Kari Patana

Mikropalvelujen tiedonsiirron tehokkuus - vertailussa REST ja JSON sekä gRPC ja Protocol Buffers

Tietotekniikan pro gradu -tutkielma

23. marraskuuta 2020

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Kari Patana

Yhteystiedot: kari.e.m.patana@student.jyu.fi

Ohjaaja: Ari Viinikainen

Työn nimi: Mikropalvelujen tiedonsiirron tehokkuus - vertailussa REST ja JSON sekä gRPC ja Protocol Buffers

Title in English: Performance of communication between micsroservices - comparing REST and JSON with gRPC and Protocol Buffers

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmistotekniikka

Sivumäärä: 65+13

Tiivistelmä: Tutkielmassa selvitetään gRPC-protokollan sekä Protocol Buffers -viestimuodon soveltuvuutta mikropalveluympäristöön etenkin tehokkuuden näkökulmasta. Tutkielma sisältää kirjallisuuskatsauksen muuhun aihealueen tutkimukseen. Lisäksi tutkielma käsittää Go-ohjelmointikielellä suoritetun kokeen.

Kirjallisuuskatsauksen perusteella gRPC ja Protocol Buffers voivat nopeuttaa tiedonsiirron nopeutta noin kaksinkertaiseksi sekä vähentää siirrettävän datan määrää. Toisaalta perinteistä JSON ja REST -tekniikkaa on mahdollista nopeuttaa käyttämällä tilanteeseen sopivaa JSON-kirjastoa sekä pakkaamalla REST-rajapinnan data Gzip-menetelmällä. Lisäksi joissain tilanteissa voi olla tehokkainta käyttää erityisesti käyttöympäristöön laadittua viestiprotokollaa. Tämä kuitenkin tekee yhteistoiminnan muiden järjestelmien kanssa hankalaksi. Myös viestijonoihin perustuva viestintäkanava voi olla hyvä ratkaisu etenkin kuorman tasaukseen mutta lisää kompleksisuutta.

Omassa tutkimuksessa gRPC ja Protocol Buffers osoittautuivat hyväksi valinnaksi mikropalveluista koostuvaan järjestelmään, jossa luetaan OPC UA -protokollalla IoT-dataa ja välitetään sitä verkon yli toiselle mikropalvelulle. Sen sijaan samalla koneella toimivien mikropalvelujen väliseen tiedonsiirtoon JSON ja REST olivat hieman nopeampi vaihtoehto.

Avainsanat: mikropalvelu, go, gRPC, protocol buffers, ProtoBuf, REST, JSON, OPC UA

Abstract: This thesis discusses using the gRPC protocol and the Protocol Buffers message format in a microservices environment especially from the performance point of view. It contains a literary review of other articles in the field. The thesis also describes a test performed with the Go programming language.

Based on the literary review, gRPC and Protocol Buffers may double the speed of data transfer and bring down the size of the data that's transmitted. On the other hand, it is possible to speed up the JSON and REST technique by using a JSON library best suited for the use case and by using Gzip compression in the REST API. Sometimes it can also be preferable to use a dedicated messaging protocol tailored for the system. This, however, makes co-operation with other systems difficult. Using a message queue may also be a good alternative, especially if there's a need to balance the load, but adds some complexity.

In the original research gRPC and Protocol Buffers turned out to be a good choice for a microservices based system used for reading IoT messages with the OPC UA protocol and then transmitting them over a network to another microservice. For a case where both microservices were run on the same computer, JSON and REST turned out to be a slightly faster alternative.

Keywords: go, gRPC, Protocol Buffers, ProtoBuf, REST, JSON, OPC UA

Termiluettelo

Go	Googlen vuonna 2009 kehittämä ohjelmointikieli (ks. “Documentation - The Go Programming Language” 2020).
gRPC	gRPC (gRPC Remote Procedure Calls) on Googlen kehittämä avoimen lähdekoodin järjestelmä etäproseduurikutsujen tekemiseen. Se käyttää HTTP/2-protokollaa tiedonsiirtoon ja Protocol Buffersia rajapintojen kuvauskielenä (ks. Du, Lee ja Kim 2018).
Gzip	on vapaan lähdekoodin pakkausmenetelmä tiedon kompressointiin ja dekompressointiin. (ks. “GNU Gzip” 2020).
IoT	Internet of Things tarkoittaa verkon yli tietoa siirtäviä tiedonkäsittelylaitteita, jotka eivät vaadi interaktiota ihmisten kanssa. (ks. Feki ym. 2013).
JAXB	Java Architecture for XML Binding on Java-kielen sovelluskehys, jonka avulla kehittäjät voivat kuvata Java-luokkia XML-esityksiksi. (ks. Fialli ja Vajjhala 2003).
JSON	JavaScript Object Notation (JSON) on avoin standardi, jossa kuvataan muoto tiedonsiirtoon. Se käyttää ihmisluettavaa tekstimuotoa, jossa objektit koostuvat nimi-arvo-pareista sekä taulukkotyypeistä tai muista serialisoituvista arvoista (ks. “Introducing JSON” 2020).
Kanava	Go-ohjelmointikielessä kanava (englanniksi channel) on tietorakenne, joka muodostaa tyypitetyn johtimen. Sen kautta on mahdollista lähettää ja vastaanottaa kanavalle tyypin määrittämiä arvoja. Kanava voi sisältää myös puskurin, johon arvot kertyvät kunnes niitä luetaan. (ks. “Channels - A Tour of Go” 2020).
Mikropalvelu	Mikropalvelu on sellainen itsenäisesti toimiva ja autonomisesti kehitetty, käyttöön otettu ja skaalautuva liiketoimintakeskeinen osa ohjelmistoa, joka on löyhästi sidoksissa muihin komponentteihin (ks. Indrasiri ja Siriwardena 2018).
OPC UA	Open Platform Communications Unified Architecture (OPC

