käyttötapaukset** ovat rakennettu niin, että ne luovat järjestelmän toiminnan kannalta huomattavan määrän dataa ja **tapahtumia**, jotka kaikki kulkevat tässä tutkielmassa parannettavan **tilauspalvelu**-sovelluskomponentin kautta käyttäjälle. Jokainen testiin syötetty käyttäjä luo saman määrän tilauksia sekä suodattimia.

Suorituskykytesteissä järjestelmään kirjautuu parametroitava määrä rinnakkaisia käyttäjiä, joilla on oma **istunto** jokaiselle. Testikäyttäjän suorittaessa jonkin käyttötapauksen, sen seurauksena syntyy **tapahtuma**. **Tilauspalvelu** käsittelee tapahtuman ja lähettää sen kaikille niille testikäyttäjien istunnoille, joille se **tilausten** sekä **suodattimien** perusteella kuuluu. Tapahtuman käsittelyn alkamisesta alkaen mitataan aika, milloin kyseinen tapahtuma saapuu viimeiselle ko. joukon istunnolle. Tätä aikaa mitataan millisekunteina ja koostetaan testien lopuksi yhteen. **Simulaation** kesto sekä käyttäjämäärä ovat konfiguroitavissa.

8 Prototyypin toteutusratkaisut

Tässä luvussa esitellään tutkielmassa kehitettävä prototyyppi, tulosten analysointi sekä analysointiin käytettävät työkalut ja teknologiat.

8.1 Prototyyppi – Tapahtumien ja tilausten yhdistäminen

Tutkielmassa toteutettavat muutokset tapahtumien ja tilausten tarkistukseen tullaan tekemään vain ja ainoastaan **tilauspalvelun** sisällä. Tämä johtuu siitä, että tapahtumien ja tilausten hallinnointi on täysin keskitetty tilauspalveluun, kuten luvussa 2.2 kerrottiin. **Tapahtuman** saapuessa tilauspalveluun luvussa 3.4 esitellystä **RabbitMQ**:n viestijonosta, tarkistetaan mille **istunnoille** kyseinen tapahtuma täytyy välittää. Tapahtumien tarkistus tilauspalvelussa voidaan jakaa karkeasti kahteen osaan:

- aiheperusteisten tilausten tarkistus,
- aiheperusteisten tilausten sekä istuntokohtaisten suodattimien tarkistus.

Suurimmat muutokset **tilauspalvelussa** tullaan tekemään yleisimpien **aiheiden** ja tapahtumatyyppien käsittelyyn sekä **suodattimien** evaluointiin. Prototyypissä tullaan hyödyntämään tilauspalvelussa jo olemassa olevaa hajautettua Hazelcast-muistitietoverkkoa tilausten käsittelyssä, mahdollistaen useamman palvelusolmun yhtäaikaisen käytön sekä korkean saavutettavuuden. Prototyypin kehittämisen tavoitteena on parantaa seuraavia osa-alueita **tilausten** käsittelyssä ja luonnissa:

- Aiheperusteisten tilauksien läpikäynti tilauskohtaisesti, istuntokohtaisen läpikäynnin sijasta.
- Suodattimien evaluoinnin optimointi.

Ensimmäinen osa-alue vastaa luvussa 2.2 esiteltyä tavoitetta suunnitella tilausten tietorakenne uudelleen ja toinen osa-alue taas kehittää parempi ratkaisu tapahtumien ja tilausten yhdistämiseen.

8.2 Aiheperusteisten tilausten käsittely

Yksittäiselle **aiheperusteiselle tilaukselle** tullaan rakentamaan uusi tietorakenne, joka hallinnoi kaikkia kyseisestä **aiheesta** (engl. *topic*) kiinnostuneita **istuntoja** tallentamalla ne. Tämä muutos hyödyntää luvussa 5.2 mainittua Publish/Subscribe -välittäjää, jonka muistinvaraisen tietorakenteen avaimena toimii tietty **tilaus** ja, jonka vastuulla on uusien tilauksien ylläpito liittyen tiettyyn aiheeseen. Myös prototyypin sisältämä uusi tietorakenne mahdollistaa kuormanjaon eri palvelusolmujen kesken.

Tilaus tallennetaan Hazelcast-muistitietoverkkoon. Hazelcast-muistitietoverkossa kyseinen tilaus tallennetaan avain-arvo-parina, jossa avaimena toimii tilauksen **aiheen** yksilöivä tunniste ja arvona kyseisen tilauksen oma **tietorakenne**. Tietorakenne koostuu seuraavista attribuuteista ja niiden tietotyypeistä:

- `String entityType;`
- `HashMap<String, Session> topicSubscriptionSessions;`

Istunnon luodessa tai päivittäessä **tilausta**, päivitetään Hazelcastissa olevaa tietorakennetta vastaamaan uusia muutoksia. Aina kun **tilauspalvelu** vastaanottaa uuden luvussa 7.2 esitellyn **tapahtuman,** tarkistetaan, onko kyseisellä tapahtumalla kenttä `mainEntityId`. Jos tapahtumalta löytyy kyseinen kenttä, voidaan istuntojen läpikäyntiä nopeuttaa huomattavasti hakemalla kyseinen tilaus suoraan uudesta tietorakenteesta. Mahdollinen `mainEntityId` vastaa tässä tapauksessa tilaus-tietorakenteen avainta. Tietorakenteen sisältämät istunnot kentässä `topicSubscriptionSessions` ovat esitelty taulukossa Taulukko 1. Kyseisen hajautustaulun avaimena toimii istunnon yksilöivä tunniste. Kenttä `entityType` sisältää tiedon kyseisen **aiheen** (engl. *topic*) tietotyypistä.

