Jukka Markkanen

Mikropalvelut ja Apache Kafka – vertailussa prosessointi-takuiden tehokkuus

Tietotekniikan pro gradu -tutkielma

3. helmikuuta 2024

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Jukka Markkanen

Yhteystiedot: juvimark@student.jyu.fi

Ohjaaja: Ari Viinikainen

Työn nimi: Mikropalvelut ja Apache Kafka – vertailussa prosessointitakuiden tehokkuus

Title in English: Microservices and Apache Kafka – comparing performance of processing guarantees

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 61+6

Tiivistelmä: Tutkielmassa tarkastellaan monipuolisten ohjelmistojen toteuttamista mikropalveluina, jotka ovat itsenäisiä ja riippumattomia toisista palveluista. Mikropalveluarkkitehtuuri tarjoaa etuja ohjelmistokehittäjille, mutta palveluiden välinen kommunikaatio voi muodostua haasteelliseksi. Perinteisiä synkronisia ja API-lähtöisiä protokollia käytetään edelleen palveluiden välisessä kommunikaatiossa, mutta asynkroninen kommunikaatio on noussut suosituksi vaihtoehdoksi, erityisesti mikropalveluiden itsenäisyyden tukemiseksi. Tässä tutkimuksessa keskitytään asynkronisen viestinnän toteuttamiseen Apache Kafka -tapahtumaväylän avulla. Tutkimuksen tavoitteena on mitata *täsmälleen kerran* -prosessointitakuun suorituskykyä mikropalveluympäristössä verrattuna *vähintään kerran* ja *korkeintaan kerran* -prosessointitakuisiin. Empiirinen osuus toteutetaan lähettämällä tapahtumia Apache Kafka -tapahtumaväylän kautta muuttujinaan muun muassa viestin koko ja käytetty prosessointitakuu.

Avainsanat: mikropalvelut, hajautetut järjestelmät, asynkroninen kommunikointi, tapahtumavirrat, Apache Kafka, prosessointitakuut, suorituskyvyn arviointi, ohjelmistokehitys

Abstract: This study examines the implementation of diverse software as microservices which are independent and autonomous from other services. Microservice architecture offers advantages to software developers but communication between services can pose

challenges. Traditional synchronous and API-based protocols are still used for inter-service communication but asynchronous communication has become a popular choice, particularly to support the independence of microservices. This research focuses on implementing asynchronous communication using Apache Kafka as an event-streaming platform. The objective is to evaluate the performance of the exactly-once processing guarantee in a microservices environment compared to at-least-once and at-most-once processing guarantees. The empirical analysis involves sending events through Apache Kafka considering factors such as message size and the chosen processing guarantee.

Keywords: microservices, distributed systems, asynchronous communication, event streams, Apache Kafka, processing guarantees, performance evaluation, software development

Kuviot

Taulukot

Sisältö

1 Johdanto

Monipuolisten ohjelmistojen toteuttaminen toisistaan irrallisina palveluina yksittäisen suuren monoliitin sijaan on kasvattanut suosiotaan merkittävästi. Tällaisia itsenäisesti toimivia, usein vastuualueeltaan hyvinkin rajattua ohjelmakokonaisuuksia kutsutaan mikropalveluiksi (Guidi, ym. 2017). Itsenäisyydellä tarkoitetaan, että kyseinen palvelu pyritään rakentamaan mahdollisimman riippumattomiksi muista palveluista, mikä helpottaa varsinkin ohjelmistojen vaakasuuntaista skaalausta (engl. horizontal scaling*)* (Kwan, ym. 2019). Tämä tarkoittaa yksinkertaistetusti yksittäisen palvelun monistamista koko monoliitin monistamisen tai sille resurssien lisäämisen (engl. vertical scaling) sijaan. Lisäksi mikropalveluarkkitehtuuri tuo huomattavia etuja ohjelmistokehittäjille helpottaen laajan ohjelmistokokonaisuuden yhtäaikaista kehittämistä (Wang, ym. 2021). Mikropalveluarkkitehtuurilla luotu järjestelmä on hyvä esimerkki hajautetusta järjestelmästä (Denis 2021).

Useissa tapauksissa mikropalveluiden täydellinen eristäminen toisistaan on kuitenkin haastavaa, ellei mahdotonta. Otetaan esimerkiksi käsittelyyn verkkokauppa, johon on toteutettu mikropalvelu, joka keskittyy käyttöliittymältä saapuviin tilauksiin. Tällöin voidaan kuvitella skenaario, jossa halutaan täydentää kyseistä tilausdataa tuotedatalla, mutta tuotedatan omistaakin tuotteisiin keskittyvä tuotemikropalvelu. Suuren päänvaivan monoliittiseen järjestelmään verrattuna aiheuttaakin näiden atomisiksi tehtyjen palveluiden välinen kommunikointi – samassa ohjelmassa sijaitsevien funktioiden suorittamisen sijaan joudutaan turvautumaan toiseen mikropalveluun, jonka vastuualuetta kyseinen operaatio on (Microsoft Corporation 2021).

Yleisesti palveluiden välisessä kommunikaatiossa on käytetty synkronista ja ohjelmointirajapintalähtöistä protokollaa, kuten palvelukeskeistä arkkitehtuuria tai REST-arkkitehtuurityyliä. Myös RPC-pohjaiset viitekehykset, kuten alun perin Googlen kehittämä gRPC, ovat yleisesti käytettyjä niin synkronisessa kuin asynkronisessakin viestinnässä (Wang;Hindman ja Stoica 2021). Monet ohjelmistokehittäjät ovatkin siirtyneet käyttämään asynkronista ajattelutapaa, joka voidaan katsoa suositeltavaksi erityisesti mikropalveluiden itsenäisyyden tukemiseksi. Asynkronisista kommunikaatioteknologioista hyviksi esimer-

keiksi voidaan laskea viestijonoihin perustuva ja AMQP-protokollan toteuttava RabbitMQ sekä tämän tutkielman keskiössä oleva Apache Kafka, joka liikuttaa dataa eräänlaisina tapahtumavirtoina.

Vaikka asynkroninen kommunikointi tarjoaa monia etuja, se ei ole ratkaisu jokaiseen mikropalveluprojektiin eikä aina sovi ainoaksi viestintätavaksi. Palatkaamme esimerkkiin tilaus- ja tuotepalveluiden välisestä suhteesta, jossa tarvitaan myös vastaus tuotepalvelulta tilauksen tietojen täydentämiseksi. Täydennetty data voi tulla tarpeeseen esimerkiksi tilauksen indeksointia varten, jotta tilaus voidaan käyttöliittymällä suoritetulla haulla löytää myös ajantasaisella tuotteen nimellä. Asynkroninen viestintä sopiikin parhaiten tilanteisiin, jossa lähettävän osapuolen tarvitsee korkeintaan varmistaa, että tieto pääsee jossain vaiheessa perille, mutta suoraa vastausta ei odoteta. Aiemmin esiteltyyn arkkitehtuuriin liittyen tällainen kunkin tilauksen tiedot asynkronisesti vastaanottava palvelu voisi olla erillinen sähköpostipalvelu tilausvahvistuksien lähettämiseen tai analytiikkapalvelu käyttäjän toimintojen seuraamiseen.

Luotettavuus, eli tiedon toimitus perille muuttumattomana on tärkeä huoli ei-synkronisessa viestinnässä – saapuuko jokainen viesti vastaanottajalle asti, saapuuko se sille kerran vai useammin ja saapuuko se sisällöltään samana. Tässä tarvitaan luottamusta valitun teknologian antamiin viestin kuljetus- ja prosessointitakuisiin. Tarjolla olevista viestinvälitysteknologioista Apache Kafka on julistanut mahdollistavan *täsmälleen kerran* -prosessointitakuun (Narkhede 2017). Tämän takuun tarkoituksena on varmistaa, että kaikki lähetetyt viestit prosessoidaan Kafkan sisällä kerran ja vain kerran.

Tämän tutkielman tavoitteena on antaa parempi ymmärrys Kafkan antaman *täsmälleen kerran* -prosessointitakuun suorituskyvystä mikropalveluympäristössä verraten sitä yksinkertaisempiin *vähintään kerran* ja *enintään kerran* -prosessointitakuisiin. Teoriaosuudessa esitellään hajautettujen järjestelmien ja niiden välisen kommunikaation historiaa sekä erilaisten prosessointitakuiden ja Kafkan toimintaperiaatteiden ymmärtämiseen vaadittavaa taustatietoa. Empiirinen osuus toteutetaan lähettämällä tapahtumia Kafka -tapahtumaväylän kautta muuttujinaan muun muassa viestin koko sekä käytetty prosessointitakuu.

Luvussa 2 esitellään lyhyesti hajautetut järjestelmät ja mikropalveluarkkitehtuuri. Luvussa 3 keskitytään tutkimuksen kannalta olennaisiin palveluiden välisiin kommunikaatiomuotoihin. Luku 3.5 on omistettu prosessointitakuusemantiikkojen esittelyyn siinä missä Luku 5 esittelee tutkimuksessa käytettävän Apache Kafka -ohjelmiston ja tämän tavat toteuttaa takuut. Luku 6 sisältää tutkielman käytännönosuuden tuloksineen ja analyyseineen, joiden pohjalta luvussa 7 pohditaan muun muassa tutkimusasetelman epäkohtia ja mahdollisia tulevia tutkimusaiheita. Lopuksi luvussa 8 vedetään koko tutkielma yhteen.

2 Hajautetut järjestelmät

Hajautettu järjestelmä (engl. distributed system) voidaan määritellä eri tasoilla ja tavoilla. Yksi tähän tutkielmaan sopiva määritelmä kuvaa sitä tiedonprosessointijärjestelmänä, joka koostuu n-kappaleesta tietokoneita, jotka työskentelevät yhteisen tavoitteen saavuttamiseksi kommunikoiden keskenään verkon kautta (Zettler 2023). Toinen osuvasti kuvaava selitys on ”kokoelma itsenäisiä laskentaelementtejä, jotka ilmenevät käyttäjilleen yksittäisenä, yhtenäisenä järjestelmänä” (Van Steen ja S. 2018). Tässä yhteydessä laskentaelementti voi olla joko fyysinen laite tai virtuaalinen sovellustason prosessi.

Historiassa ohjelmistoja on pitkään kehitetty keskitetyllä tavalla, jossa kaikki toiminnot ovat sijainneet yhdessä suoritettavassa kokonaisuudessa. Hajautetun järjestelmän vastakohtana toimiikin keskitetty järjestelmä, joka suoritetaan yhdellä tietokoneella yhdessä paikassa (Zettler 2023). Kuten edellisessä kappaleessa mainittu määritelmä kuvaa, käyttäjille nämä järjestelmät näyttäytyvät silti samalla tavalla. Keskitetyt järjestelmät ovat olleet itsestään selvä vaihtoehto aikana, jolloin ohjelmistot toimivat paikallisesti käyttäjän tietokoneessa, ilman yhteyttä verkkoon (Nuha;Nour ja Roger 2016).

Internetyhteyden kehittyessä on ensimmäisenä yleistynyt yksinkertainen asiakas–palvelin (engl. client–server) -malli, jossa laskentaa ja datan tallennusta vaativa osuus on ulkoistettu tehokkaille palvelinkoneille käyttäjän oman keskusyksikön vastatessa ainoastaan käyttöliittymästä ja siihen liittyvästä logiikasta. Vielä tällöin palvelimilla suoritettava sovellus on ollut lähes aina yksittäinen, usein valtavaksi paisunut monoliitti.

Ketterien kehitysmetodien yleistyessä ja internetyhteyksien edelleen nopeutuessa monoliittisen systeemin kehittäminen on alkanut muodostua epätehokkaaksi ja sille on alettu etsiä vaihtoehtoisia tapoja kehittää ohjelmistoa pienimmissä, itsenäisesti julkaisukelpoisissa ja toisilleen aktiivisesti kommunikoivissa osasissa (Rui;Shanshan ja Zheng 2017). Tämä tarve on lopulta johtanut nykyään hyvin laajasti käytettyyn mikropalveluarkkitehtuuriin.

Tässä tutkielmassa keskitytään mikropalveluarkkitehtuuriin, jolla toteutettu järjestelmä täyttää hajautetun järjestelmän määritelmän. Jotta voidaan ymmärtää mikropalveluarkkitehtuurin tarvetta, on tärkeää tarkastella ohjelmistoarkkitehtuurin historiaa. Tässä luvussa

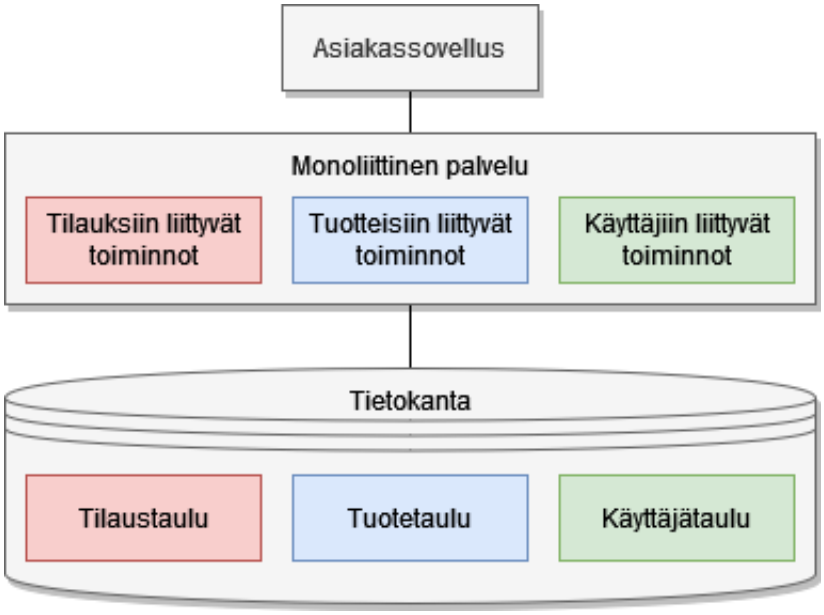
esitellään muun muassa monoliittinen ohjelmistoarkkitehtuuri, joka toimii hyvänä esimerkkinä keskitetystä järjestelmästä. Sitä pidetäänkin usein mikropalveluarkkitehtuurin edeltäjänä ja jopa vastakohtana.

2.1 Ohjelmistoarkkitehtuuri

Ohjelmistokehityksessä arkkitehtuuri pyrkii yhdistämään järjestelmälle määritellyt laadulliset vaatimukset sen toiminnallisuuteen (Dragoni, ym. 2017). Ohjelmistoarkkitehtuurin historia ulottuu jo 1960–1970-luvuille, jolloin alan tutkimuksessa huomattiin tarvetta erillisille suunnitteluvaiheen työkaluille ja merkintätavoille (Perry ja Wolf 1992). Tällöin arkkitehtuuri yhdistettiin vielä osaksi ohjelmistosuunnitteluprosessia. Tieteellisen perustan ohjelmistoarkkitehtuurikäsitteelle loivat Perry & Wolf (1992) artikkelissaan *Foundations for the Study of Software Architecture* erottaen sen lopullisesti ohjelmistosuunnittelusta ja vauhdittaen niin alan tutkimusta kuin arkkitehtuurin käytännön hyödyntämistä (Dragoni, ym. 2017). Arkkitehtuurista on siitä lähtien tullut korvaamaton osa sovelluskehitystä ja ylläpitoa.

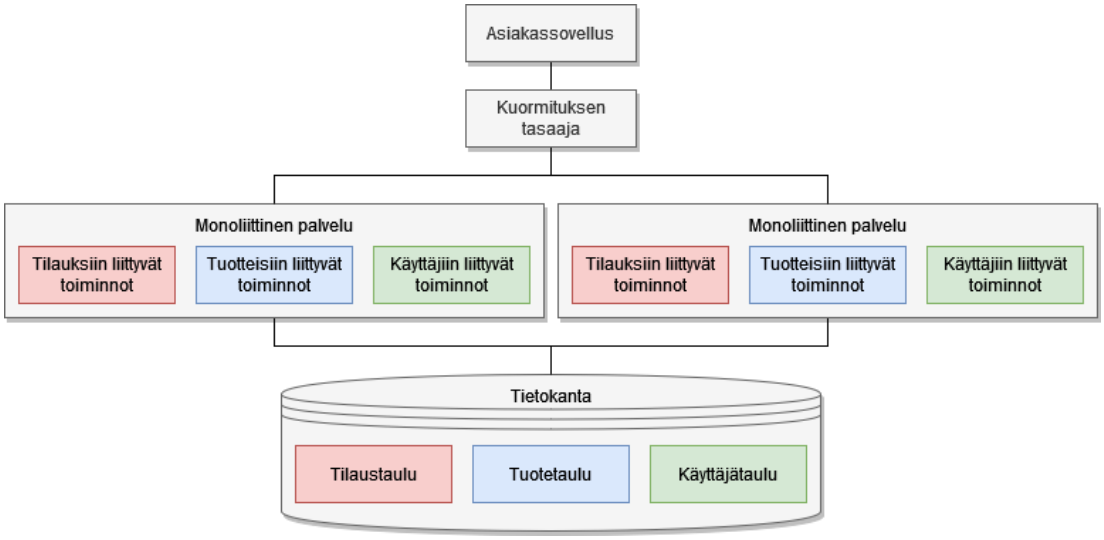
2.2 Monoliitti

Monoliitti on keskitetty järjestelmä, jonka osia ei voida ajaa itsenäisesti. Tällöin se myös suoritetaan yhdellä ja samalla loogisella laitteella (Dragoni, ym. 2017). Perinteisessä, arkkitehtuuriltaan yksinkertaisessa monoliitissa on omat hyvät puolensa. Se on lähtökohtaisesti yksinkertaisempi kehittää, testata, julkaista sekä tietyissä tilanteissa myös skaalata (Rui;Shanshan ja Zheng 2017).