	UA) on koneiden välisen kommunikaation protokolla jota käytetään teollisuusautomaatioon. Sen on kehittänyt OPC Foundation. (ks. “Unified Architecture - OPC Foundation” 2020).
Pilvipalvelu	Pilvipalvelu tarjotaan verkon kautta. Pilvipalvelun tuottaja vastaa tarpeellisista laskenta-, muisti-, tallennus- ja verkkoresursseista. Palvelun hankkija voi tilata käyttöönsä joko infrastruktuuria, johon voi sijoittaa oman verkkosovelluksen, tai koko verkkopalvelun valmiina. Palvelusta maksetaan yleensä käytön ja kuormituksen mukaan. (ks. Buyya 2010).
Protocol Buffers	Protocol Buffers (ProtoBuf) on tiedon binäärimuotoinen tiedon serialisointimenetelmä. (ks. Du, Lee ja Kim 2018).
REST	Representational state transfer (REST) on ohjelmistoarkkitehtuurin tyyli, joka määrittää joukon rajoitteita WWW-palvelun luomiseksi käyttäen tilattomia operaatioita. REST:iä noudattavaa WWW-palvelua kuvataan termillä RESTful. (ks. Sandoval 2009).
RPC	Remote Procedure Call eli RPC tarkoittaa tekniikka, jossa tietokoneohjelma saa aikaan proseduurin suorittamisen toisessa muistiavaruudessa - tyypillisesti samassa verkossa olevalla toisella tietokoneella - niin, että ohjelmoijan näkökulmasta kutsu tapahtuu samoin kuin lokaalin proseduurin kutsu tapahtuisi. (ks. Nelson 1981).
Serialisointi	Serialisointi on prosessi, jossa data (esimerkiksi olio) muunnetaan bittijonoksi, jotta se voidaan joko lähettää tiedonsiirtokanavan läpi tai tallentaa massamuistiin. Käänteistä prosessia kutsutaan deserialisoinniksi. (ks. Wibowo 2011).
SOA	Service-Oriented Architecture (SOA) tarkoittaa palvelusuuntautunutta arkkitehtuuria. SOA:ssa palvelut tarjotaan sovelluskomponentteina toisille komponenteille verkon välityksellä käyttäen tiettyä kommunikaatioprotokollaa. (ks. Erl 2016)
SOAP	Simple Object Access Protocol (SOAP) on standardi palvelusuuntautuneen arkkitehtuurin (SOA) mukaisten verkkosovel-

	lusten tekemiseen. (ks. Petrie 2016).
XML	Extensible Markup Language on World Wide Web Consortiumin kehittämä rakenteellisten merkintäkielten standardi, joka määrittää tietojen merkintämuodon loogisella rakenteella. (ks. “Extensible Markup Language (XML) - W3C” 2020).
YAML	YAML Ain’t Markup Language on datan serialisointikieli, joka on suunniteltu sekä ihmisluettavaksi että toimimaan hyvin modernien ohjelmointikielten ja tavallisten tarpeiden kanssa. (ks. “YAML - The Natural Language for Data” 2020).

Kuviot

Sisällys

1 Johdanto

Mikropalvelut ovat nousseet tärkeäksi tavaksi rakentaa sovelluksia ja tietojärjestelmiä. Niiden avulla muodostetaan kokonaisuuksia, jotka koostuvat pienistä ja helposti ylläpidettävistä komponenteista ja joilla on hyvin löyhä sidos keskenään.

Koska mikropalvelut tekevät vain hyvin rajoitetun ja tarkkaan määritellyn asian, on tarpeellista yhdistää useita mikropalveluja, jotta haluttu kokonaisuus saadaan toteutettua. Palvelut pitää saada kommunikoimaan keskenään. Perinteinen tapa toteuttaa mikropalvelujen välinen tiedonsiirto on käyttää JSON-tiedostomuotoa sekä REST-rajapintaa.

Mikropalveluja hyödynnetään myös monissa esineiden Internetin eli IoT-ratkaisuissa. Tällaisiin järjestelmiin kuuluu monesti monenlaisen mittausdatan keräämistä ja tallennusta. Tiedot pitää myös saada siirrettyä laitteesta pois.

Yksi mikropalvelu saattaa hyvin olla kovinkin rajallisin resurssein ja vaatimattoman verkkoyhteyden varassa olevassa laitteessa. Toinen mikropalvelu puolestaan voi olla pilvipalvelussa tarjoamassa tavan tallentaa IoT-data tietokantaan. Tällaisessa järjestelmässä tiedonsiirron tehokkuudella voi olla suuri merkitys, jotta kaikki data saadaan tallennettua ja välitettyä rajallisin resurssein.

REST ja JSON saattavat osoittautua rajallisten resurssien ja pienen tiedonsiirtokapasiteetin tilanteissa varsin hitaaksi ja paljon resursseja vaativaksi menetelmäksi. Yksi vaihtoehto näille on rakentaa mikropalvelujen tiedonsiirto käyttämään gRPC-rajapintatekniikkaa sekä Protocol Buffers -viestimuotoa, joilla tiedonsiirtoa voidaan tehostaa.

Tutkielmassa tutustutaan JSON- ja REST-menetelmien taustaan sekä Protocol Buffers -viestimuotoon ja gRPC-rajapintatekniikkaan. Tutkimus pyrkii vastaamaan tutkimuskysymykseen siitä, tarjoavatko gRPC ja Protocol Buffers hyvän vaihtoehdon REST- ja JSON-tekniikoille mikropalvelujen välisessä kommunikaatiossa suorituskyvyn näkökulmasta.