Tilaus-tietorakenteen **istuntojen** ylläpito vähentää **tilauspalvelun** läpikäymiä istuntoja, koska **aiheperusteisten tilauksien** tarkistuksen kohdalla tilauspalvelun ei tarvitse käydä yhtäkään ylimääräistä istuntoa läpi, kun kyseessä on `mainEntityId`:n sisältävä **tapahtuma**. Näitä tapahtumia ovat kaikki tietojärjestelmän generoimien **aliaiheiden** tapahtumat. Tapahtumien ja aiheperusteisten tilausten yhdistäminen on kuvattu alla olevassa koodiesi-

merkissä. Esitelty koodiesimerkki on ensimmäinen osa algoritmista, joka tulee korvamaan luvussa 7.4 esitellyn toteutuksen kokonaan.

```
void consume(event) {

    if (event.hasMainEntityId()) {

        container = subContainers.get(event.getMainEntityId());

        forEach (session in container.getTopicSubscripSessions)

        {

            sendEvent(session, event);

        }

    }

}
```

Koska järjestelmässä voi olla yhtäaikaisesti useita satoja **tilauksia**, täytyy Tilaus-tietorakenteen jatkuva kasvu ottaa huomioon. **Istunnon** lopettaessa tilauksen, joko HTTP-pyynnöllä tai uloskirjautumisen yhteydessä, täytyy kyseinen istunto käydä saman tien poistamassa tietorakenteesta. Tämä on perusteltua myös siksi, että käyttäjille ei lähetetä turhaan **tapahtumia aiheesta**, johon heillä ei ole enää tilausta.

8.3 Suodattimien käsittely

Koska yksittäinen **aiheperusteinen tilaus** vastaa aina jotain tiettyä **aihetta**, voidaan luvussa 8.2 esiteltyä tilaus-tietorakennetta hyödyntää myös **suodattimien** läpikäynnissä. Aikaisemmin esiteltyyn tilaus-tietorakenteeseen lisätään uusi kenttä aktiivisille istunnoille, joiden suodattimet ovat yhteensopivia kyseisen yksittäisen tilauksen `entityType`-kentän kanssa:

- `HashMap<String, Session> filterSessions;`

Kun uusi **entiteetti** eli **aihe** (engl. *topic*) luodaan siinä **mikropalvelussa**, joka vastaan sen logiikasta, lähetetään tieto luodusta aiheesta **tilauspalvelulle**. Tieto välitetään HTTP-pyynnöllä tilauspalvelun julkiseen rajapintaan. Tiedon saapuessa tilauspalveluun, se etsii ne aktiiviset **istunnot**, joiden **suodatin** sopii yhteen luodun aiheen kanssa, ja lisää kyseiset istunnot avain-arvo-parina `filterSessions`-hajautustauluun. Taulun avaimena toimii istunnon yksilöivä tunniste ja arvona istunto, joka on esitelty taulukossa Taulukko 1. Tämä prosessi toistetaan luonti- sekä muokkausoperaatioiden yhteydessä, jolloin saadaan oikeat lähtötiedot hajautustaulun ylläpitoon. On tärkeä huomata, että tämä prosessi tapahtuu ennen kuin luonti- tai muokkausoperaation seurauksena generoitu **tapahtuma** saapuu tilauspalveluun.

Motivaatio tähän toteutusratkaisuun on luvussa 5.2 esitetty idea vähentää etsittävien **tilausten** määrää ennen varsinaista **tapahtumien** ja tilausten **yhdistämistä**. Näin pystytään vähentämään kuormaa tapahtumakohtaisten tarkistusten osalta, joita **tilauspalvelu** vastaanottaa huomattavasti enemmän kuin yksittäisen aiheen luonti- tai muokkaustapahtumia. Tässä toteutusratkaisussa istuntokohtaisten **suodattimien** laskenta on siirretty lähemmäksi kyseisen suodattimen sisältämää **aihetta** (engl. *topic*).

Tietomalli ja rakenne taas mukailevat luvussa 5.2 esiteltyä indeksointirakennetta, missä saman **suodattimen** omaavat **tilaajat** luokitellaan kuuluvaksi samaan ryhmään. Lopputuloksena yksittäisen tilaus-tietorakenteen indeksissä on sisäinen tietomalli saman suodattimen omaavista tilaajista.

Tapahtuman saapuessa **tilauspalvelun** käsittelyyn, tarkistetaan, onko **aihe** sellainen, joka välitetään ainoastaan **aiheperusteisen** tilauksen perusteella vai myös **suodatuksen** perusteella. Vain aiheperusteisen tilauksen perusteella välitettävä tapahtuma on kuvattu aikaisemmassa luvussa 8.2. Jos aihe on sellainen, joka välitetään myös suodatuksen perusteella, käydään uudesta `filterSessions`-tietomallista hakemassa kaikki sen sisältämät **istunnot** ja välitetään tapahtuma niille. Tämän lisäksi tapahtuma välitetään myös kaikille aiheperusteisen tilauksen tehneille istunnoille, kuten luvussa 8.2 on esitelty. Toteutettu algoritmi korvaa luvussa 7.4 esitellyn toteutuksen kokonaan. Tämä prosessi kuvataan tarkemmin alla olevassa koodiesimerkissä.

```
void consume(event) {
   if (event.hasMainEntityId()) {
     container = subContainers.get(event.getMainEntityId());
     forEach (session in container.getTopicSubscripSessions) {
          sendEvent(session, event);
     }
   return;
   } // end of if
   else if (subContainers.getTypes().contains(event.getType) {
     container = subContainers.get(event.getEntityId());
     forEach (session in container.getFilterSessions) {
        sendEvent(session, event);
     }
     forEach (session in container.getTopicSubscripSessions) {
        sendEvent(session, event);
     }
   } // end of else if
   return;
} // end of consume
```

8.4 Tulosten analyysi

Tutkimuksessa kehitetyn **prototyypin** testaamiseen hyödynnetään järjestelmässä jo olemassa olevia Gatling-pohjaisia **suorituskykytestejä**, jotka on esitelty luvussa 7.5. Suorituskykytestit perustuvat **simulaatioihin**. Simulaatiot tullaan ajamaan niin sanotun **rasitustestin** periaatteella, joka on kuvattu luvussa 6.1.

Ajettava käyttäjämäärä, järjestelmässä suoritettavat toiminnalliset **käyttötapaukset** sekä HTTP-liikenteen seuraaminen tullaan suorittamaan viiden tunnin ajan. Viiden tunnin kesto on perusteltua, koska ajettavat **simulaatiot** sisältävät useita uudelleenkirjautumisia sekä käyttötapauksien toistoja, millä pystytään luomaan huomattavasti enemmän kuormaa **tilauspalvelulle**. Vaikka tutkimus keskittyykin ensijaisesti tilauspalvelun suorituskyvyn nopeuden parantamiseen, niin Gatling-kehyksen avulla saadaan myös prosentuaalinen osuus onnistuneista HTTP-pyynnöistä sekä tapahtumista.