Kuvio 1. Kaaviokuva yksinkertaisesta monoliitista

Monoliittien tapauksessa skaalaus tapahtuu kuitenkin usein ainoastaan pystysuuntaisesti, mikä tarkoittaa ohjelmistoa suorittavan tietokoneen suorituskyvyn nostamista. Tämä on kuitenkin rajoitettua, koska resursseja on mahdotonta nostaa määräänsä enempää (But 2019). Vaikka monoliitin skaalaus myös horisontaalisesti on teoriassa mahdollista, se edellyttää koko ohjelmiston monistamista jokaiselle tietokoneelle. Lisäksi monoliittisessa järjestelmässä tietokantayhteyksien ja integraatiomäärien kasvaessa yksinäiset, valtavaksi paisuneet taustajärjestelmät muuttuvat yhä vaikeammiksi hallita (Schmutz 2018). Kehitystyylin myös siirtyessä yhä enemmän ketteriin, moniosaaviin tiimeihin, on monoliittisen järjestelmän kehitys yhtäaikaisesti alkanut osoittautua haastavaksi.



Kuvio 2. Monoliitin horisontaalinen skaalaus

Dragoni, ym. (2017) vetävät artikkelissaan yhteen monoliittijärjestelmän haittapuolia seuraavasti:

1. Suuret monoliittijärjestelmät ovat vaikeita ylläpitää niiden kompleksisuuden takia.
2. Monoliitit johtavat usein ”riippuvuushelvettiin” (engl. dependency hell)
3. Joka ikinen muutos ohjelmakoodiin vaatii koko ohjelmiston uudelleenkäynnistyksen
4. Eri osat vaativat usein epätasaisesti resursseja ja monoliitissa joudutaan miettimään kaikkien niiden yhteismääriä
5. Ainoa keino skaalata monoliittia horisontaalisesti on monistaa koko ohjelmisto
6. Monoliitti pakottaa kehittäjiä käyttämään samoja teknologioita läpi ohjelmiston

On kuitenkin hyvä huomata, että monoliitti on edelleen relevantti ja usein paras vaihtoehto kehitysprosessissa, kun kyseessä on yksikertainen ja tarkasti rajattu ohjelmisto ja/tai pieni tiimi. Tällöin mikropalveluarkkitehtuuri tuo tarpeetonta monimutkaisuutta ja enemmän liikkuvia osia. Lisäksi monoliitissa datan ajantasaisuus (engl. consistency) on lähtökohtaisesti varmempaa (Mohr 2019). Kyseinen väittämä liittyy oleellisesti tämän työn päämäärään, sillä tutkielman on omalta osaltaan tarkoitus antaa parempi kokonaiskuva ajantasaisuuskysymyksestä mikropalveluiden välillä.

2.3 Palvelukeskeinen arkkitehtuuri

Mikropalveluiden eräänlaiseksi esiasteeksi voidaan ajatella palvelukeskeistä arkkitehtuuria (engl. service-oriented architecture, SOA), joka niin ikään täyttää hajautetun järjestelmän määritelmän. Palvelukeskeisestä arkkitehtuurista on alettu puhua ensimmäisen kerran 1990-luvun loppupuolella (IBM 2021) ja se on ajan myötä kasvanut erittäin suosituksi ratkaisuksi sekä tutkimuksen että investointien näkökulmasta, sen arvioidun markkina-arvon ollessa maailmanlaajuisesti vuonna 2020 edelleen 13,8 miljardia dollaria (Bohloul 2021).

Palvelukeskeisessä arkkitehtuurissa pyritään hyödyntämään uudelleenkäytettäviä komponentteja ja palveluita. Jokaisen palvelun on tarkoitus käsittää logiikkaa vain oman vastuualueensa tai toiminnallisuutensa osalta sisältäen siihen mahdollisesti liittyvät integraatiot. Palvelukeskeisessä arkkitehtuurissa painotetaan palveluiden löyhää kytkentää toisiinsa, mikä mahdollistaa niiden kutsumisen ilman tietoa toteutustavasta tai toteutuksessa käytetystä ohjelmointikielestä. Siitä huolimatta käytetty kommunikaatio on usein synkronista, mistä lisää luvussa 3. (IBM 2021)

2.4 Mikropalvelut

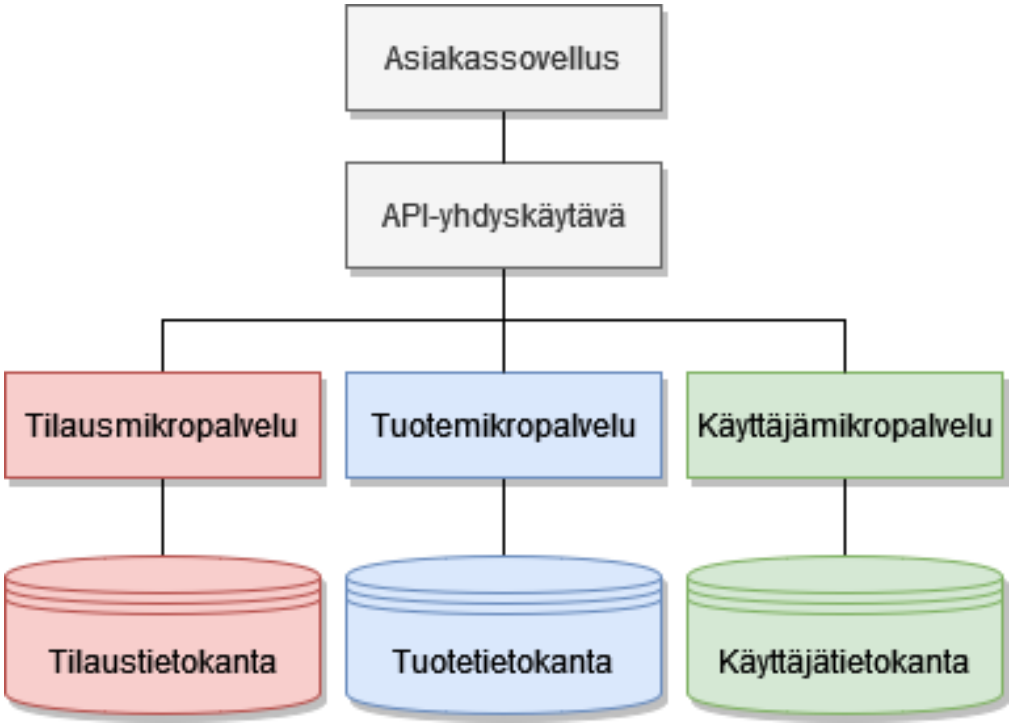
Mikropalvelu on käsitteenä verrattain tuore. Se on mainittu tiedetysti ensimmäisen kerran vuonna 2011 ja otettu virallisesti käyttöön mikropalveluarkkitehtuuria kehittäneiden arkkitehtien toimesta vuonna 2012 (Foote 2021). Thönes (2015) määrittelee mikropalvelun olevan pieni ohjelma, jota voidaan julkaista, skaalata ja testata itsenäisesti. Palvelukeskeisen arkkitehtuurin tapaan mikropalvelulähtöisen ajattelun on siis tarkoitus ratkaista aiemmin listattuja monoliittisen arkkitehtuurin ongelmia luomalla joukko ohjelmia, joilla on kullakin oma vastuualueensa.

Sekä mikropalveluarkkitehtuurissa että palvelukeskeisessä arkkitehtuurissa korostetaan palveluiden löyhää liitosta ja uudelleenkäytettävyyttä (IBM 2021). Mikropalveluarkkitehtuurin voidaan kuitenkin ajatella tavoittelevan erityisesti olio-ohjelmoinnin periaatteistakin tuttua, yksittäisen palvelun mahdollisimman atomista vastuualuetta (Guidi, ym. 2017). Yhden mikropalvelun tulisi olla vastuussa vain ja ainoastaan yhdestä toiminnallisuudesta ja

sen muuttamiseksi tai korvaamiseksi tulee olla vain yksi ainoa syy. Lisäksi mikropalveluiden välisessä kommunikaatiossa pyritään käyttämään mahdollisuuksien mukaan ei-synkronisia menetelmiä, kuten myöhemmissä luvuissa tarkemmin käsitellään.

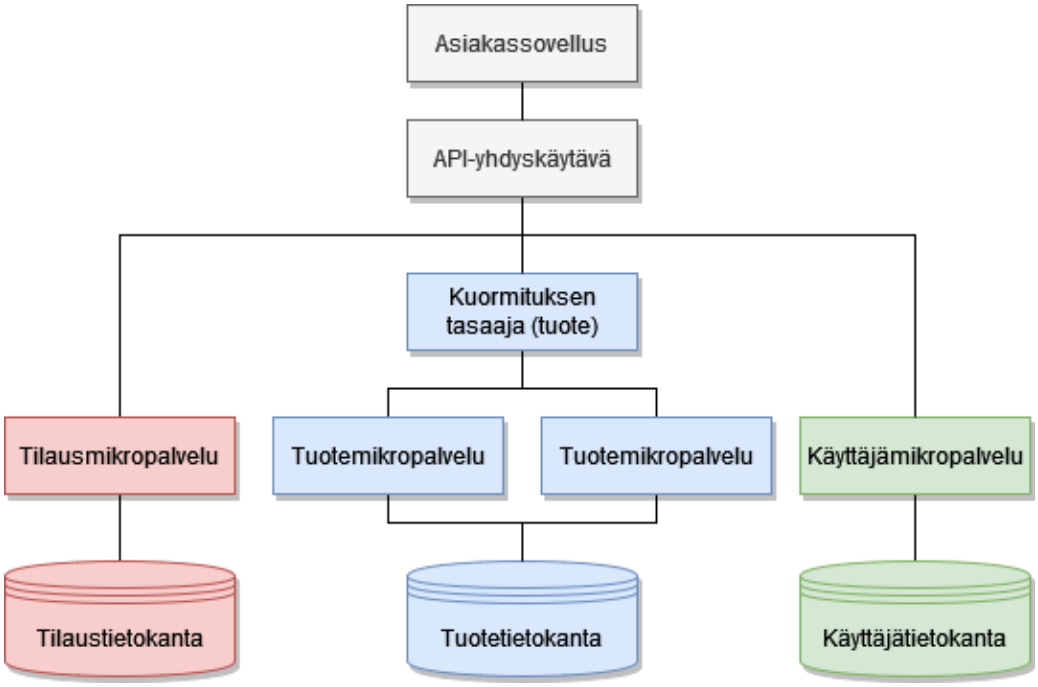
Edellä mainittujen seikkojen aiheuttaman ketteryyden menettämisen pelossa mikropalveluissa voidaan havaita enemmän lähdekooditasolla tapahtuvaa uudelleenkäyttöä, kuten yhteisten funktioiden sijoittamista samaan kirjastoon, kun taas palvelukeskeisessä arkkitehtuurissa pyritään uudelleenkäyttämään kokonaista palvelua integraatioineen (IBM 2021). Mikropalveluissa korostetaan myös korkeaa koheesiota datan suhteen, mikä viittaa siihen, että jokaisella mikropalvelulla tulisi olla oma tietokantansa, johon muilla palveluilla ei ole pääsyä (Foote 2021).

Yksinkertainen esimerkki mikropalveluilla toteutetusta järjestelmästä on johdannossakin sivuttu verkkokauppa, jossa yksi mikropalvelu vastaa tilauksista, toinen tuotteista ja lisäksi kolmas käyttäjähallinnasta. Jokaisella integraatiolla mihin tahansa kolmannen osapuolen palveluun on oma vastuumikropalvelunsa. Jokaisella palvelulla on oma tietokantansa, eikä millään niistä ole pääsyä toistensa tietokantoihin (Kuvio 3).



Kuvio 3. Esimerkkijärjestelmä toteutettuna mikropalveluina

Ohjelmistokokonaisuuden tuottaminen mikropalveluna helpottaa myös suunnitteluvaihetta, erityisesti mikäli kyseessä on laajempi kokonaisuus. Optimaalisessa tilanteessa asiakkaiden liiketoiminta-alueista voidaan suoraan muodostaa selkeät vastuualueet mikropalveluille. Lisäksi voidaan päätyä tilanteeseen, jossa joitakin mikropalveluja voidaan aloittaa kehittämään ennen muita tai jokin osa kokonaisarkkitehtuuria saadaankin jo valmiiksi tuotettuna kolmannen osapuolen tuottamana. Kehitysvaiheessa jokaista mikropalvelua kohden voidaan perustaa oma kehitystiiminsä, joka voi julkaista päivityksiä ohjelmakoodiin oman aikataulunsa mukaan.



Kuvio 4. Yksittäisen mikropalvelun horisontaalinen skaalaus

2.4.1 Reaktiivisuus

Kuten aiemmin todettua ovat käyttäjien vaatimukset verkkopalveluille kasvaneet vuosien saatossa, ja niihin sisältyy usein tarve lyhyille vasteajoille ja täydelle saavutettavuudelle mihin aikaan vuorokaudesta tahansa. Tämä tuottaa ohjelmistokehittäjille vaikeuksia, koska aiemmin tällaiselle nykykäyttäjän tarpeet täyttävälle ohjelmistolle ei ollut olemassa mallia tai määritelmää. Tästä syystä ohjelmistokehitysyhteisö on julkaissut reaktiivisen manifestin (The Reactive Manifesto 2014).

Reaktiivinen manifesti (Bonér;Farley, ym., The Reactive Manifesto 2014) määrittelee reaktiivisen järjestelmän ominaisuuksiksi seuraavat:

- **responsiivisuus** eli lyhyt vasteaika,
- **resilienssi** eli kyky palautua virhetilanteista,
- **elastisuus** eli käytössä olevien resurssien muovautuvuus ja
- **viestilähtöisyys**.

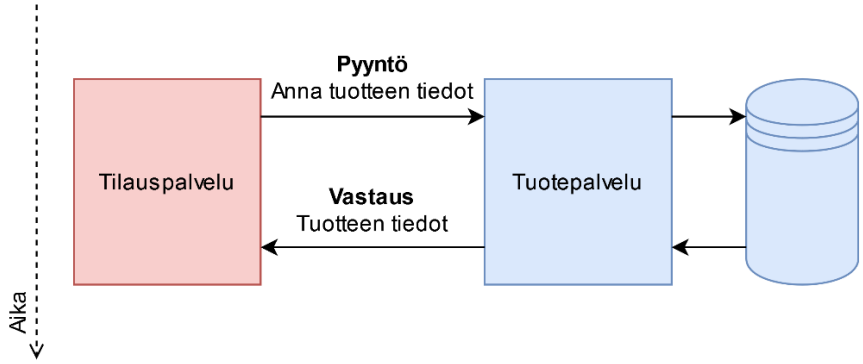
Kun yhdistetään mikropalveluarkkitehtuuri ja järjestelmän reaktiivisuus, saadaan tuloksesi reaktiivinen mikropalveluarkkitehtuuri (Mohr 2019). Tämän tutkielman tulokset liittyvät olennaisesti lähes kaikkiin edellä esitellyistä ominaisuuksista.

3 Palveluiden välinen kommunikaatio

Kuten hajautettujen järjestelmien esittelyssä aiemmin mainitaan, on järjestelmän osien saumaton kommunikointi toistensa kanssa erittäin tärkeää yhteisen tavoitteen saavuttamiseksi. Jotta voidaan ymmärtää tutkielman kannalta keskeisiä asynkronisia kommunikaatiomentelmiä, on hyödyllistä käydä ensin läpi muita vaihtoehtoja.

3.1 Synkroninen kommunikaatio

Synkronisuus tarkoittaa, että sekä asiakkaan (engl. client) roolissa toimivan palvelun, että palvelua tarjoavan (engl. serve) palvelun täytyy olla saatavilla samanaikaisesti (Raynal 2013). Asiakas pyytää jotain palvelimelta, johon palvelin vastaa asiakkaan jäädessä odottamaan vastausta. Esimerkkijärjestelmässä tämä voisi tarkoittaa tilannetta, jossa tilauspalvelu jää odottamaan vastausta tuotepalvelulta, jotta se voi jatkaa toimintoaan tuotetietoa käyttäen (Kuvio 5).



Kuvio 5. Esimerkkikaavio synkronisesta kyselystä kahden mikropalvelun välillä

Tämä voi olla hyvinkin yleinen tilanne myös mikropalveluarkkitehtuurissa, jos asynkronisuutta ei ole otettu alun perin huomioon arkkitehtuurin suunnitteluvaiheessa tai halutaan välttää liiallista kompleksisuutta. Synkronisuus on kuitenkin vahvasti ristiriidassa mikropalveluiden itsenäisyyden ja reaktiivisuuden kanssa.

3.2 Synkroniset ohjelmointirajapintaprotokollat

Seuraavissa alakohdissa esitellään synkronisessa kommunikaatiossa yleisimmin käytettävät protokollat.

3.2.1 SOAP

SOAP (Simple Object Access Protocol) on pääasiassa synkronisesti käytetty, tekstipohjainen ohjelmointirajapintaprotokolla (engl. application programming interface protocol, API protocol), joka hyödyntää sovellustasolla HTTP:tä (Hypertext Transfer Protocol) (Riley 2019). SOAP on yksi ensimmäisistä protokollista, joka mahdollistaa resurssien jakamisen palveluiden välillä järjestelmällisesti verkon kautta (Riley 2019). SOAP-sanomat perustuvat XML-tiedostomuotoon, mikä voi kokonsa puolesta aiheuttaa suorituskykyongelmia sekä käännösvaiheessa että suuren tietomäärän siirrossa (Abu-Ghazaleh;Govindaraju ja Lewis 2004). SOAP-protokollaa ei pidä sekoittaa palvelukeskeiseen arkkitehtuuriin (SOA), vaikka niitä usein käytetäänkin yhdessä.

3.2.2 REST

REST (Representational State Transfer) on vuonna 2000 esitelty ja nykyään vallitseva synkroninen ohjelmointirajapintaprotokolla, jonka tavoitteena on muun muassa ratkaista SOAP-protokollan ongelmia. XML:n lisäksi se tukee helpommin luettavaa ja pienemmän tilan vievää JSON-formaattia. REST-rajapintojen etuna pidetään myös kykyä säilyttää tietoa välimuistissa. Kuten SOAP, luottaa se sovellustasolla HTTP-protokollaan. (Riley 2019)

RESTful on termi, jota käytetään REST-protokollaa käyttävien rajapintojen yhteydessä. Se on viitekehys, jonka avulla pyritään ylläpitämään REST-rajapintojen yhtenäisyyttä ja tekemään niistä mahdollisimman intuitiivisia käyttää.