Tutkielman luvussa 2 käydään läpi mikropalveluarkkitehtuurin historiaa ja ominaisuuksia sekä verrataan sitä perinteiseen monoliittiseen arkkitehtuuriin. Lisäksi käydään läpi mikropalvelujen hyviä ja huonoja puolia. Luvussa 3 tutustutaan erilaisiin rajapinta- ja viestinväli-

tystekniikoihin ja tarkastellaan tarkemmin JSON- ja REST-menetelmiä sekä Protocol Buffers -viestimuotoa ja gRPC-rajapintatekniikkaa.

Tutkielman luvussa 4 luodaan kirjallisuuskatsaus JSON- ja REST-menetelmien nopeuttamiseen sekä olemassa olevaan tutkimukseen gRPC-rajapintojen ja Protocol Buffers -viestimuodon tehokkuudesta eri käyttötapauksissa. Luvussa 5 raportoidaan tutkielman käytännönosuudesta. Siinä laaditaan Go-kieltä käyttäen eri tekniikoilla mikropalveluympäristöt OPC UA -protokollalla luetun IoT-datan tiedonsiirtoon ja verrataan eri versioiden suorituskykyä sekä samalla koneella toimivien mikropalvelujen tapauksessa että verkon yli kommunikoivien mikropalvelujen tapauksessa.

Tutkielman tavoitteena on kartoittaa aihealueen kirjallisuutta sekä toteuttaa vertailu eri tekniikoita hyödyntävistä Go-kielisistä mikropalveluista OPC UA -protokollan mukaista dataa käsittelevässä sovelluksessa ja näin vastata tutkimuskysymykseen: "voidaanko mikropalvelujen välistä tiedonsiirtoa nopeuttaa käyttämällä gRPC:tä ja ProtoBufia REST- ja JSON -mallin sijaan?".

2 Mikropalveluarkkitehtuuri

2.1 Hajautettujen järjestelmien kehitys

Valtaosa nykyään käytössä olevista yritystason IT-sovelluksista on tavalla tai toisella hajautettuja, joko laajemmassa mittakaavassa Internetin kautta toimivina tai pienemmässä mittakaavassa paikallisverkossa toimivina. Hajautetuilta järjestelmiltä vaaditaan reaaliaikaista palvelua käyttäjille, jotka voivat sijaita maantieteellisesti laajalla alueella ja jotka saattavat käyttää hyvin erilaisia laitteita. Tämä tuo haasteita järjestelmien kehitykselle. (Lamersdorf 2011)

Perinteiset tietojärjestelmät eivät nojanneet siihen, että laitteet voisivat kommunikoida keskenään verkkoyhteyden yli, vaan kaikki logiikka oli samassa paikassa. Koska verkkoyhteydet ovat vuosien saatossa kehittyneet, merkittävään rooliin on noussut asiakas-palvelin-malli. Sen mukaisessa järjestelmässä käyttäjät voivat lähettää verkkoyhteyden kautta pyyntöjä palvelimelle, joka sitten vastaa käyttäjän pyyntöön. Tällöin asiakkaan käyttämä laite voi olla varsin tehoton, sillä tiedon prosessointi tapahtuu palvelimella. (Lamersdorf 2011)

Mobiiliagentti on eräs kehitysvaihe hajautettujen järjestelmien toteuttamisessa. Asiakas-palvelin-mallissa asiakasohjelmalla tulee olla jatkuva yhteys palvelimeen, jotta tiedot voidaan prosessoida. Mobiiliagentti on paradigma, jonka tavoitteena on ratkaista hitaan verkkoyhteyden ongelma asiakasohjelman ja palvelimen välillä asiakas-palvelin-mallissa. (Alami-Kamouri, Orhanou ja Elhajji 2016)

Tämän jälkeen on kehitetty palvelusuuntautunut arkkitehtuuri SOA (engl. Service-Oriented Architecture) (Erl 2016). Se tarjoaa integroidun ratkaisun erilaisten hajautettujen järjestelmien keskinäiseen kommunikaatioon. SOA-mallissa asiakasohjelma kommunikoi viestin käyttäen etäproseduurikutsua, joka voi olla esimerkiksi SOAP-muotoinen (Petrie 2016). SOA tarjoaa ohjelmistokomponenteille tavan toimia keskenään useiden tietoverkkojen kautta niin, että järjestelmien alustoilla tai niiden toteutukseen käytetyillä ohjelmointikielillä ei ole väliä.

SOA on skaalautuva ja vikasietoinen mutta toisaalta myös monimutkainen. Se aiheuttaa kuormitusta niin verkkoyhteydessä kuin suorittimessa, koska syötteet pitää SOA-mallissa

validoida ennen kuin ne voidaan lähettää palveluille. Näiden haittapuolten vuoksi ilmeni tarve kevyemmille kommunikaatiotavoille, joilla kuitenkin saadaan SOA-mallin hyödyt skaalautuvuudessa ja luotettavuudessa. Tähän haasteeseen ratkaisuja ovat olleet erimerkiksi REST-tyyppiset rajapinnat sekä mikropalvelut. (Salah ym. 2016)

2.2 Mikropalvelut

Mikropalvelut ovat yksi arkkitehtuurinen lähestymistapa rakentaa hajautettuja järjestelmiä. Ne ovatkin nousseet tärkeäksi ohjelmistoarkkitehtuurin tyyliksi. Ne ovat kuitenkin varsin tuore tulokas ja ensimmäisenä määritelmänä mikropalveluille pidetään vuodelta 2014 olevaa Fowlerin ja Lewisin blogikirjoitusta (Fowler ja Lewis 2014). Kirjoituksessa kuvataan mikropalveluarkkitehtuuri näin: tapa kehittää sovellus kokoelmana pieniä palveluja, jotka toimivat omissa prosesseissaan ja kommunikoivat keskenään kevyiden mekanismien kautta.