Testauskertojen testausympäristöjen välillä ei ole eroa. **Suorituskykytestit** ajetaan olemassa olevalle sekä uudelle prototyypille samalla konfiguraatiolla. Testikierroksen yhtäaikainen **käyttäjämäärä** on 1135 käyttäjää, mikä vastaa tietojärjestelmän nykyisessä kehitysympäristössä noin 100 %-kuormaa tietojärjestelmässä. Yhden **testikierroksen** kokonaiskesto on noin 5 tuntia.

Suorituskykytestien generoimista **raporteista** tutkielman kannalta kiinnostavimmat mittarit ovat seuraavat:

- Käyttäjille lähetettyjen tapahtumien määrä,
- Tilauspalvelun tapahtumien yhdistämisalgoritmin käsittelyaika per aihe,
  - keskiarvo sekä maksimiaika.
- Tilauspalvelun suoritinkuorma (CPU - %),
  - keskiarvo.

Konfiguraatio testikierroksille:

- 1135 yhtäaikaista käyttäjää
- 5 h kesto

Käyttäjille lähetettyjen **tapahtumien** määrä kuvaa tutkielmassa kehitetyn prototyypin toiminnallista puolta ja sen avulla varmistutaan, että prototyyppi on vertailukelpoinen nykyisen toteutuksen kanssa. Toisin sanoen **tilauspalvelun** käsittelemä tapahtumien lukumäärä on yhtenevä toteutusten välillä. Tilauspalvelun tapahtumien **yhdistämisalgoritmin** käsittelyaika kuvaa onko tutkielmassa toteutettu prototyyppi suorituskyvyltään nopeampi kuin nykyinen toteutus. Tilauspalvelun **suoritinkuorma** taas kertoo, onko prototyypin aiheuttama suoritinkuorma pienempi kuin nykyisessä toteutuksessa samoilla resursseilla. Resursseilla tarkoitetaan testausympäristöä.

9 Suorituskykytestauskertojen tulokset ja niiden analyysi

Tässä luvussa esitellään suorituskykytestauskertojen tulokset ja niiden analysointi.

9.1 Suorituskykytestauksen tulokset

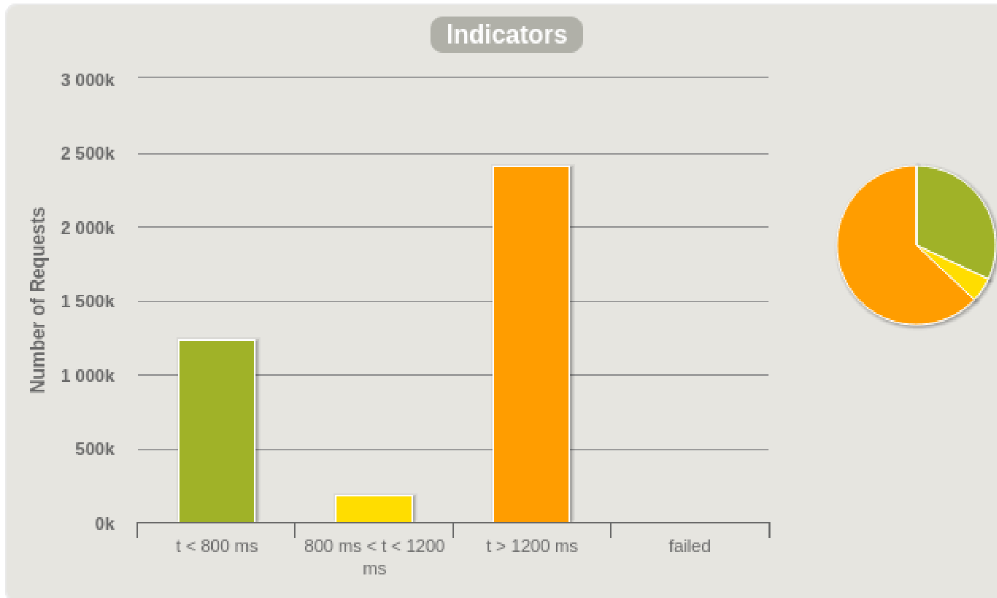
Tutkielmassa toteutettujen muutosten onnistumista arvioitiin tietojärjestelmän **suorituskykytestauksella**. Muutokset keskittyivät järjestelmän **tilauspalvelu**-sovelluskomponenttiin, jonka vastuulla on **tapahtumien** sekä **tilausten** yhdistäminen eli yhteensopivuuden tarkistaminen. Tärkeimpinä suorituskyvyn **mittareina** tarkastellaan tilauspalvelun suorittimen kuormaa sekä tapahtumien ja tilausten yhdistämisalgoritmin suoritusaikaa.

Suorituskykytestauksen raporteista saadaan myös tieto käyttäjille lähetetyistä **tapahtumista**, millä varmistutaan, että **tulokset** ovat vertailukelpoisia käsiteltävän kuorman suhteen. Nykyisen toteutuksen sekä uuden prototyypin testauskertojen aikana tietojärjestelmän **tilauspalvelu**-sovelluskomponentista oli kolme **palvelusolmua** eli erillistä **instanssia** käynnissä. Tämä mahdollisti **kuormanjaon** eri instanssien kesken. Nämä kolme instanssia esiintyvät tässä luvussa nimillä *APN-1*, *APN-2* sekä *APN-3*. Nykyisen toteutuksen tulokset ovat nimetty `master`, mikä viittaa ohjelmistokehityksen termein tietojärjestelmän pääkehityshaaraan versiohallinnassa. Tutkielmassa kehitetyn prototyypin nimenä tuloksissa toimii `pro gradu`.

9.2 Generoitujen tapahtumien lukumäärä

Sekä nykyisessä, että uudessa toteutuksessa suorituskykytestit ajettiin 1135 käyttäjällä. Nykyisessä toteutuksessa lähetettiin n. 3,82 miljoonaa tapahtumaa käyttäjille ja se on kuvattu kuviossa 5. Kun taas tässä tutkielmassa toteutetussa prototyypissä lähetettiin n. 3,76 miljoonaa tapahtumaa käyttäjille, jotka näkyvät kuviossa 6. Näiden tulosten samankaltaisuus kertoo siitä, että prototyyppi toimii oikein suhteessa nykyiseen toteutukseen.