3.2.3 RPC ja gRPC

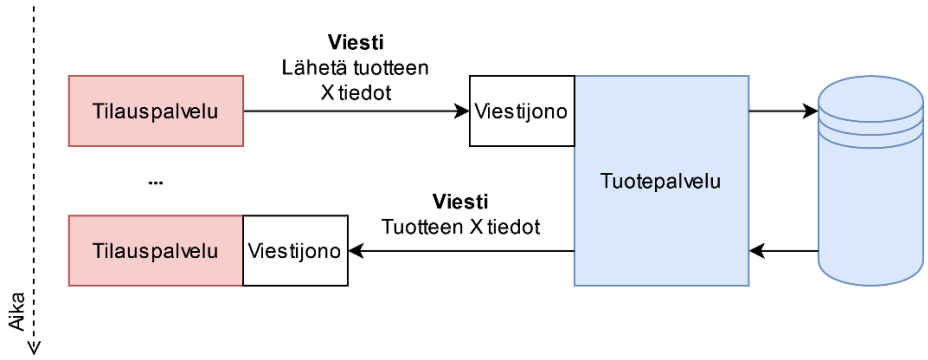
Ohjelmointirajapintaprotokolla gRPC on avoimen lähdekoodin RPC-protokollan toteutus, joka on kehitetty Googlen toimesta. Se luottaa niin ikään HTTP-protokollaan tiedonsiirros-

sa ja sitä käytetään pääasiassa synkronisesti. Toisin kuin aiemmat protokollat, gRPC ei ole sidoksissa yleisiin HTTP-verbeihin, kuten GET, POST tai DELETE. gRPC-pyynnöt näkyvät palvelevalle osapuolelle kuin mikä tahansa paikallinen metodipyyntö, mikä yksinkertaistaa sen käyttöön tarvittavaa koodia. (Riley 2019)

gRPC on saavuttanut suosiota erityisesti järjestelmissä, joissa pyyntöjen määrä on suuri (Riley 2019). Kari Patana on omassa omassa pro gradu -tutkielmassaan (2020) todennutkin, että gRPC-protokolla yhdessä sen käyttämän Protocol Buffers -viestimuodon kanssa on verkon yli REST-protokollaa nopeampi vaihtoehto.

3.3 Asynkroninen kommunikaatio

Sekä mikropalveluiden löyhä kytkentä toisiinsa että reaktiivisuus sisältävät ajatuksen niiden välisen kommunikaation asynkronisuudesta. Viesti toiselle palvelulle voidaan todeta lähetetyksi, mutta lähettäjä ei jää odottamaan vastausta — viestin vastaanottaja itse on vastuussa viestiin reagoinnista. Tämä tarkoittaa asynkronisen kommunikoinnin olevan epäsopiva käyttötapauksiin, joissa suoraa vastausta tarvitaan. Siksi asynkronisuus on erittäin tärkeä ottaa huomioon jo kokonaisarkkitehtuurin suunnitteluvaiheessa. Miten toimitaan, jos viesti ei saavu? Entä jos viesti saapuu tuplana?



Kuvio 6. Esimerkkikaavio asynkronisesta, viestijonoihin perustuvasta kommunikaatiosta

3.4 Viestilähtöisyys

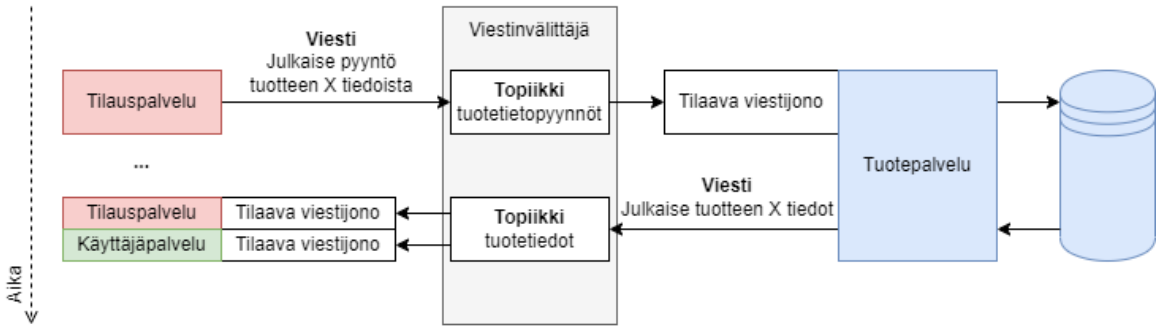
Yleisin tapa saavuttaa kommunikoinnin asynkronisuus on hyödyntää viestejä, jotka ovat mainittu jo reaktiivisen järjestelmän määrittelyssä. Järjestelmän osien löyhän liitoksen lisäksi pyritään niiden käytöllä varmistamaan palveluiden itsenäisyys ja sijainnin läpinäkyvyys, jotka ovat kaikki tärkeitä osia reaktiivisen järjestelmän määritelmää (Bonér;Farley, ym., The Reactive Manifesto 2014). Mikropalveluarkkitehtuurin tapauksessa komponentti tarkoittaa yksittäistä mikropalvelua.

Viestijärjestelmät hyödyntävät lähes poikkeuksetta julkaise—tilaa (engl. publish—subscribe) -mallia, jossa viestin lähettävää sovellusta kutsutaan julkaisijaksi (engl. publisher) ja viestejä vastaanottavaa sovellusta kutsutaan tilaajaksi (engl. subscriber). Näiden välillä toimivaa palvelinta kutsutaan viestinvälittäjäksi (engl. message-broker).

Viestinvälitysteknologioiden yleistyessä muun muassa kilpailevilla pilvipalveluntarjoajilla löytyy kullakin omat ratkaisunsa. Mainitsemisen arvoisia teknologioita ovat seuraavaksi esiteltävien RabbitMQ:n ja Apache Kafkan lisäksi vähintään Apachen AMQP -toteutus Apache ActiveMQ, Apache Pulsar, Microsoftin AMQP-toteutus Azure Service Bus ja Kafkaan perustuva Azure Event Hubs, Amazon SQS sekä Google Cloud Pub/Sub (Upsolver 2023). Jokaisella listatuista sovelluksista on omat vahvuusalueensa, ja ne soveltuvat paremmin tai huonommin erilaisiin käyttöympäristöihin, kuten pilveen.

3.4.1 RabbitMQ

RabbitMQ:n on avoimen lähdekoodin viestinvälittäjä, joka on laajalti käytössä mikropalveluiden välillä (More 2022). Sitä voidaan tietyissä tilanteissa pitää verrannollisena vaihtoehtona myöhemmin esiteltävälle Apache Kafkalle, vaikka niiden toimintaperiaate on hyvin erilainen. RabbitMQ perustuu AMQP-protokollaan (Advanced Message Queuing Protocol), joka on avoimen standardin väliohjelmisto viesteille (engl. message middleware) ja joka mahdollistaa tehokkaan ja luotettavan viestinvälityksen hajautettujen järjestelmien välillä (Fernandes, ym. 2013).

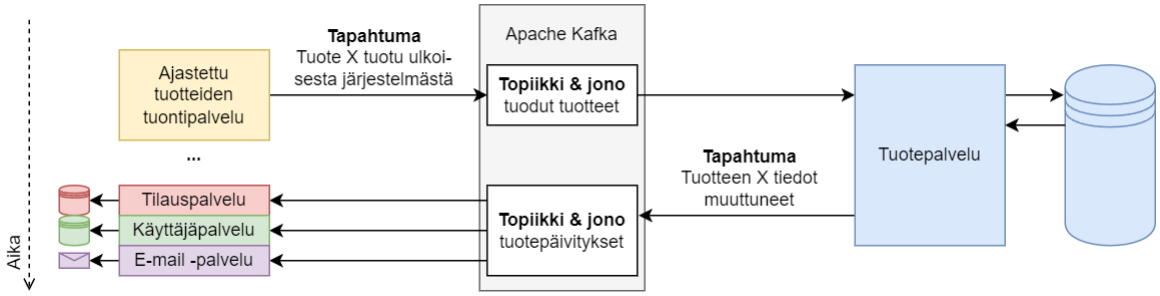


Kuvio 7. Esimerkkikaavio RabbitMQ:ta käyttävästä kommunikaatiosta

RabbitMQ lupaa toimittaa viestit prosessoitavaksi joko *enintään kerran* tai *vähintään kerran* käyttäen apuna viestien kuittauksia (VMware Inc. 2023). Näitä tutkielman aiheeseen olennaisesti liittyviä prosessointitakuita käsitellään lisää seuraavassa luvussa 3.5.

3.4.2 Apache Kafka

Tämän tutkielman tutkimusosuuden toteutusalustaksi valitun Apache Kafkan määritellään olevan virtojen prosessointialusta (engl. stream-processing platform) (Garg 2013). Se on toinen esimerkki avoimen lähdekoodin hajautetusta julkaise—tilaa -järjestelmästä ja se on kehitetty alun perin LinkedIn -sosiaalisen median syötteen käsittelyyn (Momtselidze ja Tsitsagi 2015).



Kuvio 8. Tapahtumalähtöinen esimerkki kommunikaatiosta käyttäen Kafkaa

Kafkan korostaa virtausanalogiaa, jossa tapahtumat syntyvät reaaliajassa eri lähteistä, kuten tietokannoista, sensoreista, mobiililaitteista, pilvipalveluista tai muista ohjelmistoista muodostaen jatkuvan virran. Näitä tapahtumia tallennetaan, manipuloidaan tai prosessoidaan matkan varrella ja niihin reagoidaan tai niitä reititetään tarpeen mukaan. Ytimekkäästi

ilmaistuna tapahtumavirrat varmistavat, että oikea tieto on oikeassa paikassa oikeaan aikaan. (Apache Software Foundation 2023)

Apache Kafkaa ja sen käyttötarkoituksia käsitellään yksityiskohtaisemmin sille omistetussa luvussa 5.

3.5 Viesti vai tapahtuma?

Viesti (engl, message*)* on lähettävän osapuolen lähettämää dataa, jolle on päätelty tietty päämäärä tai päämäärät. Tapahtuma (engl. event) liikkuu käytännössä samaan tapaan viestinä, mutta sen laukaisee jokin selkeästi järjestelmässä tapahtunut toiminto (Bonér;Farley, ym. 2014). Viestilähtöisessä järjestelmässä tilaavat palvelut vastaanottavat saapuvia viestejä joko suoraan toisiltaan tai yleisesti kyseistä palvelua kiinnostavasta aiheesta toimien niiden mukaisesti. Tapahtumalähtöisessä järjestelmässä tilaavat palvelut sen sijaan kuuntelevat haluamiaan aiheita tietystä tapahtumalähteestä toimien tai ei-toimien sinne julkaistun tapahtuman mukaan (Bonér;Farley, ym., Glossary - The Reactive Manifesto 2014). Tätä eroa pyritään havainnollistamaan verrattaessa RabbitMQ:ta ja Kafkaa kuvioissa Kuvio 7 ja Kuvio 8.

Viestilähtöisyyttä ja tapahtumalähtöisyyttä käytetään usein synonyymeinä. Tästä syystä esitelty jaottelu on osittain keinotekoinen ja Apache Kafka onkin tunnetusti ainoa viestinvälitysteknologia, jonka määritellään käsittelevän nimenomaan tapahtumia. Tutkimusaiheen ymmärtämisen kannalta on kuitenkin hyvä pystä erottamaan nämä kaksi termiä, viesti ja tapahtuma, toisistaan. Tämän tutkielman Kafkaa käsittelevissä osissa pyritään käyttämään termiä *tapahtuma* Kafkan tapahtumaentiteeteistä ja termiä *tapahtumaviesti* tai lyhyemmin *viesti* kyseisen entiteetin sisältävistä viesteistä.

4 Viestin prosessointitakuu

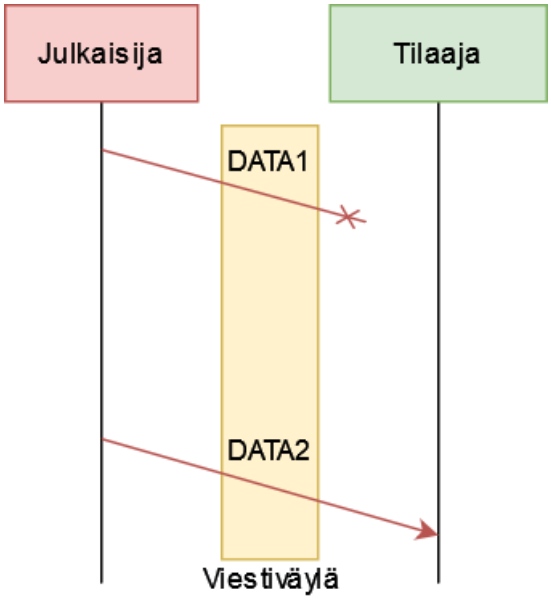
Viestilähtöisen kommunikaation luotettavuuden kannalta on keskeistä, että käytettävä viestinvälitysteknologia pystyy antamaan takuun sekä viestin tuottamisesta viestinvälittäjälle, että viestin lukemisesta viestivälittäjältä prosessoitavaksi. Tätä kutsutaan viestin prosessointitakuuksi. Yleisesti käytössä olevat prosessointitakuun semantiikat ovat (Bryant 2019):

- enintään kerran (engl. at-most-once),
- vähintään kerran (engl. at-least-once) ja
- täsmälleen kerran (engl. exactly-once).

Neljä yleisintä virhetilannetta, joita on syytä miettiä prosessointitakuuta valittaessa ovat virhetilanne viestin julkaisuvaiheessa, virhetilanne viestin kulutusvaiheessa (hakiessa viestejä viestinvälittäjältä), virhetilanne viestinvälittäjässä ja virhetilanne viestiä prosessoidessa (Bryant 2019).

4.1 Enintään kerran

Enintään kerran -takuu tarkoittaa, että lähetetty viesti luvataan toimittaa prosessoitavaksi viestiväylän tilaajalle korkeintaan yhden kerran (0–1 kertaa). Kun julkaisija on saanut omasta näkökulmastaan viestin lähetettyä viestinvälittäjälle, ei viestin liikkeistä sen jälkeen välitetä (McKee 2019). Näin ollen mistä tahansa lähetyksen jälkeisestä virhetilanteesta johtuen viesti ei välttämättä tule perille tai viestien järjestys voi olla viiveen takia muuttunut. *Enintään kerran* -semantiikka on suorituskyvyltään vähiten kuormittava vaihtoehto, koska viestejä ei tarvitse seurata eikä yrittää lähettää uudelleen.

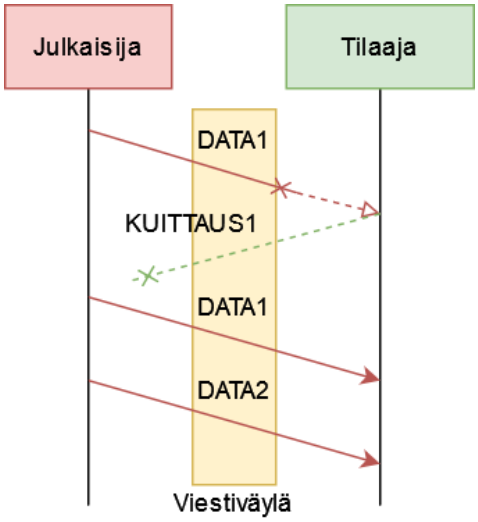


Kuvio 9. *Enintään kerran* yleisimmässä virhetilanteessaan, josta huolimatta dataa ei lähetetä uudelleen

4.2 Vähintään kerran

Vähintään kerran -takuu lupaa, että viesti tulee prosessoiduksi yhden tai useamman kerran (1—n kertaa). Viestiä yritetään lähettää niin kauan, kunnes varmistetaan, että se on saapunut tilaajien prosessoitavaksi vähintään kerran. Tämä voi johtaa viestin prosessointiin kahdesti tai useammin virhetilanteen sattuessa missä tahansa vaiheessa. *Vähintään kerran* takaa usein myös viestin järjestyksen säilymisen. (McKee 2019)

Huomioitavaa: *vähintään kerran* -semantiikan avulla on mahdollista saavuttaa järjestelmätasolla *täsmälleen kerran* -käsittely, kunhan varmistutaan siitä, ettei kahden identtisen viestin vastaanottaminen vaikuta lopulliseen tallennettuun dataan. Se ei silti tarkoita *täsmälleen kerran* -takuun toteutumista viestin välittäjäsovelluksen näkökulmasta.



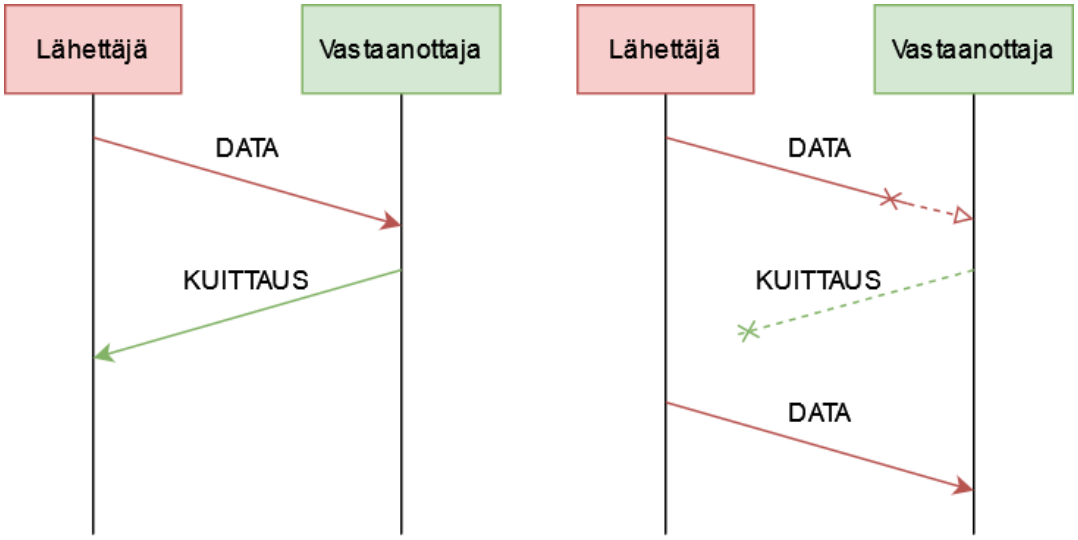
Kuvio 10. *Vähintään kerran* kahdessa yleisimmistä virhetilanteistaan, joista kuittauksen jääminen matkalle johtaa virheelliseen tilaan järjestelmässä

4.3 Täsmälleen kerran

Täsmälleen kerran on takuista halutuin ja vaativin. Se tarkoittaa sitä, että viesti tulee prosessoiduksi kerran ja vain kerran (1 kertaa) (McKee 2019). Virhetilanteen sattuessa missä tahansa edellä kuvatussa tilanteessa, pystyy tällaisen lupauksen toteuttava järjestelmä toipumaan siten, että viesti tulee varmasti prosessoiduksi tasan yhden kerran. T*äsmälleen kerran* -takuu on semantiikoista suorituskykyä vaativin, johtuen viestien seurannasta ja useammista tarkastuksista semantiikan toteutumiseksi.