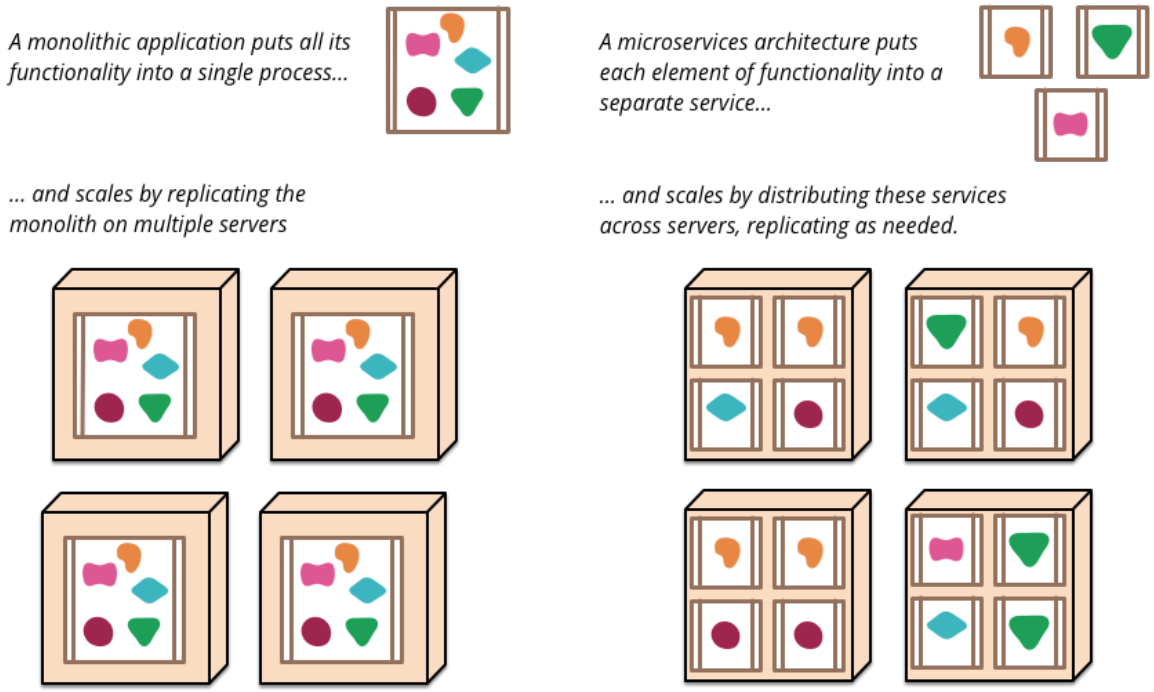
Mikropalveluarkkitehtuurin selittäminen onnistuu usein parhaiten vertaamalla sitä vaihtoehtoiseen ja perinteiseen tyyliin eli niin sanottuun monoliittiin. Perinteiset ohjelmistot sisältävät samassa prosessissa toimivia ominaisuuksia useiden eri liiketoimintatarpeiden huomioimiseen. Mikropalveluissa nämä jaetaan itsenäisesti toimiviin ja suppeata tarvetta varten kehittyihin osiin. (Fowler ja Lewis 2014)

Mikropalvelut ovat kooltaan pieniä. Koko voi toki vaihdella liiketoimintavaatimusten ja esimerkiksi autentikoinnin tuomien vaatimusten mukaan, mutta vaikkapa kyselytutkimuksen (Schermann, Cito ja Leitner 2015) mukaan tutkittujen 40 yrityksen eri kokoisten tiimien kehittämien mikropalvelujen koot olivat harvoin alle sata koodiriviä ja toisaalta yli tuhannen koodirivin mikropalveluja ei juurikaan esiintynyt.

Mikropalvelut on mahdollista ottaa käyttöön itsenäisesti ja ne ovat löyhästi sidoksissa toisiinsa. Niiden toimintaa voidaan myös skaalata tarpeen mukaan palvelukohtaisesti. Mikropalvelut on mahdollista kirjoittaa eri ohjelmointikielillä ja ne voivat käyttää eri tekniikoita tiedon tallentamiseen. (Indrasiri ja Siriwardena 2018)

Monoliitissa toiminnallisuus on yhdessä prosessissa. Jos järjestelmää halutaan skaalata, tämä tapahtuu monistamalla koko monoliitti. Mikropalvelut ovat itsenäisiä prosesseja. Niitä voi-

daan skaalata joustavammin siten, että tarpeellisia palveluja otetaan käyttöön saatavilla olevissa palvelimissa sen mukaan, paljonko kutakin palvelua tarvitaan. Monoliittien ja mikropalvelujen eroja ja skaalautuvuutta hahmottaa kuvio 1. (Fowler ja Lewis 2014)



Kuvio 1. Monoliitit ja mikropalvelut Fowler ja Lewis (2014)

Esimerkiksi verkossa toimiva kauppa vaatii toimiakseen useista osista koostuvan järjestelmän. Erillisinä ja hajautettuina sovelluksina voi olla vaikkapa tilausten hallintaan ja myyntivalikoiman selailuun tarkoitetut komponentit sekä näiden tietokannat. Liiketoimintavaatimuksien toteuttaminen vaatii erillisten sovellusten yhteen kytkemistä. (Indrasiri ja Siriwardena 2018)

Jotta mikropalveluihin perustuva ohjelmisto voi toimia, pitää mikropalveluiden välille luoda viestintäkanava tai -kanavia. Verkkokauppaesimerkissä erillisinä mikropalveluina voivat olla esimerkiksi tilausten hallinta, hakutoiminto, maksaminen ja toimittaminen. Perinteiseen sovellukseen verrattuna erillisiä osia on enemmän, jolloin myös tarve prosessien väliseen kommunikaatioon kasvaa. Prosessien välinen viestintä onkin eräitä tärkeimpiä osa-alueita nykyaikaisten hajautettujen järjestelmien kehittämisessä. (Indrasiri ja Siriwardena 2018)