> GET /notifications



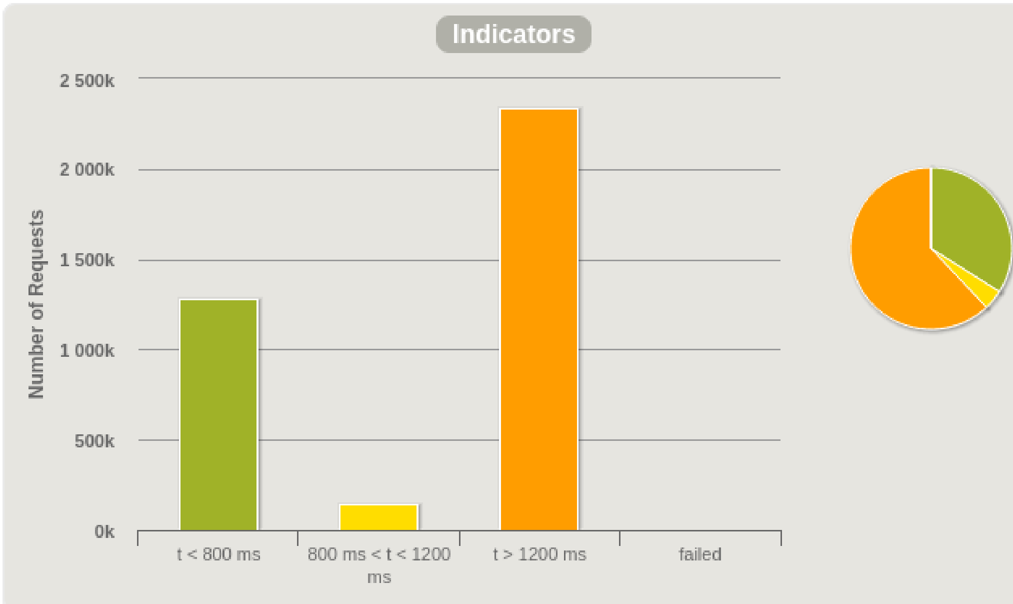
STATISTICS

Executions	Total	OK	KO
	3826575	3826575	0
Mean req/s	177.156	177.156	-

Response Time (ms)	Total	OK	KO
Min	11	11	-
50th percentile	7083	7083	-
75th percentile	10339	10339	-
95th percentile	10988	10988	-
99th percentile	11029	11029	-
Max	19980	19980	-
Mean	5581	5581	-
Std Deviation	4666	4666	-

Kuvio 5. Testin aikana lähetetyt tapahtumat testikäyttäjille nykyisessä toteutuksessa.

> GET /notifications



STATISTICS

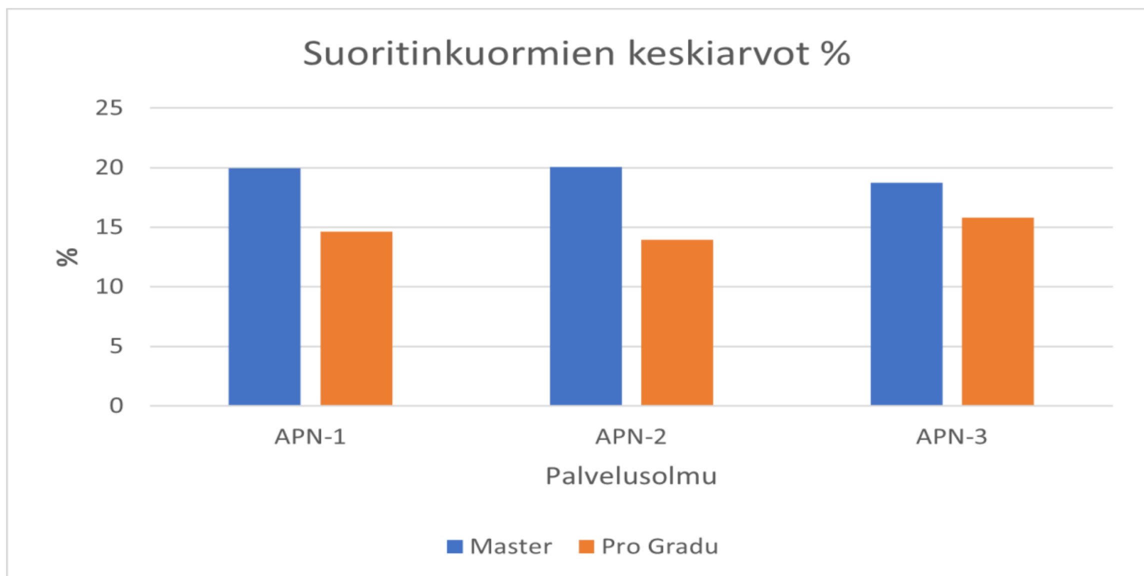
Executions	Total	OK	KO
	3762855	3762855	0
Mean req/s	174.206	174.206	-

Response Time (ms)	Total	OK	KO
Min	8	8	-
50th percentile	8084	8084	-
75th percentile	10368	10368	-
95th percentile	10988	10988	-
99th percentile	11023	11023	-
Max	20096	20096	-
Mean	5699	5699	-
Std Deviation	4725	4725	-

Kuvio 6. Testin aikana lähetetyt tapahtumat testikäyttäjille uudessa prototyypissä.