4.3.1 Kuljetustakuu ja prosessointitakuu

Kuten Tyler Treat (2015) on todennut, *täsmälleen kerran* on mahdotonta saavuttaa dataa kuljettaessa. Hän viittaa kirjoituksessaan tarkemmin ”kahden kenraalin ongelmaan”.



Kuvio 11. Kahden kenraalin ongelma tietoliikenteessä. Vasemmalla onnistunut tiedonsiirto, oikealla joko data tai kuittaus jää matkalle.

Kuvio 11 havainnollistaa yksinkertaistetusti kaksi eri datansiirtotapahtumaa luotettavan protokollan avulla, joka kuittaa kunkin erän dataa vastaanotetuksi. Vasemmanpuoleisessa esimerkissä tapahtuma kuittauksineen on onnistunut, kun taas oikeanpuolimmaisessa lähettävä taho ei ole saanut vastaanottajalta kuittausta vastaanotetusta datasta. Lähettäjän näkökulmasta ei siten voida tietää, onko itse data jäänyt matkalle vai onko ainoastaan kuittaus kadonnut. Sama data on lähetettävä uudelleen kyseisen luotettavan protokollan mukaisesti, jolloin se toteuttaa ’vain’ *vähintään kerran* -kuljetustakuun.

Kahden kenraalin ongelman vuoksi *täsmälleen kerran* -semantiikan mahdollistavien viestinvälittäjien kohdalla onkin tarkoituksenmukaista puhua kuljetustakuun (engl. delivery guarantee) sijaan prosessointitakuusta (engl. processing guarantee). Tällä tarkoitetaan, että kyseinen semantiikka tapahtuu sovellustasolla (engl. application layer) kuljetustason (engl. transport layer) sijaan (Treat 2017). Tässä tapauksessa kaikki sen toteutumiseen vaadittavat toiminnot tehdään sovellustasolla, vaikka lupaus *täsmälleen kerran* -prosessoinnista sisältääkin myös viestin kuljetuksen määränpäähänsä. Käytännössä kuljetus voidaan toteuttaa millä tahansa kuljetustason protokollalla.

Täsmälleen kerran -prosessointia ei saavuteta ilmaiseksi, vaan se vaatii paljon sekä kehitettävältä ohjelmistolta, että valitulta viestinvälittäjäsovellukselta (Treat 2017). Jälkimmäi-

seen pureudumme seuraavaksi perehtyen syvällisemmin tähän tutkimukseen valittuun Apache Kafkaan, joka lupaa kaikki vaadittavat työkalut *täsmälleen kerran* -prosessoinnin onnistumiseksi.

5 Apache Kafka

Apachen dokumentaation (2023) mukaan Kafka määritellään tapahtumien virtausalustaksi (engl. event streaming platform), mutta siitä voidaan käyttää myös aiemmin esiteltyä termiä virtojen prosessointialusta (Garg 2013). Se on kehitetty alkujaan LinkedIn-palvelun taustalle käyttäen ohjelmointikielinään Scalaa ja Javaa (Johansson 2020). Nykyään se toimii myös osana Confluent Stream Platform -alustaa. Kafkan luvataan toteuttavan hajautetusti, skaalautuvasti, mukautuvasti, virhesietoisesti ja turvallisesti kolme päätehtäväänsä (Apache Software Foundation 2023):

1. julkaista ja tilata tapahtumavirtoja,
2. tallentaa tapahtumia kestävästi ja luotettavasti määrätyn ajan sekä
3. prosessoida ne heti niiden tapahtuessa tai jälkikäteen.

Vaikka keskitymme tässä tutkielmassa mikropalvelujen viestintään, on syytä mainita, että Kafka loistaa parhaiten reaaliaikaisessa prosessoinnissa datavirtojen ollessa massiivisia. Esimerkkejä kyseisistä käyttötarkoituksista ovat käyttäjien analysointi, ohjelmiston suorituskyvyn mittaus, lokien seuranta sekä esineiden internet (engl. Internet of Things, IOT) (Garg 2013). Tutkielman empiirisen osan tulokset ovat harkinnanvaraisesti sovellettavissa myös kyseisiin käyttötarkoituksiin.

Garg (2013) kertoo kirjassaan, että Kafkaa suunniteltaessa on keskitytty erityisesti seuraaviin tunnusmerkkeihin:

- **pysyvyys** (engl. persistence): suuren määrän dataa käsittely häviöttä,
- **siirtonopeus**: jopa miljoonien viestien käsittely sekunnissa,
- **jakautuvuus**: sekä Kafka-palvelimien että viestien lukemisen jakaantuminen usealle laitteelle viestien järjestyksen muuttumatta,
- **yhteensopivuus**: asiakasohjelmat ovat helposti toteutettavissa eri ohjelmointikielillä sekä
- **reaaliaikaisuus**: viestin on oltava saatavilla sen kuluttajalle välittömästi tuottamisen jälkeen.

5.1 Arkkitehtuuri

Kafkan taustalla vaikuttaa tuttu asiakas–palvelin -malli ja kuljetustasolla luotettava TCP-protokolla. Kafkan julkaisualustat eivät poikkea juurikaan muista nykyaikaisista sovelluksista – Kafka-klusteri tai sen osia voidaan julkaista niin fyysiselle laitteelle, virtuaalikoneelle, konttiteknologian päälle kuin avaimet käteen -periaatteella toimitettuna pilvipalveluunkin. Asiakasohjelmia, eli kuluttajia ja tuottajia, voidaan kehittää millä ohjelmointikielellä tahansa. (Apache Software Foundation 2023)

Teknisesti tapahtumaviestit liikkuvat Kafkassa tavumatriiseina ja niiden data voi olla käytännössä mitä tahansa, pakollisia kenttiä niille ovat ainoastaan viestin avain ja arvo. Näiden lisäksi viestin vapaaehtoiset kentät ovat aikaleima ja otsikkokentät (engl. headers). (Johansson 2020)

5.1.1 Välittäjä ja topiikit

Kafka perustuu virtaaviin tapahtumiin, siihen että jotain on *tapahtunut* lähdejärjestelmässä. Kun tällainen virta päätyy Kafkan palvelimelle, jota kutsutaan välittäjäksi (engl. broker), tallentaa se virran tapahtumat kullekin määrätyn topiikin (engl. topic) tapahtumalokeille (Dhanushka 2021). Kullakin topiikilla voi olla n-määrä sille tapahtumia tuottavia tuottajia (engl. producer) ja n-määrä siltä lukevia kuluttajia (engl. consumer) (Johansson 2020).

Topiikki voidaan ajatella tapahtuman kategoriana tai tapahtumasyötteen selkokielisenä nimenä (Johansson 2020). Havainnollistava yksinkertaista järjestelmää simuloiva esimerkki topiikkien nimeämisestä löytyy kuviosta 8 (Kuvio 8). Yhden tai useamman välittäjän muodostamaa kokonaisuutta kutsutaan klusteriksi. Klusterin välittäjät voivat sijaita fyysisesti eri paikoissa. Jotta topiikkien myöhemmin esiteltävä ositus pääsee oikeuksiinsa, suositellaan Kafka-klusterin koostuvan useasta, yleensä vähintään kolmesta, välittäjäinstanssista (Johansson 2020). Välittäjien välistä kommunikaatiota hoitaa ja ylläpitää klusterin sisällä ajettava, erillinen Zookeeper-ohjelma, josta siitäkin on suositeltavaa ajaa useampaa instanssia (Vinka 2018).

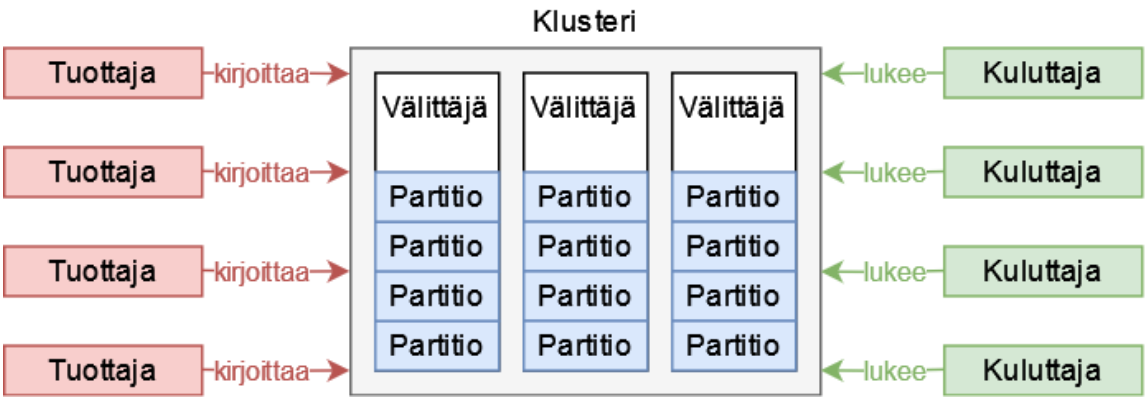
5.1.2 Tuottajat ja kuluttajat

Tuottaja on Kafkan vastine julkaise—tilaa -mallin julkaisevalle asiakasohjelmalle. Se voi olla mikä tahansa tapahtumanlähde, johon on toteutettu mahdollisuus työntää (engl. push) tapahtumia Kafka-välittäjän tapahtumajonoihin (Apache Software Foundation 2023). Työntäminen tapahtuu yhden tai useamman tapahtuman erissä (engl. batch), joita tuottaja voi lähettää useita kerralla. Erän koko määrittyy joko tavumäärän tai sen kasaamiseen kuluneen ajan mukaan (Conduktor Inc 2023).

Kuluttaja on tapatumia tilaava asiakasohjelma, joka reaaliajassa hakee (engl. pull) seuraamansa topiikin tapahtumia ja prosessoi ne (Apache Software Foundation 2023). Jos kuluttaja-asiakasohjelmasta pyörii useampia instansseja horisontaalisen skaalauksen seurauksena, voidaan niistä muodostaa kuluttajaryhmä. Tämä mahdollistaa samaan ryhmään asetettujen kuluttajien prosessoivan kunkin tapahtuman vain kerran (Sookocheff 2015).

Kafka ei useiden julkaise—tilaa -järjestelmien tapaan vie viestejä kuluttajasovelluksille, vaan kuluttajien vastuulla on itse hakea ne ja huolehtia etenemisestään topiikin tapahtumalokeilla (Dhanushka 2021). Tapahtumia ei myöskään poisteta lokilta heti niiden kuluttamisen jälkeen, mikä erottaa Kafkan tehokkaasti muista järjestelmistä ja tuo sille erilaisia käyttötarkoituksia myös datan säilömisen näkökulmasta. Edellä mainittujen ominaisuuksien ansiosta voidaan Kafka-välittäjään myös millä tahansa ajanhetkellä liittää uusi kuluttaja tai kuluttajaryhmä, jolloin se saa luettavakseen kaikki topiikkiin siihen mennessä tallentuneet tapahtumat. Lisäksi kuluttajan virhetilanteista selvitään helposti, kun kuluttamattomat tapahtumat odottavat lokilla sen toipumista (Dhanushka 2021).

Tallennustilan säästämiseksi tapahtumien säilyttämisajalle on kuitenkin määriteltävä jokin säilytysaika, jonka jälkeen tapahtumat poistetaan lokeilta (Wu;Shang ja Wolter 2019). Oletuksena tämä aika on yksi viikko. Säilytettyjen tapahtumien suurenkaan määrän ei kuitenkaan väitetä vaikuttavan Kafkan suoritustehoon (Apache Software Foundation 2023).

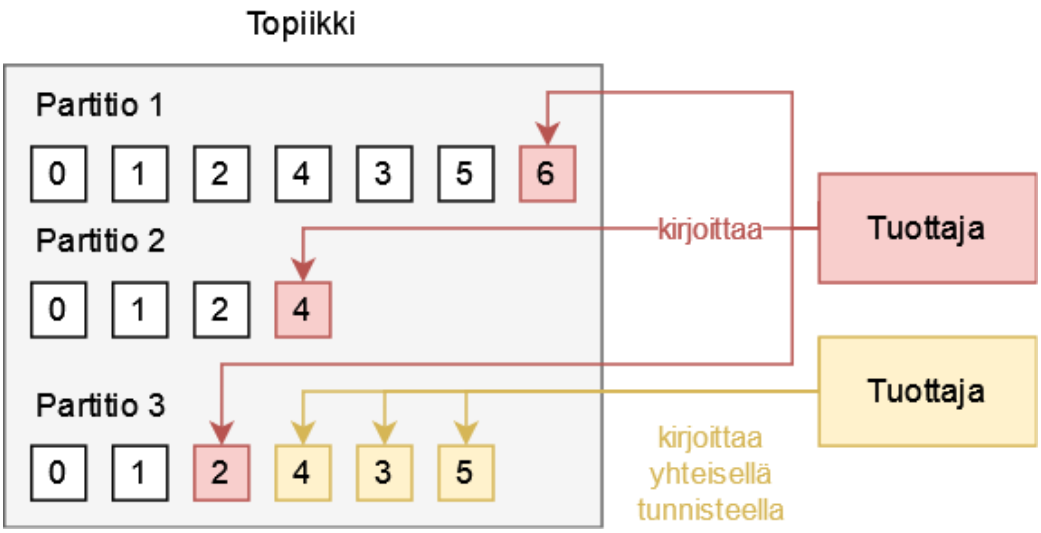


Kuvio 12. Kafka-klusteri (Johansson 2020)

5.1.3 Partitiot ja tiedon hajautus

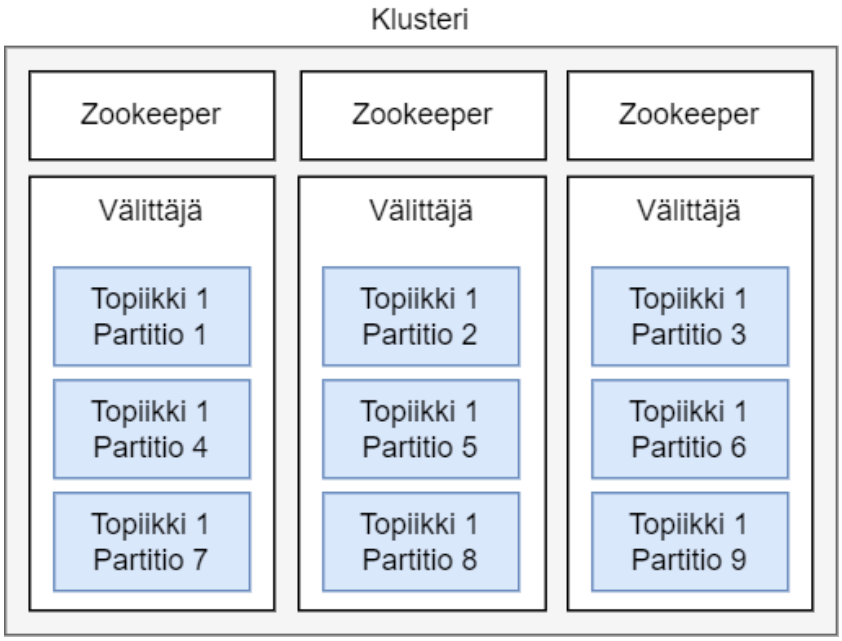
Jotta Kafka saavuttaisi kaikki luvatut päätehtävänsä, on topiikeihin tallennettava data syytä hajauttaa vielä osiin. Kafkan pienintä tallennusyksikköä kutsutaan partitioksi (engl. partition) ja se sisältää osan yhden ja vain yhden topiikin tapahtumista. Kunkin topiikin kukin partitio sisältää tapahtumalokin, jonka viimeiseksi tuottajilta saapuvat tapahtumat lisätään. Kukin tapahtuma voi päätyä vain yhdelle partitiolle, jolloin sille määräytyy partitiokohtainen, juokseva (0…) offsetarvo. Tämä arvo kertoo tapahtuman etäisyyden tapahtumalokin ensimmäisestä tapahtumasta. Tämä arvo on muuttumaton, vaikka aiempia tapahtumia poistetaan. (Dhanushka 2021)

Kuluttajat lukevat kuluttajaryhmälleen lukematta olevat tapahtumat tilaamiensa topiikkien partitioilta FIFO-periaatteella (First in, First out), eli tarkalleen siinä järjestyksessä missä ne on Kafkalle tuottajilta saapuneet. Onnistuneen lukuoperaation jälkeen kuluttaja tallentaa kuluttajaryhmä- ja partitiokohtaisesti uuden etäisyyden alkutilanteesta, eli viimeisimmäksi prosessoidun tapahtuman offsetarvon. Sen perusteella tiedetään tarkalleen, mihin asti kyseinen kuluttajaryhmä on kyseisen partition tapahtumajonoa lukenut. (Baey 2016)