Prosessien välinen viestintä toteutetaan usein joko synkronisella pyyntö-vastaus-mallilla tai asynkronisena tapahtumamallilla. Synkronisessa mallissa asiakasprosessi lähettää verkon yli

pyynnön palvelinprosessille ja odottaa vastausta. Asynkronisessa tapahtumapohjaisessa viestinnässä prosessi kommunikoi viestiväylän (englanniksi event broker) kautta. Erään mallin synkroniseen kommunikaatioon tarjoavat REST ja JSON. Ohjelmistorajapintoja ja niiden soveltuvuutta mikropalveluihin esitellään tarkemmin luvussa 3. (Indrasiri ja Siriwardena 2018)

2.3 Mikropalvelujen hyötyjä ja haittoja

Mikropalveluihin pohjaavalla arkkitehtuurilla on erilaisia hyötyjä ja haittoja monoliittiseen arkkitehtuuriin verrattuna. Tässä alaluvussa käydään joitakin näkökulmia läpi.

Kyselytutkimuksen Taibi, Lenarduzzi ja Pahl (2017) mukaan ylläpidettävyys on eräs tärkeimpiä lähtökohtia, kun organisaatiot siirtyvät käyttämään mikropalveluarkkitehtuuria. Mikäli eri toiminnot saadaan jaettua eri mikropalveluihin hyvin, on ne mahdollista testata ja ottaa käyttöön itsenäisesti. Lisäksi ohjelmakoodi on ymmärrettävämpää, koska mikropalvelut pyritään tekemään mahdollisimman pieniksi kokonaisuuksiksi.

Lähde Gouigoux ja Tamzalit (2017) toi esiin etuna mahdollisuuden käyttää ohjelmakoodia uudelleen. Monet mikropalvelut voivat olla sellaisia, että niitä voidaan hyödyntää myös jossakin toisessa ohjelmistossa. Lisäksi mikropalvelut on helppo korvata joko toisilla mikropalveluilla tai uusilla versioilla, koska ne ovat itsenäisiä kokonaisuuksia. Myös laadunvarmistaminen helpottuu, sillä jonkin mikropalvelun muuttuessa ei koko ohjelmistoa ole pakko testata.

Toisaalta haasteena on mikropalvelujaon toteuttaminen niin, että muodostuvat mikropalvelut ovat järkeviä kokonaisuuksia. Kirjassa Newman (2015) arvioidaan, että voi olla parempi pitää ohjelma monoliittina, jos ei ole selvää, kuinka se tulisi jakaa mikropalveluihin. Väärien jakolinjojen tekeminen saattaa vaatia myöhemmin suuria muutoksia kaikkiin mikropalveluihin.

Kyselystä Taibi, Lenarduzzi ja Pahl (2017) ilmeni myös, että mikropalvelujen etuna pidetään hyvää skaalautuvuutta. Samaa mikropalvelua on helppo tarpeen mukaan lisätä ajoon joko samalle tai kokonaan eri palvelimelle kuormituksen mukaan. Monoliitissa tämä olisi hankalampaa, koska ohjelmistoa täytyy skaalata kokonaisuutena. Mikropalvelut mahdollistavat

eniten käytettyjen ohjelmiston osien skaalaamisen.

Käyttöönotto voi olla mikropalveluarkkitehtuurin haittapuolia kirjan Newman (2015) mukaan. Tähän vaikuttaa hajautetun ohjelmiston kompleksisuus. Hallintaan on syytä rakentaa automaatio, jotta kaikki mikropalvelut ja niiden kytkökset saadaan asianmukaisesti käyttöön.

Mikropalveluihin pohjaava arkkitehtuuri vaikuttaa myös kustannuksiin. Kyselyn Taibi, Lenarduzzi ja Pahl (2017) perusteella mikropalvelut aiheuttavat etenkin niitä käyttöön otettaessa enemmän kustannuksia. Toisaalta pidemmällä aikavälillä kustannukset voivat laskea, koska mikropalvelujen käyttö helpottaa prosesseja.

Mikropalveluihin liittyy myös tietoturvahaasteita. Tutkimus Yarygina ja Bagge (2018) esittelee erilaisia ongelmakohtia. Niitä ovat esimerkiksi laitteistoon ja virtualisaatioon sekä käytettyyn pilvipalveluun liittyvät ongelmat ja toisaalta mikropalvelujen kommunikaatioon ja hallintaan liittyvät ongelmat. Näistä erityisesti kommunikaatioon ja mikropalvelukokonaisuuden hallinnan eli orkestraation tietoturvahaasteet puuttuvat monoliittiin pohjaavista arkkitehtuureista.

Haastattelututkimuksessa Ghofrani ja Lübke (2018) selvitettiin mikropalveluarkkitehtuurin asiantuntijoiden näkemyksiä heidän kokemistaan kehityshaasteista. Tärkeinä asioina mainittiin puutteita tarjolla olevissa notaatioissa, metodeissa ja sovelluskehyksissä, joilla mikropalveluarkkitehtuurin mukaisia sovelluksia voidaan kehittää. Kyselyn perusteella asiantuntijat pitävät tietoturvaa, vastausaikaa ja suorituskykyä tärkeämpinä kehityskohteina kuin kyky palautua, luotettavuus, vikasietoisuus ja muistinkäyttö.

Artikkelissa Taibi ja Lenarduzzi (2018) on tutkittu mikropalvelujen mahdollisten ongelmien havainnointia. Artikkelissa käytetään nimitystä ”paha haju” kuvaamaan asiaa, joka saattaa olla oire jostakin ongelmasta arkkitehtuurissa. Termi on otettu käyttöön perinteisen ohjelmistoarkkitehtuurin tapauksessa tutkimuksessa Garcia ym. (2009). Kyselytutkimuksessa 72 kokenutta mikropalvelujen kehittäjää pyydettiin kertomaan heidän kokemistaan ongelmista mikropalvelujen kehittämisessä. Näistä tutkijat muodostivat yksitoista pahaa hajua mikropalveluissa sekä mahdollisia ratkaisuja ongelmiin. Näitä kuvataan seuraavassa.