9.3 Tilauspalvelun suoritinkuorma

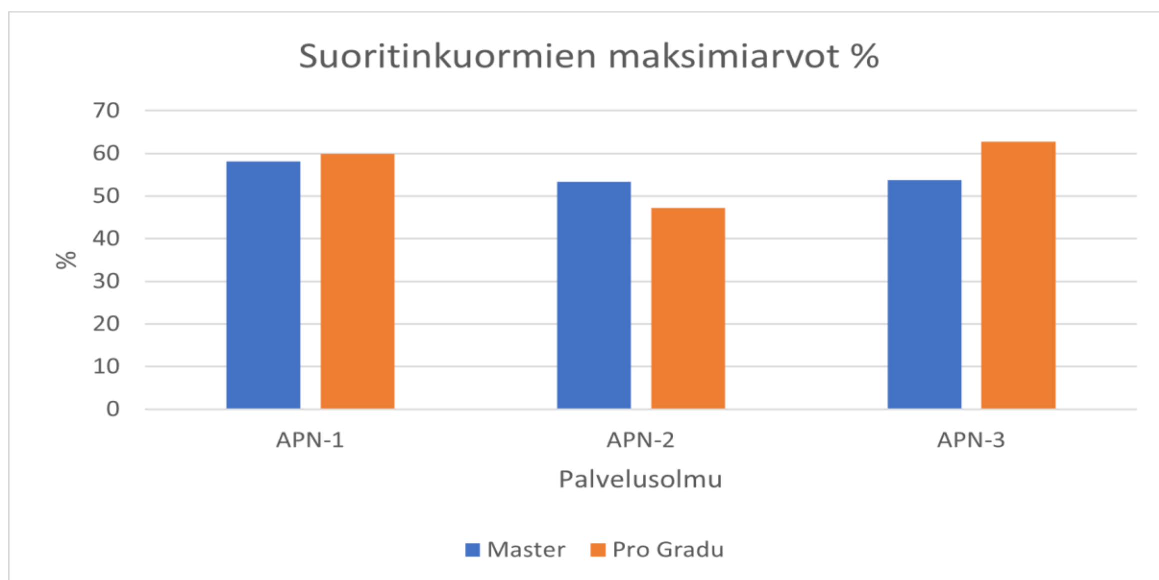
Suorituskykytestien aikana tietojärjestelmän **tilauspalvelusta** oli kolme erillistä **palvelusolmua**, joka mahdollisti järjestelmän **kuormanjaon** testien aikana. Palvelusolmujen kuormanjako ei ole osa tämän tutkielman aihetta ja ei toimi joka kerta samalla tavalla. Vaikka yksittäisen palvelusolmun suoritinkuorman vertailu eri toteutusten kesken ei käytännössä ole mahdollista tämän tutkielman puitteissa, saadaan jokaisen palvelusolmun suoritinkuorman **keskiarvo** sekä **maksimiarvo** kuitenkin selville. Suorituskykytestien **tuloksissa** näkyi suoritinkuorman keskiarvojen lasku jokaisen palvelusolmun kohdalla. Nämä keskiarvot ovat kuvattu kuviossa 7. Keskiarvot on laskettu koko testauskerran eli 5 tunnin ajalta.



Kuvio 7. Suoritinkuormien keskiarvo palvelusolmukohtaisesti.

Nykyisen toteutuksen **tilauspalvelun** suoritinkuormista näkee, että jokaisen **palvelusolmun** suoritinkuorman **keskiarvo** pyörii noin 20 %. Kuten kaaviosta voi todeta, uudessa prototyypissä **suoritinkuormien** keskiarvo laski useita prosenttiyksiköitä jokaisessa palvelusolmussa. Uuden prototyypin suoritinkuormien keskiarvot ovat noin 15 %. Vaikka keskiarvoltaan prototyypin suoritinkuorma oli pienempi niin kuviosta 8 taas näkee, että vain yhdellä palvelusolmulla suoritinkuorman maksimiarvo oli pienempi. Suuremmat maksi-

miarvot voivat selittyä prototyypissä ilmenneinä logiikkavirheinä, jotka ovat aiheuttaneet suuremman hetkellisen kuorman.



Kuvio 8. Suoritinkuormien maksimiarvot palvelusolmukohtaisesti.

9.4 Tapahtumien ja tilausten yhdistämisalgoritmin suoritusnopeus

Suoritinkuormien lisäksi toinen tärkeä onnistumisen **mittari** on tässä tutkielmassa parannetun **yhdistämisalgoritmin suoritusnopeus** verrattuna nykyiseen toteutukseen. Tietojärjestelmän arkaluontoisuuden vuoksi taulukoissa esitetyt **aiheet** (engl. *topic*) ovat nimetty tunnistamattomiksi aakkosien mukaan.

Taulukossa 3 on esitetty molempien toteutuksien **suoritusnopeuksien** keskiarvot **aiheittain** sekä **palvelusolmukohtaisesti**. Tässä tutkielmassa toteutetun prototyypin suoritusnopeudet ovat esitetty punaisella värillä. Taulukossa 4 on myös esitetty kolmen raskaimman aiheen, eli *Topic_A*, *Topic_B* sekä *Topic_C*:n maksimikestot.

Aihe	APN-1	APN-2	APN-3
Topic_A_master **Topic_A_Gradu**	2,43 sek *1,47 sek*	2,18 sek *1,38 sek*	2,39 sek *1,33 sek*
Topic_B_master **Topic_B_ Gradu**	2,49 sek *0,462 sek*	2,40 sek *0,431 sek*	2,40 sek *0,550 sek*
Topic_C_master **Topic_C_ Gradu**	478 ms *164 ms*	541 ms *149 ms*	488 ms *148 ms*
Topic_D_master **Topic_D_ Gradu**	226 ms *87 ms*	221 ms *81 ms*	221 ms *86 ms*
Topic_E_master **Topic_E_ Gradu**	162 ms *162 ms*	150 ms *159 ms*	150 ms *127 ms*
Topic_F_master **Topic_F_ Gradu**	259 ms *216 ms*	241 ms *223 ms*	251 ms *211 ms*
Topic_G_master **Topic_G_ Gradu**	93 ms *6 ms*	90 ms *6 ms*	94 ms *6 ms*

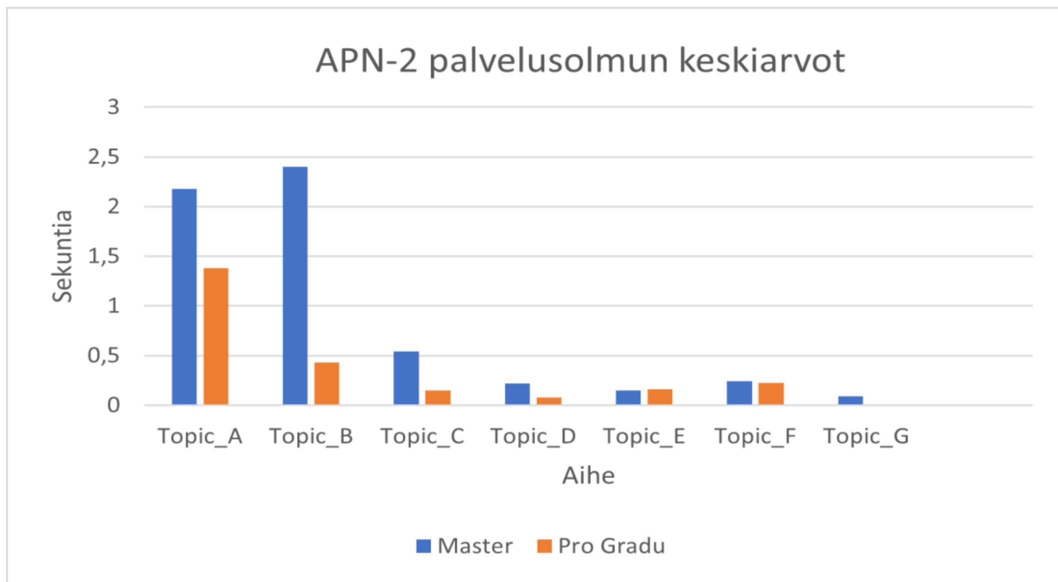
Taulukko 3. Yhdistämisalgoritmin suoritusnopeuksien keskiarvot eri toteutusten välillä.