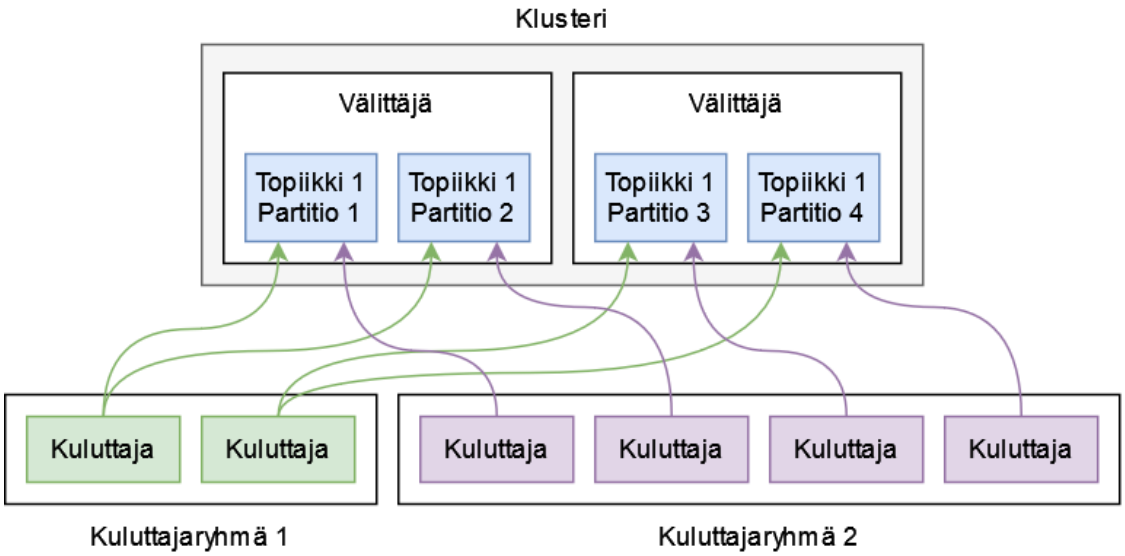
Kuvio 13. Topiikin anatomia, uudemmat tapahtumat oikealla (Sookocheff 2015)

Kafka-toteutusta suunnitellessa on hyvin tärkeää huomata, että tapahtumien järjestys on taattu vain partitio-, ei topiikkikohtaisesti (Dhanushka 2021). Tästä syystä tapahtumien, joiden on välttämätöntä tulla prosessoiduksi muuttumattomassa järjestyksessä, on syytä määrätä menevän yhdelle ja samalle partitiolle. Tämä tapahtuu määrittelemällä niille yhteinen tunniste (engl. partition key). Tällaisesta käyttötapauksesta hyvä esimerkki on käyttäjäseuranta verkkosivulla, jolloin kaikkien käyttäjien toiminnoista syntyvät tapahtumat päätyisivät yhteiseen ’käyttäjätapahtumat’ -topiikkiin. Kunkin tapahtuman tunnisteena on tällöin syytä käyttää käyttäjän yksilöivää tunnusta, jotta kuhunkin käyttäjään kohdistuvat tapahtumat tulevat varmasti tallennettua ja prosessoitua alkuperäisessä järjestyksessään (Johansson 2020). Huono tunniste on sellainen, joka ohjaa ison osan tapahtumista samaan partitioon (Dhanushka 2021). Tällöin partitioinnin, eli osituksen tuoma hyöty menetetään.



Kuvio 14. Yksittäisen topiikin partitiot, partitioiden määrä yhdeksän (Sookocheff 2015)

Kunkin topiikin partitiot jakaantuvat tasaisesti klusterin eri välittäjille sen mukaan, moneenko partitioon topiikki on määritelty jakaantuvan (Sczip 2021). Partitioiden määrä on hyvä miettiä tarkkaan ohjelmiston suunnitteluvaiheessa, sillä niiden lisääminen säilyttäen aiemman, yksilöivän tunnuksen avulla ylläpidetyn järjestyksen saattaa olla haastavaa jälkikäteen (Conduktor Inc 2023). Partitioiden suurestakaan määrästä ei kuitenkaan ole hyötyä, jos kyseisen topiikin kuluttajaryhmissä ei ole yhtä paljon kuluttajia – yhden kuluttajan vastuulle jää useampi partitio. Vastaavasti kuluttajien suuresta määrästä per ryhmä ei ole hyötyä, jos topiikissa ei ole yhtä montaa partitiota (Brebner 2023).

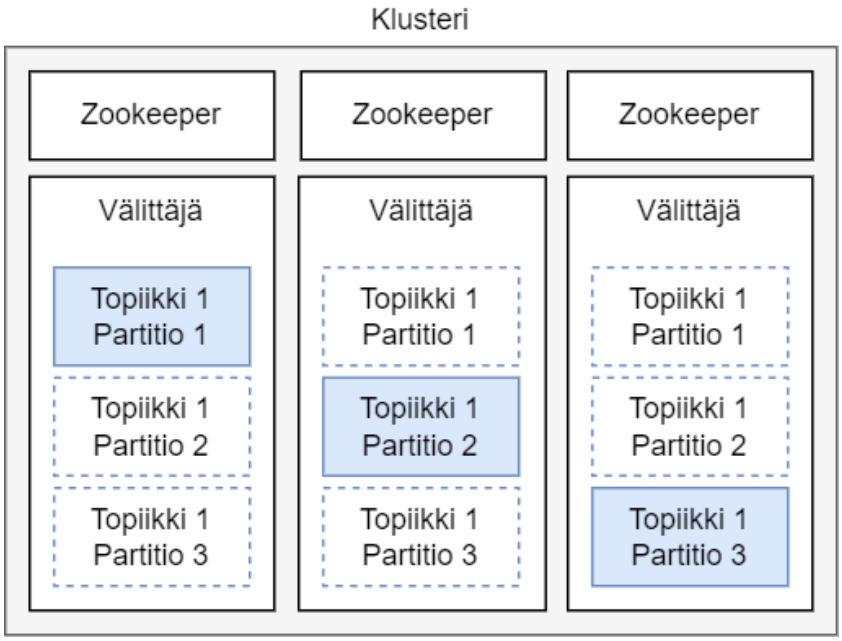


Kuvio 15. Esimerkki yhden topiikin lukuvastuiden jakautumisesta kuluttajaryhmittäin (Sookocheff 2015)

Osituksen tarkoitus on ennen kaikkea mahdollistaa horisontaalinen skaalaus, jolloin yhden välittäjän resurssit ja tiedonsiirtonopeus eivät ole rajana topiikkia käsitellessä. Lisäksi osituksella mahdollistetaan useamman kuluttajan yhtäaikaiset lukuoperaatiot topiikeista sekä kunkin kuluttajaryhmään kuuluvan kuluttajan vastuualueen kohdistaminen vain yhteen, tiettyyn partitioon (Kuvio 15). Nämä vaikutukset mahdollistavat ison osan Kafkan nopeudesta. (Dhanushka 2021)

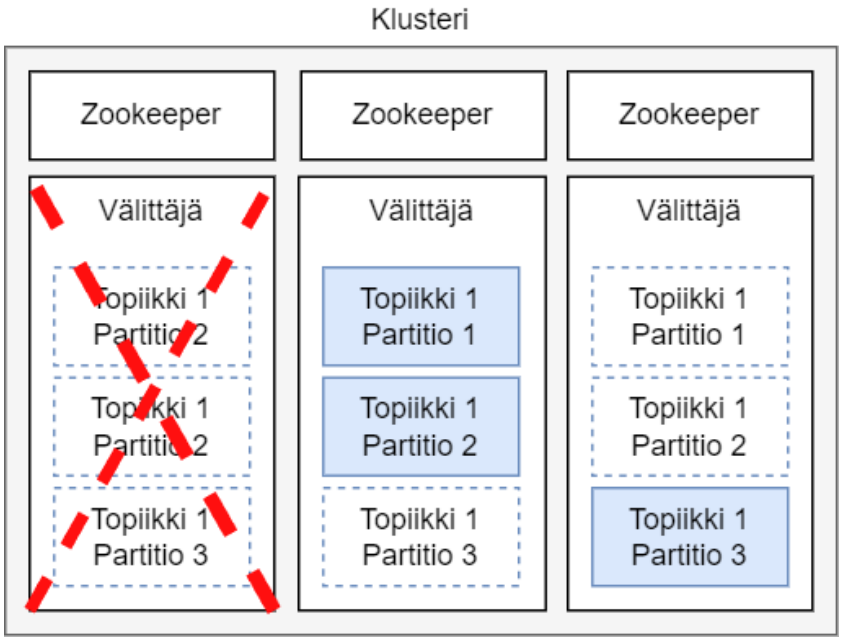
5.1.4 Replikointi

Osituksen ollessa avainasemassa Kafkan reaaliaikaisuudessa ja jakautuvuudessa muodostaa replikointi perustan sen pysyvyydelle ja luotettavuudelle. Jokaisesta partitiosta luodaan ja ylläpidetään kopioita, joiden määrä perustuu topiikin luonnin yhteydessä määriteltyyn replikaatiokertoimeen (engl. replication factor). Replikaatiokertoimeksi suositellaan vähintään kahta ja enintään neljää (Conduktor Inc 2023). Hyödyn saavuttaakseen replikat jakaantuvat eri välittäjille, eikä niitä täten ole mahdollista määritellä muodostuvan enempää, kuin klusterissa on välittäjiä.



Kuvio 16. Yksittäisen topiikin replikat. Partitioiden määrä kolme, replikointikerroin kolme (Sookocheff 2015).

Jokaiselle replikoidulle partitiolle on määritelty aina johtava partitio (engl. leader). Kun kyseinen partitio vastaanottaa tapahtuman, tallentuu se ensimmäiseksi sen tapahtumalokille. Onnistuneen tallentumisen jälkeen johtavan partition vastuulla on lähettää tapahtuma edelleen replikoille niiden lokeille lisättäväksi. Lukuoperaatiot tapahtuvat lähtökohtaisesti aina johtavan partition lokilta. Jos virhetilanteen takia sen sisältävä välittäjä kuitenkin kaatuu tai ei muuten ole saavutettavissa, nostetaan jokin replikoista uudeksi johtavaksi partitioksi. Partitioiden ja replikoiden jakaantuessa tasaisesti eri välittäjille, mahdollistavat ne todella luotettavan kokonaisuuden. (Sookocheff 2015)



Kuvio 17. Yksi välittäjistä on saavuttamattomissa, Partitio 1 on saanut uuden johtajan

5.2 Vähintään kerran -semantiikan ongelmatilanteet Kafkassa

Kafkan tarjoama *vähintään kerran* -semantiikan mukainen prosessointitakuu ei täytä *täsmälleen kerran* -prosessointitakuuta seuraavissa virhetilanteissa:

1. Tuottaja jää ilman kuittausta tapahtuman saapumisesta välittäjälle, jolloin se lähetetään uudelleen (Kuvio 10).
2. Kuluttajainstanssi kaatuu ennen kuin se on saanut tallennettua uuden offsetarvon, jolloin tilalle syntynyt kuluttajainstanssi prosessoi tapahtuman uudelleen.
3. Kuluttava tai tuottava palvelu kaatuu, tai menettää väliaikaisesti yhteyden ulkopuolelleen johtaen uuteen instanssiin. Jos vanha instanssi herää uudestaan eloon, se jää niin kutsutuksi zombie-instanssiksi, joka kuluttaa tai tuottaa samoja tapahtumia kuin uusi. (Mehta ja Gustafson 2017)

Seuraavassa alaluvussa esitellään, kuinka Kafkan täsmälleen kerran -prosessointitakuu ratkaisee yllä esitellyt ongelmat.

5.3 Täsmälleen kerran -semantiikka ja Kafka

Täsmälleen kerran -prosessoinnin toteutuminen vaatii omat toimensa sekä kuluttajilta että tuottajilta, mutta Kafkan luvataan antavan siihen kaikki tarvittavat työkalut (Narkhede 2017). Keskitymme tämän tutkielman tutkimusosuudessa arvioimaan kyseisen semantiikan suorituskykyä käyttäen Kafkaa viestinvälitysjärjestelmänä. Siksi on hyvä perehtyä, miten se saavutetaan.

5.3.1 Tuottajan idempotenssi

Matematiikasta tuttu termi idempotenssi tarkoittaa, että tietyn operaation tapahtuminen useampaan kertaan ei muuta operaation alkuperäistä lopputulosta siihen nähden, että se tapahtuisi vain kerran (Kala ja Korbelář 2018).

Täsmälleen kerran -semantiikan yhteydessä Kafka-tuottajan lähetysoperaatiosta tehdään idempotentti – vaikka tapahtuman lähetys virhetilanteessa tapahtuisi useampaan kertaan, päätyy se välittäjän lokille vain kerran (Narkhede 2017). Tämä on mahdollistettu antamalla jokaiselle tuottajainstanssille uniikki tunniste ja sekvenssinumeroimalla jokainen tapahtumaerä. Nämä tiedot yhdistämällä välittäjän on mahdollista ehkäistä duplikaattierien tallentuminen hylkäämällä erät, joiden yhdistetty tunniste on identtinen tai pienempi (Conduktor Inc 2023). Jotta tämä tieto ei olisi vain muistinvaraista eikä häviäisi myöskään johtavan partition virhetilanteessa, on kyseinen tunniste itsessään myös tallennettu replikoidulle tapahtumalokille (Narkhede 2017).

Idempotentisuus ratkaisee tuottajan ongelmia sekä puuttuvan kuittauksen että zombie-instanssin tilanteissa.

5.3.2 Transaktiot

Toinen *täsmälleen kerran* -semantiikan mahdollistava tekijä on Kafkan transaktio-ohjelmointirajapinta (engl, transactions API). Sen avulla voidaan tuottaa tapahtumia yhdelle tai useammalle topiikille ja/tai partitiolle siten, että joko kaikki erän tapahtumat saapuvat kulutettavaksi tai ei yksikään. Samalla ratkaistaan luvussa 5.2 esitelty kuluttajan ongelma-

tilanne ylläpitämällä myös kuluttajan offsetarvoa omassa topiikissaan ja yhdistämällä tämän asetus atomisesti yhteen ja samaan transaktioon muiden erässä tallennettavien tapahtumien kanssa. (Narkhede 2017)

Koodiesimerkki transaktiorajapinnan käytöstä (Narkhede 2017):

```
producer.initTransactions();
try {
    producer.beginTransaction();
    producer.send(record1);
    producer.send(record2);
    producer.commitTransaction();
} catch(ProducerFencedException e) {
    producer.close();
} catch(KafkaException e) {
    producer.abortTransaction();
}
```

Confluent Inc (2023) määrittelee transaktioille yhteensä kolme tehtävää:

1. mahdollistaa tapahtumien lähettämisen erissä atomisesti eri topiikeihin,
2. ehkäistä duplikaatteja ja
3. mahdollistaa Kafka Streams -rajapinnan atomisuus.

Tässä kohtaa on hyvä muistaa, että Kafkan transaktiot eivät pysty takaamaan *täsmälleen kerran* -semantiikan toteutumista kuljetustasolla paremmin kuin mikään muukaan järjestelmä. Kafka takaa *täsmälleen kerran* -prosessoinnin ainoastaan sovellustasolla Kafkan sisällä. Mikä tahansa kutsu ulkoiseen palveluun, tietokantaan tai datan tulostamiseen saattaa siis tuplaantua. (Confluent Inc 2023)

5.4 Kafka Streams

Kafka Streams -ohjelmointirajapinta on tarkoitettu helpottamaan Kafkan soveltamista sen yleisimmissä käyttötarkoituksissa, joissa samaa dataa käsitellään ja jalostetaan useampaan

kertaan. Se tarkoittaa käytännössä joukkoa peräkkäisiä tuottajia ja kuluttajia, jotka on yhdistetty toisiinsa topiikein. Tällöin kuluttaja käsittelee saapuvan tapahtuman, päättelee tai prosessoi sen perusteella jotain ja toimii itse tuottajana lähettämällä päättelyn tuloksen seuraavaan topiikkiin. (Narkhede 2017)

Aiemmissa luvuissa esitellyt seikat huomioon ottaen tällaisessa suljetussa ympäristössä *täsmälleen kerran* -prosessointi pääsee oikeuksiinsa parhaiten. Kafka Streams -rajapinnan kuvauksessa luvataankin, että se voidaan laittaa helposti päälle yhtä asetusta muuttamalla (Narkhede 2017). Tällöinkin täytyy huomioida, että täsmälleen kerran toteutuu ainoastaan tässä suljetussa järjestelmässä, ei sen ulkopuolella.

6 Empiirinen osuus

Tutkielman empiirisessä osuudessa kehitetään Kafka-tuottaja ja Kafka-kuluttaja, joiden välillä pyritään lähettämään massoittain tapahtumia Kafka-välittäjän kautta. Yksittäisen tapahtuman luomisen ja sen käsittelyn välinen aika otetaan talteen ja niistä kertyvän datan avulla kerätään tietoa eri mittarein. Sama testi toistetaan käyttäen kaikkia kolmea prosessointitakuuta vuorotellen.

Kokeessa käytetyt tuottaja ja vastaanottaja kehitetään Java-ohjelmointikielellä käyttäen pohjana Apachen virallisten testaus- ja esimerkkityökalujen ohjelmistokoodia[1]. Kokeen lähdekoodit ovat julkisesti saatavilla ja käytettävissä Apache-2.0 lisenssin mukaisesti[2].

6.1 Tutkimusympäristö

Kokeen isäntäkoneena toimii pöytätietokone suorittimenaan Intel Core i9-9900K, jossa on kahdeksan maksimissaan 4700 Mhz kellotaajuuteen yltävää ydintä ja 16 loogista prosessoria. Muisti on tyypiltään DDR4 SDRAM ja sitä on yhteensä 32 gibitavua. Sen nopeus on 3200 Mhz ja CAS-arvoltaan se on 14. Isäntäkoneen käyttöjärjestelmäksi on asennettu 64-bittinen Microsoft Windows 11 Pro, versio 10.0.22621. Varsinainen koe suoritetaan tämän sisällä toimivassa virtuaalisessa WSL (Windows subsystem for Linux) -ympäristössä, jossa käyttöjärjestelmänä toimii Ubuntu versio 20.04.6 LTS.