Ensimmäinen paha haju oli rajapinnan versioinnin puute. Mikropalvelujen käyttämät raja-

pinnat saattavat muuttua. Jos muutos tapahtuu niin, että rajapintaa käyttävä toinen osapuoli ei sitä havaitse, voi seurauksena olla esimerkiksi yhteysongelmia, sillä rajapintaa saatetaan kutsua eri tavalla tai eri tiedoin. Ongelma voidaan ratkaista käyttämällä semanttista versiointia (esimerkiksi API v1.1, v.1.2, jne.).

Toinen paha haju olivat sykliset riippuvuudet. Tämä tarkoittaa, että palvelu A kutsuu palvelua B, joka kutsuu palvelua C, joka puolestaan kutsuu palvelua A. Tällaisia palveluja on hankala ylläpitää tai käyttää uudelleen yksittäisinä. Ratkaisuksi tarjottiin syklien jaottelu muodon mukaan ja rajapintayhdyskäytävän käyttäminen.

Kolmas tutkijoiden toteama paha haju oli ESB:n käyttäminen. ESB tulee sanoista Enterprise Service Bus. Se tarjoaa väylän kommunikaatiolle mutta tuo kompleksisuutta palvelujen rekisteröinnissä ja rekisteröinnin peruutuksessa. Parempana ratkaisuna on käyttää kevyempää viestiväylää yritystason ESB:n sijaan.

Neljäs paha haju oli kovakoodatut päätepisteet. Tällä viitataan IP-osoitteisiin ja porttinumeroihin, jotka on kovakoodattu ohjelmakoodin sisään. Tämä aiheuttaa ongelmia silloin, kun päätepisteiden sijaintia pitää muuttaa. Ratkaisuna on käyttää palvelunselvityksen mallia (engl. Service Discovery).

Viides paha haju oli palvelujen asiaton intimiteetti, jolla viitataan siihen, että mikropalvelut käyttävät toisten palvelujen privaattia dataa. Tämä voi ilmetä esimerkiksi niin, että mikropalvelu lukee toisen mikropalvelun omistamaa dataa suoraan tietokannasta. Tällöin mikropalvelut kytkeytyvät liiaksi toisiinsa ja niiden erillinen kehittäminen on hankalaa. Ratkaisuna voi olla yhdistää tällaiset mikropalvelut.

Kuudes paha haju oli mikropalveluahneus. Tällä viitataan siihen, että jokaista mahdollista ominaisuutta varten luodaan erillinen mikropalvelu ja ääritapauksessa jokainen yksittäinen HTML-sivu saattaa tulla omasta mikropalvelusta. Tällöin mikropalvelujen määrä järjestelmässä voi kasvaa hallitsemattoman korkeaksi. Ratkaisuna on huolellinen harkinta mietittäessä tarvitaanko uutta mikropalvelua.

Seitsemäs paha haju oli rajapintayhdyskäytävän eli API Gatewayn puute. Tällöin mikropalvelut keskustelevat suoraan toistensa kanssa. Kun mikropalvelujen määrä järjestelmässä

kasvoi yli viidenkymmenen, oli monien kehittäjien mielestä mahdotonta hallita mikropalvelujen välisiä yhteyksiä. Ratkaisuna on käyttää API Gateway -mallia, jossa rajapintayhdyskäytävä huolehtii mikropalvelujen sijainnista ja niiden välisistä yhteyksistä.

Kahdeksas paha haju olivat jaetut kirjastot. Jos usea mikropalvelu käyttää samoja kirjastoja, voivat niiden väliset sidokset muuttua vahvemmiksi. Jos yhden palvelun käyttämä kirjasto muuttuu, voi muutos vaatia päivityksen myös muissa samaa kirjastoa käyttävissä palveluissa. Ratkaisuna voi olla kirjaston muokkaaminen omaksi mikropalvelukseen. Tällöin funktiokirjaston virkaa hoitavan palvelun muutos tulee käyttöön kaikissa sitä käyttävissä mikropalveluissa yhdellä kertaa.

Yhdeksäs paha haju oli jaetun tallennustilan käyttö. Tällöin usea mikropalvelu käyttää esimerkiksi samaa relaatiotietokantaa. Seurauksena voi olla, että mikropalveluille tulee liian vahva sidos, joka vähentää niin palvelujen kuin niitä kehittävien tiimien itsenäisyyttä. Ratkaisuina voi olla esimerkiksi erillisten tietokantojen tai tietokantaskeemojen käyttö eri palveluilla tai tietokantataulujen muuttaminen niin, että ne ovat yksityisiä tietylle palvelulle.

Kymmenes paha haju olivat liian useat standardit. Tällä tarkoitetaan esimerkiksi liian monen eri ohjelmointikielen, protokollan tai ohjelmistokehyksen käyttöä. Vaikka mikropalvelut kannattaa tehdä sillä tekniikalla, joka kyseisen palvelun toteuttamiseen parhaiten sopii, voivat liian useat eri tekniikat aiheuttaa ongelmia vaikkapa kehitystiimien jäsenten vaihtuessa. Ratkaisuna on tarkka harkinta siinä, mitä tekniikoita on järkevää käyttää. Tutkijat suosittelevat viimeisimpien hype-tekniikoiden välttämistä.