Aihe	APN-1	APN-2	APN-3
Topic_A_master **Topic_A_Gradu**	18,83 sek *14,61 sek*	17,52 sek *12,89 sek*	19,35 sek *15,59 sek*
Topic_B_master **Topic_B_Gradu**	16,93 sek *13,91 sek*	16,08 sek *11,85 sek*	19,90 sek *13,52 sek*
Topic_C_master **Topic_C_Gradu**	13,01 sek *2,09 sek*	13,26 sek *2,52 sek*	14,01 sek *2,36 sek*

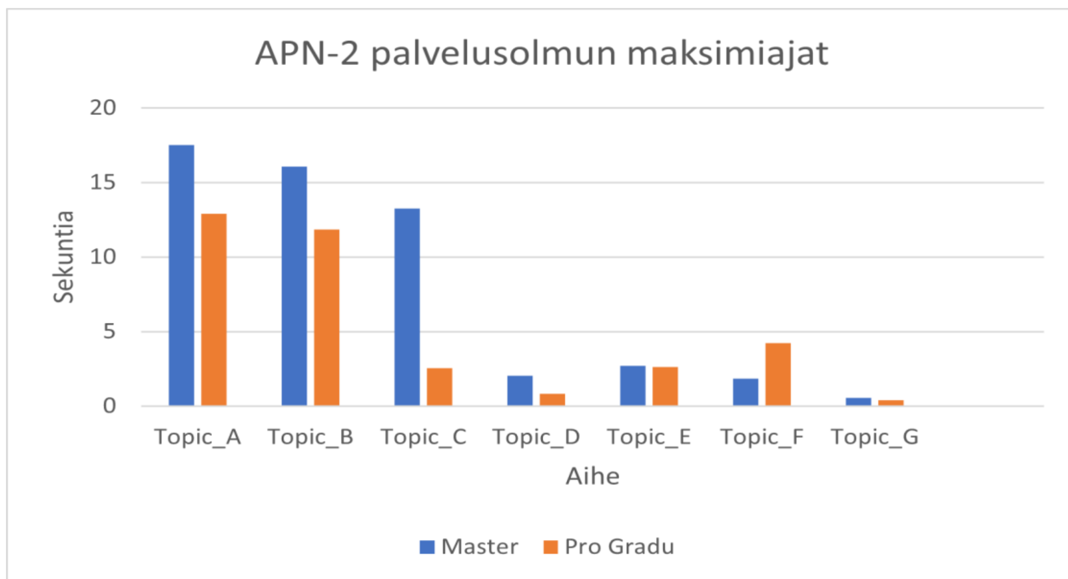
Taulukko 4. Yhdistämisalgoritmin suoritusnopeuksien maksimiarvot eri toteutusten välillä.

Kuten taulukoista 3 ja 4 huomaa, ovat suorituskykytestauksen tulokset yhdistämisalgoritmin **suoritusnopeuksien** kannalta myös pienentyneet. Melkein kaikkien **aiheiden** suoritusnopeudet laskivat jokaisella palvelusolmulla. Suurin ero **keskiarvoissa** on *Topic_B*:llä nykyiseen toteutukseen verrattuna, jolla keskiarvo laski noin 2,4 sekunnista alle puoleen sekuntiin. Kun taas suurin ero **maksimiarvoissa** on *Topic_C*:llä, jonka maksimiarvo laski noin 13 sekunnista noin 2,5 sekuntiin.

Vaikka **palvelusolmujen** välillä on pieniä eroja suoritusnopeuksissa, ei se tämän tutkielman kannalta ole oleellista. Palvelusolmujen kuormanjako ei ole yhtenevä ja ei toimi joka kerta samalla tavalla. Kuviossa 9 havainnollistetaan yksittäisen *APN-2*-palvelusolmun **suoritusnopeuksien keskiarvoja** molempien toteutusten välillä. Kuviossa 10 taas havainnollistetaan suoritusnopeuksien **maksimiaikojen** eroja toteutusten välillä kyseisellä palvelusolmulla. Keskiarvojen mittakaava on 0–3 sekuntia ja maksimiaikojen mittakaava taas 0–20 sekuntia.



Kuvio 9. APN-2-palvelusolmun yhdistämisalgoritmin keskinopeudet toteutusten välillä.



Kuvio 10. APN-2-palvelusolmun yhdistämisalgoritmin maksimiarvot toteutusten välillä.

Yhdistämisalgoritmin suoritusnopeuksista on tärkeä huomata, että kaikkein nopeinten käsitellyt **aiheet** `Topic_D`, `Topic_E`, `Topic_F` ja `Topic_G` ovat vain **aiheperusteisen** tilauksen perusteella välitettyjä aiheita kuten luvussa 8.2 on esitelty. Näiden kohdalla pro-

totyypin **tilauspalvelu** ei ole käynyt yhtään ylimääräistä istuntoa läpi, joten niiden suoritusnopeudet ovat huomattavasti pienempiä. Myös nykyisessä toteutuksessa kyseisten aiheiden käsittely on nopeaa, joten suuria eroavaisuuksia toteutusten välillä ei juurikaan ole. Aiheen `Topic_F` maksimiajassa on selvä hitaus nykyiseen toteutukseen verrattuna, mikä voi johtua virheellisestä käsittelystä prototyypissä. Kyseisen aiheen keskiarvo ei myöskään ole juurikaan pienentynyt. Kyseinen logiikkavirhe on todennäköisesti myös syy, miksi prototyyppi välitti n. 60 000 **tapahtumaa** vähemmän käyttäjille kuin nykyinen toteutus, kuten kuvioita 5 ja 6 vertailemalla voi todeta.