Kafka-klusteria suoritetaan Docker Compose -työkalulla, joka mahdollistaa useamman Docker-levykuvan ajamisen etukäteen määrätyin ympäristömuuttujin. Työkalun avulla rakennettaan kokonaisuus, joka käynnistää suositusten mukaisesti kolme Kafka-välittäjää ja kolme näiden välistä yhteyttä hoitavaa Zookeeper-sovellusta. Koetta varten luodaan topiikki, jonka partitiomäärä on yhdeksän ja replikaatiokerroin kolme. Docker Compose -määrittelytiedosto ja ohjeet topiikin luomiseen löytyvät kokeen repositoriosta.

[1] https://github.com/apache/kafka
[2] https://github.com/JkkaMr/kafkatester/

Isäntäkoneella pyörivän Docker-sovelluksen versio on v4.17.0. Kafka-välittäjän suorittamiseen käytetään Confluentin julkaisemaa *confluentinc/cp-kafka*-levykuvaa (versio 7.1.1) ja Zookeeperin suorittamiseen niin ikään Confluentin julkaisemaa *confluentinc/cp-zookeeper*-levykuvaa (versio 7.1.1). Kokeessa käytetty topiikki luotiin Apache Kafkan omalla komentorivityökalulla[3], Kafkan version ollessa tässä 3.4.0.

6.2 Käytetyt Kafka-asetukset

Tässä luvussa esitellään kokeessa käytettävien Kafka-sovellusten olennaiset asetukset.

6.2.1 Välittäjä

Kokeessa käytettyyn topiikkiin määritellään partitioiden määräksi yhdeksän (`9`) ja replikaatiokertoimeksi kolme (`3`).

6.2.2 Kuluttaja

Kuluttajan asetukset jätetään *enintään kerran* tai *vähintään kerran* -semantiikkoja testattaessa oletuksiin. *Täsmälleen kerran* -semantiikkaa testattaessa kuluttajalle on kuitenkin tarpeellista asettaa määritys

```
isolation.level=read_committed
```

6.2.3 Tuottaja

Tuottajalle asetetaan *enintään kerran* -semantiikkaa testattaessa seuraavat määritykset

```
acks=0
retries=0
max.in.flight.requests.per.connection=2147483647
enable.idempotence=false
```

[3] https://github.com/apache/kafka/tree/3.4/bin

vähintään kerran -semantiikkaa testattaessa asetetaan määritykset

```
acks=all
min.insync.replicas=1
retries=2147483647
max.in.flight.requests.per.connection=5
enable.idempotence=true
```

ja *täsmälleen kerran* -semantiikkaa testattaessa määritykset

```
acks=all
min.insync.replicas=2
retries=2147483647
max.in.flight.requests.per.connection=5
enable.idempotence=true
transactional.id=test-producer
```

Tuottajaohjelmaa ajettaessa sille annettaan parametrina kuhunkin kokeeseen valittu tapahtumaviestimäärä sekä kunkin ajon viestikoko tavuina. *Täsmälleen kerran* -kokeessa annetaan lisäksi kulloinkin testattava transaktioaika. Viestien pakkaus ei ole käytössä. Tapahtumien eräkäsittely (engl. batching) ei ole käytössä siihen vaikuttavan tuottajan asetuksen `linger.ms` ollessa oletuksena `0` (Apache Software Foundation 2023).

6.2.4 Asetusten selitteet

Kuluttajan asetus `isolation.level` määrittää kuinka transaktioiden avulla kirjoitetut tapahtumat luetaan. Kun arvoksi on määritetty `read_committed`, luetaan kaikkien tapahtumien sijaan vain transaktioiden avulla varmuudella kommitoidut (engl. committed) tapahtumat (Apache Software Foundation 2023). Tämän vuoksi kyseinen arvo on tärkeä määrittää testattaessa *täsmälleen kerran* -takuuta.

Tuottajan asetus `acks` määrittää kuittausten määrän, jota tuottaja odottaa johtavalta partiolta ennen kuin tapahtumaviestin lähetys voidaan tulkita onnistuneeksi. Kun se on asetettu nollaksi (`0`), tuottaja ei odota kuittausta lainkaan ja tapahtuman lähetys tulkitaan välit-

tömästi onnistuneeksi (Apache Software Foundation 2023) vastaten *enintään kerran* -semantiikkaa. Kun arvo on asetettu ykköseksi (`1`), odotetaan kuittaus johtavalta partitiolta, mutta ei tämän replikoilta. Yleisestä harhaluulosta huolimatta tämä ei silti riitä takaamaan tapahtuman päätymistä kuluttajalle *vähintään kerran* johtuen skenaariosta, jossa johtava partitio ehtii kuitata tapahtuman vastaanotetuksi ja hajoaa välittömästi tämän jälkeen, ennen replikointia (Apache Software Foundation 2023). Tämän takia sekä *vähintään kerran* että *täsmälleen kerran* vaativat arvon `acks=all`, jolloin odotetaan johtavan partition lisäksi kuittaus vähintään yhdeltä replikalta.

Tuottajan `acks` -asetukseen liittyy vahvasti myös asetukset `min.insync.replicas` ja `retries`. Ensimmäinen määrittää, monenko replikan kuittausta odotetaan (Apache Software Foundation 2023). Täten replikointikertoimen ollessa kolme, riittää *vähintään kerran* -toteutukseen yhden (`1`) ja *täsmälleen kerran* -toteutukseen kaikkien kahden (`2`) replikan kuittaus. `retries` määrittää, kuinka monta kertaa epäonnistunutta lähetystä yritetään uudestaan (Apache Software Foundation 2023). Tästä syystä se on asetettu maksimiinsa *vähintään kerran* ja *täsmälleen kerran* -testeissä, sen oletusarvon ollessa nolla (`0`).

Tuottajan asetus `max.in.flight.requests.per.connection` määrittää, kuinka monta yhtäaikaista erälähetystä tuottajalle sallitaan. *Enintään kerran* -takuussa arvo voi olla suurempi, mutta *vähintään kerran* -prosessoinnin kohdalla on oltava tarkkana. Mikäli kyseinen asetus on määritelty suuremmaksi kuin yksi (`1`) ja uudelleenyritykset (`retries`) ovat käytössä, vallitsee uudelleenyrityksen tapahtuessa suuri riski tapahtumien järjestyksen muuttumiselle. Tämän vuoksi sekä *täsmälleen kerran* että myös *vähintään kerran* -prosessointien yhteydessä on hyödyllistä käyttää Kafkan idempotenssiä, joka mahdollistaa viiden (`5`) yhtäaikaisen lähetyksen säilyttäen niiden alkuperäisen järjestyksen. Tämä on usein jätetty huomiotta testattaessa *vähintään kerran* -takuuta. Idempotenssi asetetaan käyttöön asetuksella `enable.idempotence=true`. Lisäksi *täsmälleen kerran* -prosessoinnin käyttämiä transaktioita varten asetetaan niiden vaatima yksilöllinen tunniste `transactional.id`. (Apache Software Foundation 2023)

Tapahtumien eritykseen oleellisesti liittyvä asetus `linger.ms` tarkoittaa enimmäisaikaa, jota yhden erän kokoamiseksi odotetaan ja asetus `batch.size` enimmäiskokoa kilota-

vuina, jota yhteen erään mahtuu. Toisin sanoen jommankumman täyttyessä erä tapahtumaviestejä lähetetään. `linger.ms` -asetuksen ollessa oletuksena nolla (`0`), ei tapahtumien eräkäsittely ole käytössä ollenkaan. (Apache Software Foundation 2023)

6.3 Kokeen eteneminen

Koe suoritettiin ajamalla sitä varten suunniteltu Kafka-tuottaja, joka pyrki lähettämään 100 000 tapahtumaviestiä testiympäristön läpi mahdollisimman nopeasti. Tuottaja tallentaa jokaiseen viestiin tarvittavaa otsikkodataa, joka koostuu kunkin viestin erottavasta yksilöllisestä tunnuksesta, kunkin viestierän erottavasta yksilöllisestä tunnisteesta, aikaleimasta viestin generointihetkeltä ja kullakin ajolla lähetettävästä viestimäärästä (100 000 kpl). Viestin loppu tila täytetään satunnaisella datalla. Koetta varten kehitettiin myös Kafka-kuluttaja, joka osaa tulkita tuottajan viestejä.

Kokeen ensimmäisenä muuttujana toimivat kaikki kolme esiteltyä prosessointitakuuta (*enintään kerran*, *vähintään kerran* ja *täsmälleen kerran*) ja *täsmälleen kerran* -prosessointitakuun osalta kolme eri transaktioaikaa 10 ms, 100 ms ja 1000 ms. Toisena muuttujana oli viestin koko.

Tuottajan ja välittäjän väliltä kirjattiin ensin ylös seuraavien mittareiden tulokset: keskiarvoinen yhden viestin viive (viestin lähetyksestä tuottajalta viestin saapumiseen viestinvälittäjälle kulunut aika millisekunteina), 99. persentiili viiveestä, 95. persentiili viiveestä, 90. persentiili viiveestä, kokonaisaika ja datan keskiarvoinen siirtonopeus (engl. throughput).

Viestien otsikkodatasta muodostuvan datan avulla kirjattiin tuottajan ja kuluttajan väliltä vastaavasti seuraavien mittareiden tulokset: keskiarvoinen yhden viestin viive (viestin lähetyksestä tuottajalta viestin saapumiseen kuluttajalle kulunut aika millisekunteina), 99. persentiili viiveestä, 95. persentiili viiveestä, 90. persentiili viiveestä, matkalle pudonneet viestit, useamman kerran vastaanotetut viestit, kokonaisaika ja datan keskiarvoinen siirtonopeus.

Mittareista mielenkiintoisimmiksi tehokkuusvertailun kannalta valikoitui viiveen keskiarvo, 95. persentiili viiveestä sekä keskiarvoinen siirtonopeus. Jokaista ajoa suoritettiin

kymmenen, joiden tuloksista valittiin kunkin kiinnostavan mittarin osalta mediaaniarvo. Vaihtelevien tulosten syytä ajokertojen välillä pohditaan luvussa 7.

6.4 Kokeen tulokset

6.4.1 Tuottajan ja välittäjän välillä

Taulukko 1 sisältää yksittäisen tapahtumaviestin latenssin keskiarvon sekä sen 95. persentiilin millisekunteina testituottajan ja Kafka-välittäjän väliltä mitattuna, kun viestejä lähettiin yksittäisen tuottajainstanssin toimesta satatuhatta kappaletta mahdollisimman nopeasti perättäin.

tapahtuma-viestin koko	**enintään kerran**	**vähintään kerran**	**täsmälleen kerran (transaktion pituus 10 ms)**	**täsmälleen kerran (transaktion pituus 100 ms)**	**täsmälleen kerran (transaktion pituus 1000 ms)**
128 t	1,08 (4)	8,45 (18)	27,45 (32)	17,90 (35)	6,01 (14)
512 t	1,37 (8)	24,61 (58)	26,71 (31)	26,88 (55)	14,28 (36)
1024 t	3,74 (23)	58,55 (145)	26,22 (30)	25,55 (49)	40,77 (153)

Taulukko 1. Kafka 100 000 viestiä, tuottajan ja välittäjän välinen latenssi per viesti millisekunteina, keskiarvo (suluissa 95. persentiili). Pienempi tulos parempi.

Latenssin keskiarvossa mitattuna paras tulos 1,08 ms saavutettiin käyttäen *enintään kerran* -semantiikkaa ja kokeen pienimpiä 128 tavun kokoisia viestejä. Samaisella muuttujalla mitattuna huonoin tulos 58,55 ms saavutettiin *vähintään kerran* -semantiikalla ja kokeen isoimmilla, 1024 tavun kokoisilla tapahtumaviesteillä. Parhaan ja huonoimman keskiarvon erotus oli 57,47 ms.

Saman välin ja saman viestimäärän keskiarvoinen siirtonopeus mibitavuina mitattuna löytyy seuraavasta taulukosta (Taulukko 2).

tapahtuma-viestin koko	enintään kerran	vähintään kerran	täsmälleen kerran (transaktion pituus 10 ms)	täsmälleen kerran (transaktion pituus 100 ms)	täsmälleen kerran (transaktion pituus 1000 ms)
128 t	21,80	19,68	13,56	31,92	33,49
512 t	52,95	47,92	22,74	53,52	62,97
1024 t	56,39	56,75	24,54	58,67	63,31

Taulukko 2. Kafka 100 000 viestiä, tuottajan ja välittäjän välinen siirtonopeus MiB/s, keskiarvo. Suurempi tulos parempi.

Keskimääräisen siirtonopeuden paras tulos 63,31 MiB/s saavutettiin *täsmälleen kerran* -semantiikalla käyttäen transaktioaikana tuhatta millisekuntia ja viestinä kokeen suurinta, 1024 tavun kokoista tapahtumaviestiä. Huonoiten keskimääräisellä siirtonopeudella 13,56 MiB/s suoriutui *täsmälleen kerran* kymmenen millisekunnin transaktioajalla ja 128 tavun viestikoolla. Parhaan ja huonoimman tuloksen eroksi muodostui 49,75 MiB/s.

Lopputuloksiksi on kunkin mittarin osalta valittu mediaani kymmenen toisistaan erillisen testikerran tuloksista. Kaikkien testikertojen tulokset liitteinä tutkielman lopussa.

6.4.2 Tuottajan ja kuluttajan välillä

Taulukko 3 sisältää yksittäisen tapahtumaviestin latenssin keskiarvon sekä sen 95. persentiilin millisekunteina testituottajan ja testikuluttajan väliltä mitattuna, kun viestejä lähetettiin ja vastaanotettiin yksittäisen tuottajainstanssin ja yksittäisen kuluttajainstanssin toimesta satatuhatta kappaletta mahdollisimman nopeasti perättäin.

tapahtuma-viestin koko	enintään kerran	vähintään kerran	täsmälleen kerran (transaktion pituus 10 ms)	täsmälleen kerran (transaktion pituus 100 ms)	täsmälleen kerran (transaktion pituus 1000 ms)
128 t	11,22 (26)	7,90 (18)	38,18 (45)	70,44 (119)	198,68 (342)
512 t	27,76 (77)	26,54 (63)	38,03 (45)	100,12 (159)	497,61 (739)
1024 t	66,49 (189)	60,31 (148)	37,37 (44)	98,10 (154)	683,76 (980)

Taulukko 3. Kafka 100 000 viestiä, tuottajan ja kuluttajan välinen latenssi per viesti millisekunteina, keskiarvo (suluissa 90. persentiili). Pienempi tulos parempi.

Latenssin keskiarvossa mitattuna tuottaja—kuluttajavälin paras tulos 7,90 ms saavutettiin käyttäen *vähintään kerran* -semantiikkaa ja kokeen pienimpiä 128 tavun kokoisia viestejä. Samaisella muuttujalla mitattuna huonoin tulos 638,76 ms saavutettiin *täsmälleen kerran* -semantiikalla, tuhannen sekunnin transaktioajalla ja 1024 tavun kokoisilla viesteillä. Parhaan ja huonoimman keskiarvon erotukseksi muodostui täten 630,86 ms.

Saman välin ja saman viestimäärän keskiarvoinen siirtonopeus mibitavuina mitattuna löytyy alaluvun toisesta taulukosta (Taulukko 4).

tapahtuma-viestin koko	enintään kerran	vähintään kerran	täsmälleen kerran (transaktion pituus 10 ms)	täsmälleen kerran (transaktion pituus 100 ms)	täsmälleen kerran (transaktion pituus 1000 ms)
128 t	21,47	19,69	13,27	31,42	26,81
512 t	49,96	48,20	22,68	52,79	50,03
1024 t	53,12	56,91	24,56	58,41	46,31

Taulukko 4. Kafka 100 000 viestiä, tuottajan ja kuluttajan siirtonopeus MiB/s, keskiarvo. Suurempi tulos parempi.

Keskimääräisen siirtonopeuden tuottaja—kuluttajavälin paras tulos 58,41 MiB/s saavutettiin *täsmälleen kerran* -semantiikalla käyttäen transaktioaikana sataa millisekuntia ja viestinä 1024 tavun kokoista tapahtumaviestiä. Huonoiten keskimääräisellä siirtonopeudella 13,27 MiB/s suoriutui *täsmälleen kerran* kymmenen millisekunnin transaktioajalla ja 128 tavun viestikoolla. Parhaan ja huonoimman tuloksen eroksi muodostui 45,14 MiB/s.

Lopputuloksiksi on kunkin mittarin osalta valittu mediaani kymmenen toisistaan erillisen testikerran tuloksista. Kaikkien testikertojen tulokset liitteinä tutkielman lopussa. Testiympäristön häiriötiloilta vältyttiin täysin, eli yhtäkään viestiä ei jäänyt saapumatta kuluttajalle, eikä yksikään viesti tuplaantunut yhdessäkään testisuorituksessa.

6.5 Tulosten analyysi

Latenssiltaan pienimmän menetelmän tulos kuluttajan ja välittäjän välillä vastasi pitkälti odotuksia, sillä *enintään kerran* -semantiikalla tehty suoritus oli selkeästi kaikilla eri tapahtumaviestiko'oilla. Samaisella välillä *täsmälleen kerran* sijoittui yllättäen neljän pienimmän latenssiajan joukkoon keskiarvolla 6,01 ms kokeen muuttujina olleessa 128 tavun kokoiset viestit ja tuhannen millisekunnin transaktioaika. Kymmenen ja sadan millisekunnin transaktioajoilla sen tulokset olivat hyvinkin lähellä toisiaan, 26 millisekunnin tietämissä. Selvästi huonoiten tuottajan ja välittäjän välisen latenssin osalta suoriutui ennakko-odotuksia vasten *vähintään kerran* kokeen huonoimmalla keskiarvolla 58,55 ms, kun tuotettavana oli 1024 tavun kokoiset viestit. Tämä ei tosin erottunut enää tapahtumien saapuessa kuluttajalle.