Yhdestoista tutkijoiden havaitsema haju olivat väärät jakokohdat. Tällä tarkoitetaan sitä, miten järjestelmä on jaettu mikropalveluihin. Mikropalveluarkkitehtuurin jakoa voidaan yrittää tehdä teknisten kerrosten (kuten tiedon näyttäminen, toimintalogiikka ja tietokanta) mukaan sen sijaan että käytettäisiin jaottelua liiketoimintatarpeiden mukaan. Tällöin eri mikropalvelujen vastuut ja niiden väliset liitokset voivat olla epäselviä. Ratkaisuna on analysoida liiketoimintaprosessit tarkkaan ja miettiä niiden jakautuminen erillisiksi mikropalveluiksi.

Yhteenvetona tutkijat kokosivat viisi oppituntia kehittäjien ja ohjelmistoarkkitehtien avuksi:

Oppitunti 1: Perinteisten pahojen hajujen (Garcia ym. 2009) lisäksi mikropalveluja koskettavat

pahat hajut voivat tuottaa ongelmia niihin perustuvien järjestelmien kehittämisessä ja ylläpidossa. Kehittäjät voivat hyötyä tutkielmassa kerättyjen pahojen hajujen kokoelmasta ottamalla niistä oppia, jotta voivat välttää kokemasta niihin liittyviä huonoja käytäntöjä.

Oppitunti 2: Ohjelmistoarkkitehdin rooli on jälleen tulossa tärkeäksi. Arkkitehtuuriin ja järjestelmätason valinnat tulee tehdä syvän mikropalveluosaamisen perusteella.

Oppitunti 3: Monoliitin jakaminen mikropalveluihin vaatii monoliitista eristettävien itsenäisten liiketoimintaprosessien tunnistamista sen sijaan, että ominaisuudet ulkoistetaan erillisiin verkkopalveluihin.

Oppitunti 4: Mikropalvelujen väliset yhteydet - sisältäen yhteydet yksityiseen dataan sekä jaettuihin kirjastoihin - tulee analysoida huolellisesti.

Oppitunti 5: Perussääntönä kehittäjiä tulee varoittaa, jos he tarvitsevat syvällistä tietämystä muiden palvelujen sisäisestä toiminnasta tai jos muutokset yhdessä mikropalvelussa vaativat muutoksia myös jossakin toisessa mikropalvelussa.

3 Ohjelmistorajapinnat ja viestinvälitystekniikat

Ohjelmistorajapinta tarjoaa hyvin määritellyn tavan, jolla jokin ohjelmakomponentti saa ohjelmallisesti pääsyn johonkin toiseen komponenttiin. Englanninkielinen nimitys termille on Application Programming Interface ja siitä tuleva lyhenne API on monesti käytössä myös suomenkielisessä tekstissä.

Lähteen Mumbaikar ja Padiya (2013) mukaan Carnegie Mellon -yliopistossa toimiva Software Engineering Institute on 2003 määrittänyt API:n kahden tai useamman erillisen sovelluksen väliseksi tavaksi siirtää viestejä tai tietoa. Nykyään API käsitetään laajemmin niin, että se on hyvin määritelty rajapinta, joka kuvaa palvelun, jota jokin komponentti, moduuli tai sovellus tarjoaa muille ohjelmistoelementeille.

Kuten sana rajapinta kuvaa, ohjelmistorajapinnat toimivat kahden tai useamman ohjelmakomponentin välillä. Yksi esimerkki ohjelmistorajapinnoista on Windows-käyttöjärjestelmän rajapinnat, jotka tarjoavat sovelluksille tavan hyödyntää järjestelmän resursseja, kuten tiedostojärjestelmää ja prosessien ajastusta. Monesti Java-tyyppisissä ohjelmointikielissä API vastaa kokoelmaa julkisia luokkia ja rajapintoja sekä niihin liittyvää dokumentaatiota. Eri sovellukset, jotka käyttävät yhteistä rajapintaa, ovat usein erillisten ohjelmointitiimien tekemiä. API tarjoaa tavan määritellä liittymäkohdat niin, että eri tiimit voivat tehdä oman projektinsa tuotokset itsenäisesti niin, että yhteistoiminta voidaan varmistaa. Toisaalta rajapintojen avulla voidaan erotella ohjelman julkinen osuus (API) sekä yksityinen osuus eli varsinainen toteutus, jolloin muutos sisäiseen toimintaan voidaan tehdä ilman, että ennallaan pysyvän rajapinnan käyttäjä huomaa. Näin järjestelmien välinen riippuvuus vähenee. (Mumbaikar ja Padiya 2013)

Rajapintoja voidaan luokitella monin eri tavoin. Opinnäytteessä Speth (2017) on jaettu rajapintoja tällaisiin osittain päällekkäin meneviin tyyppeihin: liitännäis- ja kirjastorajapinnat, päätepisterajapinnat, REST, etäproseduurikutsu, viestinvälitysrajapinnat, virtausrajapinnat sekä synkroniset ja asynkroniset rajapinnat.

Kirjassa Newman (2015) huomautetaan, että mikropalvelujen välisen kommunikaation tulee tapahtua verkkokutsujen välityksellä. Tämä pakottaa jakamaan palvelujen vastuut erilleen,

jolloin palvelujen väliset tiukat kytkökset vähenevät. Mikropalvelut ovat siis luonnostaan myös verkkopalveluita.

Petrie (2016) toteaa, että verkkopalvelut (engl. web service) on perinteisesti määritelty olemaan SOAP-muotoisia RPC-kutsuja ja että ne noudattavat palveluorientoitunutta arkkitehtuuria (engl. SOA, Service Oriented Architecture). Hän kuitenkin korostaa, että näin ei ole pakko olla.

Petrie (2016) jatkaakin määrittelemällä, että verkkopalvelu on mahdollisesti etänä suoritettava proseduuri. Sitä kutsutaan tavalla, joka ilmaistaan standardilla ja koneluettavalla syntaksilla. Se on saavutettavissa standardeilla Internet-protokollilla. Sillä on kuvaus, joka sisältää vähintäänkin hyväksytyt syöte- ja tulosteviestit, ja sillä voi olla semanttinen annotaatio palvelukuvauksesta ja datan merkityksestä.