Aiheet `Topic_A`, `Topic_B` ja `Topic_C` taas ovat myös **suodattimien** perusteella välitettäviä aiheita, jotka on esitelty luvussa 8.3. Koska monimutkaisten suodattimien läpikäynti sekä **istuntojen** laskenta on raskaampi prosessi näiden kohdalla, niin myös suoritusnopeudet ovat huomattavasti hitaampia. Aiheet ovat myös kaikkein yleisimpiä, jolloin niiden kohdalla **tilauspalvelu** käy läpi huomattavasti enemmän istuntoja kuin vain **aiheperusteisen** tilauksen kohdalla. Keskiarvoltaan nopeampi käsittely kuitenkin näkyy myös palvelusolmujen suoritinkuormien keskiarvon alenemisessa.

10 Yhteenveto

Tämän tutkielman tarkoituksena oli parantaa olemassa olevan tietojärjestelmän **tilauspalvelu**-sovelluskomponentin suorituskyvyn nopeutta. Tutkielman tavoitteet olivatkin seuraavat:

- Parantaa tilauspalvelun nykyistä toteutusta uudelleen suunnittelemalla suodattimien sekä tilausten tietorakenteet sekä niiden läpikäynti.
- Kehittää tapahtumien ja tilausten yhdistämiseen tehokkaampi toteutusratkaisu.
- Arvioida toteutettujen muutoksien onnistuminen suorituskykytestien avulla.

Tietojärjestelmä perustuu hajautettuun **mikropalveluarkkitehtuuriin**. Mikropalveluiden välinen viestinvälitys on toteutettu **RabbitMQ**:n viestinvälitystekniikalla, joka perustuu **AMQP**-protokollaan. Viestinvälityksen **kommunikointimallina** on Publish/Subscribe -tyylinen malli, joka perustuu **tilaajiin** (engl. *subscribers*) sekä **julkaisijoihin** (engl. *publishers*). **Tilauspalvelu** kuuntelee kaikkien muiden mikropalveluiden generoimia **tapahtumia** sekä vastaa niiden välittämisestä järjestelmän loppukäyttäjien istunnoille.

Tapahtumien ja tilausten yhteensopivuuden tarkistus on keskeinen osa Publish/Subscribe -kommunikointimalliin perustuvien tietojärjestelmien **suorituskykyä**. Tutkielman kohteena olevan tietojärjestelmän nykyinen toteutus ei ole tarpeeksi tehokas suurilla käyttäjämäärillä. **Tilauspalvelun** vastaanottaessa **tapahtuman** joltain mikropalvelulta, käy se läpi kaikki aktiiviset **istunnot** löytääkseen ne tilaajat, joille tietty tapahtuma täytyy välittää.

Tutkielman tavoitteiden mukaisesti tilauspalveluun luotiin uusi tietorakenne sekä tapahtumien ja tilausten **yhdistämisalgoritmiin** tehtiin muutoksia, jotta **tilauspalvelu**-sovelluskomponentin **suorituskyky** paranisi. Tutkielmassa toteutetun prototyypin ideana oli vähentää itse yhdistämiseen kohdistuva kuorma siirtämällä **tapahtumien** ja **tilausten** yhdistäminen lähemmäksi kyseisen tilauksen **aiheen** (engl. *topic*) luonti- tai muokkausoperaatiota. Tällöin tilauspalvelulla on jo etukäteen tieto **istunnoista**, joille tietty tapahtuma täytyy välittää. Ratkaisu vähentää ylimääräisten istuntojen läpikäyntiä yksittäisen tapahtuman käsittelyn yhteydessä.

Tutkielman viimeisenä tavoitteena arvioitiin toteutettujen muutosten onnistumista tietojärjestelmän **suorituskykytesteillä**. Suorituskykytestit ajettiin **rasitustesti**-periaatteella ja testien konfiguraatio oli identtinen sekä nykyisen toteutuksen, että uuden prototyypin välillä. Suorituskykytestien tuloksissa otettiin huomioon istunnoille lähetettyjen **tapahtumien** määrä, tilauspalvelun **suoritinkuorman** keskiarvo sekä tilauspalvelun yhdistämisalgoritmin **suoritusnopeus** aihekohtaisesti.

Suorituskykytestien tuloksien perusteella prototyypin suorituskyky simuloidussa käytössä oli nopeampi kuin nykyinen toteutus. **Tilauspalvelusta** oli kolme **instanssia** eli **palvelusolmua** yhtäaikaisesti käytössä, mahdollistaen palvelun kuormanjaon. Prototyypin kohdalla tilauspalvelun jokaisen instanssin suoritinkuorman keskiarvo oli pienempi järjestelmän nykyiseen toteutukseen verrattuna. Suoritinkuorman maksimiarvot kuitenkin kasvoivat uuden prototyypin kohdalla. Myös itse tapahtumien ja tilausten **yhdistämisalgoritmin** suoritusnopeuksien **keskiarvot** ja **maksimikestot** laskivat melkein jokaisen aiheen kohdalla. Suoritinkuorman maksimiarvojen kasvu sekä aiheet, joiden kohdalla suoritusnopeudet olivat hitaampia kuin nykyisessä toteutuksessa vaativat lisää tutkimista sekä testaamista, jotta mahdolliset logiikkavirheet saadaan selville.

Koska tutkielmassa kehitetyn **prototyypin** onnistumista arvioitiin simulaatioiden avulla, ei suorituskykytestauksen tuloksia voi suoraan todeta käytännössä toimivaksi. Aiheen tutkimista on mahdollista jatkaa tietojärjestelmän kehityksen yhteydessä. Esimerkiksi **tapahtumien** ja **tilausten** keskittäminen yksittäiseen mikropalveluun sekä mikropalveluiden väliset riippuvuudet ovat hajautetun arkkitehtuurin perusoletuksen vastaisia. Prototyypin jatkokehitystä varten **suorituskykytestaus** tulisi suorittaa stressitesti-periaatteella, jolloin kuormitusta asteittain lisäämällä saadaan lisätietoa **tilauspalvelun** suorituskyvystä. Asteittaisella kuorman kasvattamisella saataisiin myös realistisempi kuva, kuinka prototyyppi toimisi todellisessa käytössä.