Enintään kerran oli odotetusti latenssiltaan nopea myös koko testijärjestelmän läpi, mutta parhaimman tuloksen tuottajalta kuluttajalle saavutti silti *vähintään kerran*, kun tapahtumaviestien kooksi oli asetettu 128 tavua. Molemmilla semantiikoilla koko matkan latenssi nousi huomattavasti viestikoon kasvaessa, kuten myös *täsmälleen kerran* -kokeissa, lukuun ottamatta kymmenen millisekunnin transaktioajalla toteutettua koetta. Siinä tulokset pysyivät noin 38 millisekunnissa viestien koosta riippumatta. Myös 100 ms transaktioajalla 1024 tavuisten tapahtumaviestien latenssiajat olivat yllättäen 512 tavuisia vistejä pienempiä.

Siirtonopeuden suhteen parhaat tulokset painottuivat käytetystä prosessointitakuusta riippumatta isompiin viestikokoihin. *Täsmälleen kerran* sadan millisekunnin transaktioajalla oli tuottajan ja kuluttajan välillä mitattuna menetelmistä jopa nopein siirtonopeudeltaan, joskin tuhannen millisekunnin transaktioajalla se oli keskiluokkaa ja kymmenen millisekunnin transaktioajalla se suoriutui latenssin lisäksi myös siirtonopeudessa kehnosti.

Keskityttäessä analysoimaan pelkästään *täsmälleen kerran* -tuloksia voidaan huomata, että transaktioajalla ei vaikuta olevan merkitystä keskiarvoiseen latenssiin tuottajan ja välittäjän välillä siinä missä koko järjestelmän läpi suurempi transaktioaika vaikuttaa latenssiin selvästi negatiivisesti. Lähes päinvastaisesti taas suurella transaktioajalla on huomattava positiivinen merkitys siirtonopeuteen tuottajan ja välittäjän välillä siinä missä parhaat tulokset koko järjestelmän läpi saatiin keskimmäisellä, sadan millisekunnin transaktioajalla.

Tuloksia tarkastellessa viestikoon näkökulmasta suurempi tavumäärä vaikuttaa joitain aiemmissa kappaleissa mainittuja poikkeuksia lukuun ottamatta positiivisesti siirtonopeuteen, mutta negatiivisesti latenssiin kaikilla väleillä.

6.6 Tulosten yhteenveto

Kokeen suurin löydös on latenssin ja siirtonopeuden osoittautuminen joitakin poikkeuksia lukuun ottamatta kääntäen verrannollisiksi. Tämä vaikeuttaa prosessointitakuiden tehokkuuden yksiselitteistä määrittelemistä ja täten vertailua keskenään, joten kokeen yhteenvedossa on syytä keskittyä näihin yksitellen. Mielenkiintoisimmat tulokset ovat nähtävissä siirtonopeuden osalta sen selvästi kärsiessä lähes millä tahansa prosessointitakuulla viestikokojen pienetessä. Kuitenkin yllättävää on, että *täsmälleen kerran* vaikuttaa siirtonopeudeltaan kilpailukykyiselle vaihtoehdolle yltämällä sadan millisekunnin transaktioajalla ja 1024 tavun viestikoolla jopa koko välin kärkisijalle.

Pelkästään tuottajan ja välittäjän välillä tarkasteltaessa tulokset eivät *enintään kerran* -menetelmää lukuun ottamatta olleet niin johdonmukaisia, kun ennalta oli ajateltu. Huomiot matalasta siirtonopeudesta pienimmällä 128 tavun viestikoolla lähes millä tahansa prosessointitakuulla saa miettimään, olisiko tuottajien tai kuluttajien asetusta säätämällä pystynyt optimoimaan siirtonopeutta tai toisaalta suurien viestien latenssia.

Kuten todettua, latenssin kannalta pienemmät viestit suoriutuivat luonnollisesti nopeammin. *Täsmälleen kerran* -takuulla vasteajat pysyivät kuitenkin melko vakioina kymmenen ja sadan millisekunnin transaktioaikoja käytettäessä ja tuhannen millisekunnin transaktioajalla ja pienimmillä tapahtumaviesteillä saatiin todella nopea tulos. Tämä saa pohtimaan, oliko testiympäristö optimaalinen, vai asettiko se omat rajoitteensa tuloksille.

Yhteenvetona voidaan mainita, että mikäli halutaan selvästi minimoida koko järjestelmän latenssi, voidaan pyrkiä pieniin tapahtumiin ja *enintään kerran* tai *vähintään kerran* -prosessointitakuuseen. Jos taas halutaan maksimoida järjestelmän läpi kulkeva datan määrä, voidaan pyrkiä suuriin tapahtumiin käyttäen *enintään kerran*, *vähintään kerran*, tai *täsmälleen kerran* -prosessointitakuuta, viimeistä sadan millisekunnin transaktioajalla. Jos tapahtumat ovat lähtökohtaisesti suuria, voidaan latenssi minimoida käyttämällä *täsmälleen kerran* -takuuta kymmenen millisekunnin transaktioajalla. Jos tapahtumat taas ovat pieniä, voidaan siirtonopeus maksimoida käyttämällä *täsmälleen kerran* -takuuta sadan millisekunnin transaktioajalla.

Jos järjestelmän esivaatimus on nojata *täsmälleen kerran* -takuuseen tiedonsiirrossa, on syytä valita pieni transaktioaika (noin 10 ms) minimoidakseen latenssin tai keskisuuresta suuren (>100 ms) transaktioaika maksimoidakseen tiedonsiirtonopeuden. Lisäksi viestikokoa pienentämällä voidaan parantaa yksittäisen viestin keskimääräistä latenssia tai kokoa suurentamalla järjestelmän tiedonsiirtonopeutta.

7 Pohdintaa

Tutkielman empiirisen kokeen suurimmaksi kompastuskiveksi muodostui selvästi ajojen tuloksien epätasaisuus. Vaikka viestejä lähetettiin valtavasti (100 000), saattoi jonkin suorituskerran keskimääräinen viive per viesti vaihdella satoja millisekunteja kuluttajan ja tuottajan välillä mitattuna. Sama ongelma toistui myös keskimääräisen siirtonopeuden kohdalla. Lopulta kokeet jouduttiin suorittamaan alkuperäisen suunnitelman vastaisesti useampaan kertaan ja valitsemaan tuloksista kunkin mittarin mediaani. Testituottajan ja kuluttajan välisten viiveiden ja siirtonopeuksien kohdalla tällaista ongelmaa ei ollut, joten ongelman täytyi olla kulutuspuolella. Vasta jälkikäteen selvisi, että hyvin luultavasti kyseessä oli niin kutsuttu kuluttajaviive (engl. consumer lag) (Mellor 2021). Epätasaisien tulosten takia tutkielman lopputuloksiin otettiin myöhemmin verrattavaksi edellä mainitut, tasaisempia tuloksia tuottaneet testituottajan ja välittäjän väliset viiveet ja siirtonopeudet.

Kuvio 18. Kuluttajaviive (Mellor 2021)

Kuluttajaviive johtuu siitä, ettei Kafka-kuluttaja pysy mukana tuottajan tahdissa. Tämä pullonkaula muodostuu viimeisimmän kuluttajan lukeman viestin ja viimeisimmän tuottajan kirjoittaman viestin välille (Mellor 2021). Tilannetta olisi voinut yrittää parantaa säätämällä kuluttajan aikaperusteisen eräjaon (engl. time-based batching) ylärajaa `fetch.max.wait.ms` tai kokoperusteisen eräjaon (engl. size-based batching) alarajaa `fetch.min.bytes`, mutta näidenkin osalta olisi todennäköisesti täytynyt tasapainoilla viiveen ja siirtonopeuden välillä. Suurin helpotus tällaiseen tilanteeseen olisi saatu perustamalla kuluttajaryhmä ja asettamalla vastaanottavien kuluttajakopioiden määrä vastaavaksi kuin partitioiden lukumäärä, joka testissä käytetyssä topiikissa oli yhdeksän. Tämä olisi vaatinut myös kuluttajaohjelmistolta kyvykkyyttä horisontaaliseen skaalaukseen, joka ei

olisi mahtunut tämän tutkielman toteutuksen puitteisiin. Nyt osittaminen oli kuluttajan näkökulmasta turhaa (Kuvio 15).

Kuten edellisessä luvussa todettua, olisi myös tuottajan asetuksia säätämällä saattanut voida vaikuttaa positiivisesti joko viiveeseen tai siirtonopeuteen eri kokoisilla tapahtumaviesteillä. Nyt tuottajien muuttujat pyrittiin minimoimaan ja esimerkiksi asetukset `linger.ms` ja `batch.size` jätettiin oletuksiinsa, mikä käytännössä poistaa tapahtumien lähettämisen joko tietyn ajan tai tietyn tavumäärän kokoisissa erissä (Apache Software Foundation 2023). Tästä saattoi kärsiä esimerkiksi sekä siirtonopeus pienillä viesteillä että viive täsmälleen kerran -semantiikalla. Tulevan tutkimuksen aiheeksi voisikin sopia joko viiveen tai siirtonopeuden maksimaalinen optimointi keskittyen vain *täsmälleen kerran* -prosessointitakuuseen.

Vaikka koeasetelman yksine tuottajine ja kuluttajineen oli suunniteltu simuloimaan mahdollisimman yksinkertaista mikropalveluarkkitehtuuria käyttäen Kafkaa kommunikointiin, pätee tulos loppujen lopuksi yhtä lailla muihinkin Kafkan käyttötarkoituksiin. Tulevassa tutkimuksessa olisikin syytä rakentaa vielä monimutkaisempi ja todellisuutta vastaavampi mikropalvelurakenne ja valjastaa Kafka sen käyttöön. Jos taas unohdetaan mikropalveluarkkitehtuuri ja keskitytään Kafkan muihin käyttötarkoituksiin, voisi olla mielenkiintoista toistaa vastaava tutkimus käyttäen Kafka Streams -ohjelmointirajapintaa, joka sisältää *täsmälleen kerran* -lupauksen oletuksena (Golder 2022). Kafkan sisäisestä arkkitehtuurista löytyy myös useita kiinnostavia kohteita lisätutkimukselle, kuten transaktiorajapinnan toiminta pellin alla.

Tämän tutkielman tutkimustulokset olisivat laajennettavissa ottamalla muuttujiksi mukaan perille pääsemättömät ja/tai tuplaantuneet tapahtumat kutakin prosessointisemantiikkaa käyttäessä ja arvioiden siten takuiden luotettavuutta. Kyseisessä tutkimuksessa olisi mahdollista hyödyntää tätä tutkielmaa varten luotua Docker-verkkoa lisäämällä siihen esimerkiksi pakettihäviötä tai viivettä generoiva Pumba-testaustyökalu[4]. Myös tässä tutkimuksessa käyttämättömäksi jääneen viestien pakkauksen vaikutusta siirtonopeuteen voisi testata.

[4] https://github.com/alexei-led/pumba

Yksi hyödyllisistä aiheista tulevalle tutkimukselle olisi myös verrata tässä tutkielmassa vähälle huomiolle jääneitä pilvipalveluntarjoajien omia viestinvälitysteknologioita.

8 Yhteenveto

Tämän tutkielman aluksi käytiin läpi ohjelmistoarkkitehtuurin ja erityisesti hajautettujen järjestelmien historiaa. Myöhemmin keskityttiin nykyaikaisiin mikropalveluihin sekä niiden väliseen kommunikaatioon. Kommunikaation osalta tutkielmassa erikoistuttiin tutkimaan mikropalveluiden reaktiivisuuden täyttävää viestilähtöisyyttä, jonka ymmärtämiseksi käytiin läpi palveluiden välisen kommunikaation historiaa. Tämän jälkeen perehdyttiin erityisesti viestiväylänä, tarkemmin määriteltynä tapahtumavirtana toimivan Apache Kafkan toimintaperiaatteisiin. Kafkan osalta rajoituttiin sen käyttöön ja toimintaan mikropalveluympäristössä.

Viestien prosessoimiseksi esiteltiin erilaisia takuita *enintään kerran*, *vähintään kerran* tai *täsmälleen kerran* -semantiikkojen toteutumiseksi, sekä menetelmiä, joilla Kafkan ne luvataan saavuttavan. Empiirisen tutkimuksen tuloksena saatiin analysoitavaksi joitakin ennakko-oletuksia vahventavia tuloksia eri semantiikkojen tehokkuudesta, mutta myös tuloksia, jotka todistivat *täsmälleen kerran* -prosessointitakuun kilpailukykyisyyden. Lopuksi pohdittiin tutkimusosuuden epäkohtia ja tulevia tutkimusaiheita.

Lähteet

Abu-Ghazaleh, Nayef, Madhusudhan Govindaraju, ja Michael J. Lewis. ”Optimizing Performance of Web Services with.” *Proceedings of the International Conference.* Las Vegas, 2004. 482-485.

Apache Software Foundation. ”Apache Kafka Documentation.” 2023. https://kafka.apache.org/documentation.html (haettu 28.8.2023).

Baey, Jean-Christophe. *What is Apache Kafka ?* 2016. https://medium.com/@jcbaey/what-is-apache-kafka-e9e73884e367 (haettu 15.11.2023).

Bohloul, Seyed Mahdi. ”Service-Oriented Architecture: A Review of State-of-the-Art Literature From an Organizational Perspective.” *Journal of Industrial Integration and Management 2021 06:03*, 2021: 353 - 382.

Bonér, Jonas, Dave Farley, Roland Kuhn, ja Martin Thompson. ”Glossary - The Reactive Manifesto.” 2014. https://www.reactivemanifesto.org/glossary (haettu 23.3.2021).

—. ”The Reactive Manifesto.” 2014. https://www.reactivemanifesto.org/ (haettu 23.3.2021).

Brebner, Paul. *Improving Apache Kafka® Performance and Scalability With the Parallel Consumer: Part 1*. 2023. https://www.instaclustr.com/blog/kafka-parallel-consumer-part-1/ (haettu 4.9.2023).

Bryant, Andy. ”Processing guarantees in Kafka.” *Medium.* 2019. https://medium.com/@andy.bryant/processing-guarantees-in-kafka-12dd2e30be0e (haettu 23.10.2022).

But, Colin. *Scaling a Monolith — Vertical Scaling & Horizontal Scaling simply defined.* 2019. https://colin-but.medium.com/scaling-a-monolith-vertical-scaling-horizontal-scaling-simply-defined-4337c8a07326 (haettu 24.4.2023).

Clements, Paul, David Garlan, Reed Little, Robert Nord, ja Stafford Judith. ”Documenting Software Architectures: Views and Beyond.” *25th International Conference on Software Engineering, 2003.* IEEE, 2003.

Conduktor Inc. *Idempotent Kafka Producer.* 2023. https://www.conduktor.io/kafka/idempotent-kafka-producer/ (haettu 22.11.2023).

—. *Kafka Producer Batching.* 2023. https://www.conduktor.io/kafka/kafka-producer-batching/ (haettu 4.9.2023).

—. *Kafka Topics Choosing the Replication Factor and Partition Count.* 2023. https://www.conduktor.io/kafka/kafka-topics-choosing-the-replication-factor-and-partitions-count/ (haettu 4.9.2023).

Confluent Inc. *Building Systems Using Transactions in Apache Kafka®.* 2023. https://developer.confluent.io/learn/kafka-transactions-and-guarantees/ (haettu 14.11.2023).

Denis, Zolotariov. ”Microservice architecture for building high-availability distributed automated computing system in a cloud infrastructure.” *No. 3: Innovative Technologies and Scientific Solutions for Industries* , 2021.

Dhanushka, Dunith. *Understanding Kafka Topic Partitions.* 2021. https://medium.com/event-driven-utopia/understanding-kafka-topic-partitions-ae40f80552e8 (haettu 31.8.2023).

Di Francesco, Paolo, Ivano Malavolta, ja Patricia Lago. ”Research on architecting microservices: Trends, focus, and potential for industrial adoption.” *2017 IEEE International conference on software architecture (ICSA).* IEEE, 2017.

Dragoni, Nicola, ym. ”Microservices: Yesterday, Today, and Tomorrow.” Teoksessa *Present and Ulterior Software Engineering*, tekijä: Manuel Mazzara ja B Meyer, 195-216. Cham: Springer, 2017.

Fernandes, Joel L., Ivo C. Lopes, Joel J. P. C. Rodrigues, ja Ullah Sana. ”Performance Evaluation of RESTful Web Services.” *2013 Fifth international conference on ubiquitous and future networks (ICUFN).* IEEE, 2013. 810-815.

Foote, Keith D. *A Brief History of Microservices.* 2021. https://www.dataversity.net/a-brief-history-of-microservices/ (haettu 27.8.2023).

Garg, Nishant. *Apache Kafka.* Birmingham: Packt Publishing, 2013.

Golder, Rob. *Kafka Streams: Transactions & Exactly-Once Messaging.* 2022. https://www.linkedin.com/pulse/kafka-streams-transactions-exactly-once-messaging-rob-golder/ (haettu 3.2023).

Guidi, Claudio, Ivan Lanese, Manuel Mazzara, ja Fabrizio Montesi. ”Microservices: a Language-based Approach.” Teoksessa *Present and Ulterior Software Engineering,* tekijä: M. Mazzara ja B. Meyer. Cham: Springer, 2017.

IBM. *SOA vs. Microservices: What’s the Difference?* 2021. https://www.ibm.com/blog/soa-vs-microservices/ (haettu 27.8.2023).

Johansson, Lovisa. *Part 1: Apache Kafka for beginners - What is Apache Kafka?* 2020. https://www.cloudkarafka.com/blog/part1-kafka-for-beginners-what-is-apache-kafka.html (haettu 31.8.2023).

Kala, Vítězslav, ja Miroslav Korbelář. ”Idempotence of finitely generated commutative semifields.” *Forum Mathematicum Volume 30 Issue 6,* 2018.

Kwan, Anthony, Jonathon Wong, Hans-Arno Jacobsen, ja Vinod Muthusamy. ”HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres.” 2019.

Lightbend. ”Message Driven vs Event Driven.” 2021. https://developer.lightbend.com/docs/akka-platform-guide/concepts/message-driven-event-driven.html (haettu 23.3.2021).