Lähteet

Aderaldo, C. M., Mendonça, N. C., Pahl, C., & Jamshidi, P. (2017, May). Benchmark requirements for microservices architecture research. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)* (pp. 8-13). IEEE.

Alshuqayran, N., Ali, N., & Evans, R. (2016, November). A systematic mapping study in microservice architecture. In *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)* (pp. 44-51). IEEE.

AMQP. (12.7.2021). AMBQ About. Haettu osoitteesta: `https://www.amqp.org/about/what/.`

Ashayer, G., Leung, H. K. Y., & Jacobsen, H. A. (2002, July). Predicate matching and subscription matching in publish/subscribe systems. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops* (pp. 539-546). IEEE.

Banno, R., Takeuchi, S., Takemoto, M., Kawano, T., Kambayashi, T., & Matsuo, M. (2014, July). A distributed topic-based pub/sub method for exhaust data streams towards scalable event-driven systems. In *2014 IEEE 38th Annual Computer Software and Applications Conference* (pp. 311-320). IEEE.

Di Francesco, P. (2017, April). Architecting microservices. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 224-229). IEEE.

Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, *150*, 77-97.

Fabret, F., Jacobsen, H. A., Llirbat, F., Pereira, J., Ross, K. A., & Shasha, D. (2001, May). Filtering algorithms and implementation for very fast publish/subscribe systems. In *Pro-*

ceedings of the 2001 ACM SIGMOD international conference on Management of data (pp. 115-126).

Fernandes, J. L., Lopes, I. C., Rodrigues, J. J., & Ullah, S. (2013, July). Performance evaluation of RESTful web services and AMQP protocol. In *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)* (pp. 810-815). IEEE.

Gatling. (12.7.2021). Gatling - Why Gatling? Haettu osoitteesta: `https://gatling.io/features/.`

Goel, D., & Nayak, A. (2019, December). Reactive microservices in commodity resources. In *2019 IEEE International Conference on Big Data (Big Data)* (pp. 3658-3665). IEEE.

Jokinen, T. (19.02.2021) Konstruktiivinen tapaustutkimus ja suunnittelutiede - kaksi insinööritieteisiin soveltuvaa tutkimusotetta. `https://blogi.oamk.fi/2021/02/19/konstruktiivinen-tapaustutkimus-ja-suunnittelutiede-kaksi-insinooritieteisiin-soveltuvaa-tutkimusotetta/.`

Jones, C. (10.05.2015). Performance, Load, Stress or Endurande Test? Which do you want? `https://www.linkedin.com/pulse/performance-load-stress-endurance-test-which-do-you-want-chris-jones/`

Kookarinrat, P., & Temtanapat, Y. (2016, July). Design and implementation of a decentralized message bus for microservices. In *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)* (pp. 1-6). IEEE.

Leontiadis, I. (2007, November). Publish/subscribe notification middleware for vehicular networks. In *Proceedings of the 4th on Middleware Doctoral Symposium* (pp. 1-6).

Leontiadis, I., Costa, P., & Mascolo, C. (2009). A hybrid approach for content-based publish/subscribe in vehicular networks. *Pervasive and Mobile Computing*, *5*(6), 697-713.

Lian, M., Yang, D., Li, M., Yu, S., & Gao, S. (2020, February). An Improved Subscription Classification Filtering Method Based on Multi-Index. In *2020 12th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)* (pp. 842-847). IEEE.

Magnoni, L. (2015, April). Modern messaging for distributed systems. In *Journal of Physics: Conference Series* (Vol. 608, No. 1, p. 012038). IOP Publishing.

Netto, M. A., Menon, S., Vieira, H. V., Costa, L. T., De Oliveira, F. M., Saad, R., & Zorzo, A. (2011, May). Evaluating load generation in virtualized environments for software performance testing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (pp. 993-1000). IEEE.

Pietzuch, P. R., & Bacon, J. M. (2002, July). Hermes: A distributed event-based middleware architecture. In *Proceedings 22nd International Conference on Distributed Computing Systems Workshops* (pp. 611-618). IEEE.

Pietzuch, P. R. (2004). *Hermes: A scalable event-based middleware* (No. UCAM-CL-TR-590). University of Cambridge, Computer Laboratory.

Qian, S., Cao, J., Zhu, Y., & Li, M. (2014, April). Rein: A fast event matching approach for content-based publish/subscribe systems. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications* (pp. 2058-2066). IEEE.

RabbitMQ. (5.7.2021). RabbitMQ Features. Haettu osoitteesta: `https://www.rabbitmq.com/#features.`

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016, December). The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)* (pp. 318-325). IEEE.

Sarojadevi, H. (2011). Performance testing: methodologies and tools. *Journal of Information Engineering and Applications*, *1*(5), 5-13.

Setty, V. J. (2015). Publish/subscribe for large-scale social interaction: Design, analysis and resource provisioning. *Doctoral Thesis*.

Setty, V., Kreitz, G., Vitenberg, R., Van Steen, M., Urdaneta, G., & Gimåker, S. (2013, June). The hidden pub/sub of spotify: (industry article). In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems* (pp. 231-240).

Turau, V., & Siegemund, G. (2017, June). Scalable routing for topic-based publish/subscribe systems under fluctuations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)* (pp. 1608-1617). IEEE.

Wu, Q., & Wang, Y. (2010, March). Performance testing and optimization of J2EE-based web applications. In *2010 Second International Workshop on Education Technology and Computer Science* (Vol. 2, pp. 681-683). IEEE.

Xie, W., Navathe, S. B., & Prasad, S. K. (2005, April). Filter indexing: A scalable solution to large subscription based systems. In *International Conference on Database Systems for Advanced Applications* (pp. 288-299). Springer, Berlin, Heidelberg.

Yang, J., Fan, J., & Jiang, S. (2016, December). DOCO: An efficient event matching algorithm in content-based publish/subscribe systems. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)* (pp. 200-207). IEEE.

Yu, M., Li, G., Wang, T., Feng, J., & Gong, Z. (2014). Efficient filtering algorithms for location-aware publish/subscribe. *IEEE Transactions on Knowledge and Data Engineering*, *27*(4), 950-963.

Zaarour, T., & Curry, E. (2019, September). Adaptive filtering of visual content in distributed publish/subscribe systems. In *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)* (pp. 1-5). IEEE.