McKee, Hugh. *How Akka Works: 'At Least Once' Message Delivery.* 2019. https://www.lightbend.com/blog/how-akka-works-at-least-once-message-delivery (haettu 15.11.2023).

Mehta, Apurva, ja Jason Gustafson. *Transactions in Apache Kafka.* 2017. https://www.confluent.io/blog/transactions-apache-kafka/ (haettu 28.8.2023).

Mellor, Paul. *Optimizing Kafka consumers.* 2021. https://strimzi.io/blog/2021/01/07/consumer-tuning/ (haettu 29.8.2023).

Microsoft Corporation. *Communication in a microservice architecture.* 2021. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (haettu 30.3.2021).

Mohr, Tom. *In The Loop — Chapter 24: Reactive Systems and Microservices Architecture.* 2019. https://medium.com/ceoquest/in-the-loop-chapter-24-reactive-systems-and-microservices-architecture-e92493ea60 (haettu 24.4.2023).

Momtselidze, Nodar, ja Ana Tsitsagi. ”Apache Kafka - Real-time Data Processing.” *Journal of Technical Science and Technologies, Volume 4, Issue 2, 2015*, 2015: 31 - 33.

More, Vikram Narayan. ”Rabbitmq microservice with asp.net core web api for async operations.” *International research journal of computer science, 9(8)*, 2022: 336-341.

Narkhede, Neha. *Exactly-Once Semantics Are Possible: Here’s How Kafka Does It.* 2017. https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/ (haettu 28.8.2023).

Nuha, Alshuqayran, Ali Nour, ja Evan Roger. ”A systematic mapping study in microservice architecture.” *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA).* IEEE, 2016.

Patana, Kari. *Mikropalvelujen tiedonsiirron tehokkuus - vertailussa.* Pro gradu -tutkielma, Jyväskylä: Jyväskylän yliopisto, 2020.

Perry, Dwayne E., ja Alexander L. Wolf. ”Foundations for the Study of Software Architecture.” *SOFTWARE ENGINEERING NOTES vol 17 no 4*, 1992: 40 - 52.

Raynal, Michel. ”A Short Introduction to Synchronous Communication.” *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA).* Barcelona: IEEE, 2013. 1136-1143.

Riley, Chris. *Know your API protocols: SOAP vs. REST vs. JSON-RPC vs. gRPC vs. GraphQL vs. Thrift.* 2019. https://www.mertech.com/blog/know-your-api-protocols (haettu 25.9.2023).

Rui, Chen, Li Shanshan, ja Li Zheng. ”From Monolith to Microservices: A Dataflow-Driven Approach.” *2017 24th Asia-Pacific Software Engineering Conference (APSEC).* IEEE, 2017.

Schmutz, Guido. *Building event-driven (Micro)Services with Apache Kafka.* 2018. https://www.youtube.com/watch?v=IR1NLfaq7PU (haettu 24.4.2023).

Sczip, João Guilherme Berti. *Apache Kafka: What Is and How It Works.* 2021. https://medium.com/swlh/apache-kafka-what-is-and-how-it-works-e176ab31fcd5 (haettu 3.9.2023).

Sookocheff, Kevin. *Kafka in a Nutshell.* 2015. https://sookocheff.com/post/kafka/kafka-in-a-nutshell/ (haettu 3.9.2023).

Thönes, Johannes. ”Microservices.” *IEEE Software* 32, nro 1 (2015): 116-116.

Treat, Tyler. *You Cannot Have Exactly-Once Delivery.* 2015. https://bravenewgeek.com/you-cannot-have-exactly-once-delivery/ (haettu 27.8.2023).

—. *You Cannot Have Exactly-Once Delivery Redux.* 2017. https://bravenewgeek.com/you-cannot-have-exactly-once-delivery-redux/ (haettu 27.8.2023).

Upsolver. *Comparing Message Brokers and Event Processing Tools.* 2023. https://www.upsolver.com/blog/comparing-message-brokers-and-event-processing-tools (haettu 27.11.2023).

Van Steen, Marten, ja Tanenbaum Andrew S. *Distributed Systems.* Maarten van Steen, 2018.

Vinka, Elin. *What is Zookeeper and why is it needed for Apache Kafka?* 2018. https://www.cloudkarafka.com/blog/cloudkarafka-what-is-zookeeper.html (haettu 3.9.2023).

VMware Inc. *RabbitMQ - Reliability Guide.* 2023. https://www.rabbitmq.com/reliability.html (haettu 25.9.2023).

Wang, Siao, Chenglie Du, Jinchao Chen, Ying Zhang, ja Mei Yang. ”Microservice Architecture for Embedded Systems.” IEEE, 2021.

Wang, Stephanie, Benjamin Hindman, ja Ion Stoica. ”In reference to RPC: it's time to add distributed memory.” *HotOS '21: Proceedings of the Workshop on Hot Topics in Operating Systems.* Ann Arbor, Michigan: Association for Computing Machinery, 2021. 191 - 198.

Wu, Han, Zhihao Shang, ja Katink Wolter. ”Performance prediction for the apache kafka messaging system.” *IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS).* IEEE, 2019. 145 - 161.

Zettler, Kev. *What is a distributed system?: Atlassian.* 2023. https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture.

Liitteet

A Kokeen aineisto, 128 tavun viestikoko

vähintään kerran 128 t

tuottaja-välittäjä

	latenssi	95th	siirtonopeus
	8,13	16,00	15,69
	7,63	16,00	17,51
	11,16	22,00	19,95
	7,97	16,00	19,63
	15,60	45,00	20,08
	11,74	29,00	19,72
	13,27	27,00	20,41
	6,97	14,00	17,19
	8,00	15,00	19,41
	8,76	19,00	19,98
mediaani	**8,45**	**17,50**	**19,68**

tuottaja-kuluttaja

	latenssi	95th	siirtonopeus
	7,68	16,00	15,79
	7,49	17,00	17,67
	9,94	19,00	11,05
	7,73	17,00	19,78
	14,69	43,00	20,24
	11,84	30,00	20,01
	13,60	31,00	20,65
	7,08	16,00	18,06
	7,76	15,00	19,59
	8,03	19,00	20,14
mediaani	**7,90**	**18,00**	**19,69**

enintään kerran 128 t

tuottaja-välittäjä

	latenssi	95th	siirtonopeus
	1,49	7,00	21,12
	1,53	7,00	22,52
	0,49	2,00	18,22
	0,72	3,00	21,64
	1,16	5,00	21,68
	1,31	4,00	21,27
	1,14	5,00	23,43
	0,86	4,00	22,40
	0,87	3,00	22,69
	1,01	4,00	21,92
mediaani	**1,08**	**4,00**	**21,80**

tuottaja-kuluttaja

	latenssi	95th	siirtonopeus
	16,71	37,00	21,01
	12,77	35,00	22,40
	8,21	24,00	18,08
	13,46	34,00	21,30
	11,50	18,00	11,48
	15,98	37,00	11,38
	10,94	27,00	23,08
	8,38	20,00	22,44
	9,68	20,00	22,77
	10,29	24,00	21,64
mediaani	**11,22**	**25,50**	**21,47**

täsmälleen kerran (10 ms transaktiot) 128 t

tuottaja-välittäjä

	latenssi	95th	siirtonopeus
	27,62	31,00	12,77
	29,63	47,00	11,93
	28,24	34,00	12,17
	27,45	32,00	13,36
	28,36	33,00	13,17
	26,83	31,00	13,75
	27,45	32,00	13,97
	26,94	31,00	14,38
	26,83	30,00	13,87
	27,09	31,00	14,87
mediaani	**27,45**	**31,50**	**13,56**

tuottaja-kuluttaja

	latenssi	95th	siirtonopeus
	42,16	49,00	8,53
	42,94	63,00	11,94
	39,11	46,00	12,17
	38,28	45,00	13,36
	38,48	53,00	13,18
	36,37	42,00	13,76
	38,07	45,00	13,97
	36,32	42,00	14,40
	37,68	43,00	9,06
	36,97	43,00	14,87
mediaani	**38,18**	**45,00**	**13,27**

täsmälleen kerran (100 ms transaktiot) 128 t

tuottaja-välittäjä

	latenssi	95th	siirtonopeus
	17,40	35,00	31,54
	16,88	32,00	32,55
	14,87	30,00	32,47
	19,36	38,00	33,35
	21,01	42,00	29,85
	18,96	37,00	31,30
	16,40	33,00	32,29
	16,02	33,00	33,54
	20,34	35,00	31,46
	18,40	34,00	29,70
mediaani	**17,90**	**34,50**	**31,92**

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
68,87	118,00	30,44
71,68	122,00	31,79
67,12	118,00	32,21
69,59	119,00	33,35
72,50	122,00	29,34
68,67	117,00	30,59
63,32	111,00	31,62
75,54	129,00	33,08
72,13	127,00	31,22
71,29	114,00	28,66
70,44	**118,50**	**31,42**

täsmälleen kerran (1000 ms transaktiot) 128 t

tuottaja-välittäjä

	latenssi	95th	siirtonopeus
	5,77	16,00	32,73
	7,20	18,00	29,49
	5,64	13,00	33,44
	6,39	17,00	34,58
	6,67	18,00	33,08
	5,84	13,00	35,90
	5,40	13,00	34,39
	6,63	14,00	34,48
	5,79	10,00	31,96
	6,18	14,00	33,54
mediaani	**6,01**	**14,00**	**33,49**

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
217,29	370,00	25,43
227,13	390,00	23,70
194,33	342,00	27,13
190,64	334,00	27,74
203,44	341,00	26,25
177,90	317,00	28,26
186,68	326,00	28,26
198,67	341,00	27,49
208,71	366,00	25,33
198,68	344,00	26,48
198,68	**341,50**	**26,81**

B Kokeen aineisto, 512 tavun viestikoko

vähintään kerran 512 t

tuottaja-välittäjä

latenssi	95th	siirtonopeus
53,76	116,00	46,06
21,70	52,00	49,98
22,99	56,00	49,88
27,80	75,00	47,73
23,64	59,00	51,67
40,62	102,00	46,41
25,58	51,00	48,83
17,42	48,00	47,68
33,22	75,00	47,54
21,78	54,00	48,11
mediaani	**24,61**	**57,50**	**47,92**

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
55,32	118,00	46,24
23,09	57,00	49,47
24,88	63,00	49,47
29,27	78,00	47,96
24,92	63,00	51,94
42,69	105,00	46,64
28,16	59,00	49,07
18,99	51,00	48,01
34,57	77,00	47,82
23,11	55,00	48,39
mediaani	**26,54**	**63,00**	**48,20**

enintään kerran 512 t

tuottaja-välittäjä

latenssi	95th	siirtonopeus
1,50	8,00	54,62
1,23	7,00	53,83
4,59	23,00	54,31
0,82	5,00	56,00
6,09	40,00	53,95
5,15	36,00	52,06
1,16	6,00	42,20
1,19	6,00	43,71
1,68	9,00	44,67
0,84	4,00	41,77
mediaani	**1,37**	**7,50**	**52,95**

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
35,27	88,00	52,33
21,72	54,00	51,94
42,41	113,00	52,45
15,27	32,00	54,01
60,39	153,00	50,65
38,82	137,00	49,27
25,11	62,00	41,38
30,41	82,00	42,09
22,30	64,00	43,71
22,23	72,00	40,42
mediaani	**27,76**	**77,00**	**49,96**

täsmälleen kerran (10 ms transaktiot) 512 t

tuottaja-välittäjä

latenssi	95th	siirtonopeus
26,61	30,00	23,43
26,59	30,00	23,22
26,77	32,00	22,73
26,97	32,00	22,18
26,64	31,00	22,74
26,94	33,00	22,97
26,80	31,00	23,17
26,56	30,00	22,51
26,78	31,00	21,89
26,49	30,00	22,67
mediaani	**26,71**	**31,00**	**22,74**

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
38,00	45,00	23,44
37,61	44,00	23,26
38,05	47,00	22,75
38,10	45,00	22,24
38,07	46,00	22,76
38,39	47,00	22,97
38,07	45,00	18,75
37,36	44,00	22,54
37,55	46,00	21,91
37,23	44,00	22,61
mediaani	**38,03**	**45,00**	**22,68**

täsmälleen kerran (100 ms transaktiot) 512 t

tuottaja-välittäjä

	latenssi	95th	siirtonopeus
	28,56	61,00	55,87
	27,75	59,00	54,13
	26,35	49,00	52,90
	30,59	65,00	52,28
	25,62	49,00	55,24
	26,90	54,00	52,45
	21,57	39,00	54,99
	31,89	55,00	49,82
	26,58	67,00	52,11
	26,85	56,00	57,58
mediaani	**26,88**	**55,50**	**53,52**

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
100,53	157,00	53,60
104,74	161,00	52,67
101,62	160,00	52,90
101,44	165,00	51,72
93,42	150,00	55,05
95,79	156,00	52,39
94,55	166,00	53,95
103,66	158,00	49,57
99,71	192,00	51,94
94,76	146,00	56,12
100,12	**159,00**	**52,79**

täsmälleen kerran (1000 ms transaktiot) 512 t

tuottaja-välittäjä

	latenssi	95th	siirtonopeus
	21,31	79,00	60,51
	27,26	78,00	61,96
	12,26	26,00	67,72
	15,99	36,00	63,74
	20,42	60,00	65,10
	13,76	38,00	63,17
	9,52	21,00	63,25
	10,54	30,00	62,76
	14,79	36,00	60,81
	11,04	25,00	62,68
mediaani	**14,28**	**36,00**	**62,97**

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
525,82	769,00	48,63
504,24	734,00	49,93
457,97	692,00	53,60
461,36	727,00	50,81
472,37	707,00	34,03
497,15	733,00	50,13
498,07	743,00	50,34
494,16	743,00	50,29
506,31	766,00	32,55
515,08	762,00	33,38
497,61	**738,50**	**50,03**

C Kokeen aineisto, 1024 tavun viestikoko

vähintään kerran 1024 t

tuottaja-välittäjä

latenssi	95th	siirtonopeus
39,26	100,00	57,72
42,06	112,00	59,19
33,87	84,00	56,29
54,45	150,00	56,78
67,85	139,00	55,39
68,42	161,00	58,23
62,65	161,00	55,08
41,76	108,00	57,48
66,10	208,00	56,71
93,02	264,00	54,89
58,55	**144,50**	**56,75**

mediaani

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
41,18	104,00	57,96
44,46	114,00	59,33
35,62	86,00	56,51
55,90	153,00	56,98
70,15	142,00	55,64
70,89	162,00	58,37
64,71	164,00	54,62
44,10	111,00	57,68
67,70	210,00	56,84
94,79	266,00	55,08
60,31	**147,50**	**56,91**

mediaani

enintään kerran 1024 t

tuottaja-välittäjä

latenssi	95th	siirtonopeus
5,15	29,00	57,31
3,91	23,00	54,47
4,38	27,00	59,11
2,96	16,00	56,74
2,89	18,00	54,25
3,74	23,00	56,35
4,29	29,00	56,42
3,73	22,00	54,77
2,27	12,00	53,98
3,35	18,00	60,81
3,74	**22,50**	**56,39**

mediaani

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
82,39	229,00	53,28
61,43	215,00	52,96
72,13	185,00	56,58
67,43	161,00	53,95
55,15	139,00	52,93
109,90	293,00	52,79
63,89	189,00	53,63
65,55	200,00	52,64
56,82	132,00	51,97
81,10	188,00	58,13
66,49	**188,50**	**53,12**

mediaani

täsmälleen kerran (10 ms transaktiot) 1024 t

tuottaja-välittäjä

latenssi	95th	siirtonopeus
26,53	31,00	24,14
26,72	32,00	23,71
26,29	30,00	24,67
26,40	30,00	24,39
26,14	29,00	24,61
25,97	30,00	24,62
26,05	30,00	24,49
26,39	31,00	23,59
26,05	29,00	24,97
26,13	30,00	24,59
26,22	**30,00**	**24,54**

mediaani

tuottaja-kuluttaja

latenssi	95th	siirtonopeus
37,97	45,00	24,15
37,80	46,00	23,74
37,50	44,00	24,69
37,41	44,00	24,41
37,40	43,00	24,62
36,72	43,00	24,64
36,91	43,00	24,51
37,33	44,00	23,62
36,86	43,00	25,00
37,15	44,00	24,60
37,37	**44,00**	**24,56**

mediaani

täsmälleen kerran (100 ms transaktiot) 1024 t

	tuottaja-välittäjä			tuottaja-kuluttaja		
	latenssi	95th	siirtonopeus	latenssi	95th	siirtonopeus
	25,90	50,00	58,37	101,30	160,00	58,23
	25,80	49,00	58,03	104,15	166,00	57,99
	26,53	48,00	58,03	98,67	157,00	57,48
	25,13	47,00	58,97	96,51	148,00	58,37
	26,88	51,00	58,44	102,73	171,00	58,44
	26,61	54,00	58,83	96,65	151,00	58,72
	22,62	43,00	60,21	97,83	155,00	60,21
	25,30	49,00	58,51	97,03	148,00	58,16
	23,89	53,00	61,23	98,36	153,00	60,36
	25,00	44,00	59,26	96,46	146,00	59,29
mediaani	**25,55**	**49,00**	**58,67**	**98,10**	**154,00**	**58,41**

täsmälleen kerran (1000 ms transaktiot) 1024 t

	tuottaja-välittäjä			tuottaja-kuluttaja		
	latenssi	95th	siirtonopeus	latenssi	95th	siirtonopeus
	41,30	125,00	64,59	666,99	965,00	46,41
	43,11	157,00	63,21	686,97	975,00	45,74
	39,23	175,00	61,93	680,55	967,00	44,73
	40,24	125,00	65,28	655,42	972,00	47,06
	86,73	207,00	53,98	782,24	1081,00	50,34
	55,80	135,00	62,48	635,05	984,00	58,65
	50,61	198,00	60,66	733,70	1017,00	44,65
	37,38	165,00	63,54	715,95	1006,00	45,85
	36,43	133,00	63,41	692,32	993,00	46,28
	36,81	148,00	63,79	656,64	964,00	46,33
mediaani	**40,77**	**152,50**	**63,31**	**683,76**	**979,50**	**46,31**