Mainittu SOA-mallinen arkkitehtuuri koostuu kolmesta entiteetistä: palveluntarjoaja, palvelurekisteri ja palvelun pyytäjä. Näistä ensimmäinen on varsinainen käytettävä palvelu, joka suorittaa palvelun kuluttajilta tulevia pyyntöjä. Palvelun kuluttaja puolestaan on sovellus, palvelu tai muunlainen ohjelmistomoduuli, joka pyytää palvelua. Palvelurekisteri taas on verkkopohjainen hakemisto, joka sisältää tarjolla olevat palvelut. Palvelun kuluttaja löytää rekisteristä palvelukuvauksen, jonka perusteella se voi alkaa vuorovaikutuksen palvelun kanssa. Liikenne entiteettien välillä perustuu XML-kieleen ja SOAP-protokollaan. SOAP-viestit koostuvat kuoresta, otsikosta sekä viestistä. Kuori identifioi XML-dokumentin SOAP-viestiksi. Otsikko sisältää kutsu- ja vastaustiedot. Viestit ja invokaatiot on määritelty XML-dokumentteina, jotka lähetetään viestiprotokollan yli. (Mumbaikar ja Padiya 2013)

Garcia ja Abilio (2017) vertasi SOAP-tekniikkaa REST-tyyppiseen (representational state transfer) tekniikkaan ja totesi, että koska SOAP tarvitsee aina ympärilleen XML-rakenteen, on se hitaampi ja enemmän tiedonsiirtokapasiteettia vaativa tekniikka. Lisäksi tutkimuksessa verrattiin kahta erilaista REST-tekniikkaa, joissa toisessa käytettiin viestimuotona XML-kieltä ja toisessa JSON-notaatiota. Tutkimuksen mukaan XML-muotoinen viesti on usein jopa kolme kertaa isompi kuin JSON-muotoinen viesti. Lisäksi XML-viestin prosessointi vaatii enemmän laskenta-aikaa. Tutkimuksessa päädyttiinkin tulokseen, joka puoltaa nykyään vallalla olevaa tapaa toteuttaa verkkopalvelujen tiedonsiirtoa: REST- ja JSON-tekniikat ovat

tehokkaampia kuin XML-kieleen pohjaava SOAP.

Alaluvuissa esitellään tarkemmin REST- ja JSON-pohjaisia rajapintoja sekä niille vaihtoehdon tarjoavia gRPC- ja Protocol Buffers -tekniikoita. Lisäksi esitellään rajapinnan määrittelyyn sopivaa OpenAPI-määritystä. Tämän jälkeen käsitellään viestijonoihin pohjaavaa kommunikaatiokanavaa.

3.1 REST

Termi REST eli representational state transfer sai alkunsa Roy Fieldingin väitöskirjasta (Fielding 2000) vuonna 2000. Se käsittää kokoelman rajoitteita, joilla voidaan muodostaa arkkitehtuuri WWW-sovelluksille. Rajoitteita noudattavaa järjestelmää kuvataan RESTful-sanalla ja sellaista kuvaavia tekijöitä ovat seuraavat:

1. Sen on oltava asiakas-palvelin-tyylinen.
2. Sen on oltava tilaton niin että jokainen järjestelmässä tapahtuva kutsu on itsenäinen eikä palvelimen tarvitse pitää yllä tietoa käyttäjän istunnosta.
3. Sen tulee tukea välimuistiratkaisuja.
4. Se tarjoaa yhtenäisen rajapinnan. Jokaisella resurssilla on yksilöllinen tunniste.
5. Sen on oltava kerrostettu ja tukea skaalautuvuutta.
6. Sen tulisi tarjota mahdollisuus ladata ajettava koodi silloin, kun sitä tarvitaan. Tämä rajoite on kuitenkin valinnainen.

Rajoitteet eivät määritä teknologiaa ja monilla ennen REST-termin muodostamista kehitetyillä tekniikoilla ja protokollilla voidaan luoda REST-tyylinen arkkitehtuuri. REST-arkkitehtuuri on helposti ylläpidettävä, laajennettava ja hajautettu. (Sandoval 2009)

Myös WWW-palvelut saattoivat noudattaa REST-mallia jo ennen termin kehittämistä. Esimerkiksi staattiset WWW-sivut noudattavat tällaista arkkitehtuuria. Toisaalta useat dynaamiset WWW-palvelut eivät ole REST-mallin mukaisia, sillä ne eivät ole tilattomia vaan vaativat joko palvelinpuolen sessioita tai selainpuolen evästeitä pitääkseen yllä tietoa siitä, mitä asiakas on tekemässä. (Sandoval 2009)

REST-mallinen järjestelmä voidaan jakaa tällaisiin abstraktioihin: resurssit, representaatiot,

URI:t ja HTTP:n eri tyyppiset pyynnöt, jotka muodostavat pysyvän rajapinnan asiakas-palvelin-liikenteelle. Resurssi on jotain, jonka voi osoittaa ja siirtää WWW:n yli, kuten uutisartikkeli, mittaustulos tietyllä ajanhetkellä tai listaus koodiversioista versiohallintajärjestelmässä. Representaatio on esitystapa resurssille ja se voi olla esimerkiksi HTML-sivu, kuva tai JSON-virta. URI eli uniform resource identifier on yksilöllinen hyperlinkki resurssiin ja se on REST-mallissa ainoa tapa, jolla asiakasohjelma ja palvelin voivat välittää representaatioita.

HTTP:hen kuuluu useita eri metodeja, joilla resursseja käsitellään representaatioiden kautta. Niitä voidaan käyttää representaatioiden välittämiseen niin, että resurssien tilaa on mahdollista myös muuttaa. Tärkeimmät HTTP-metodit ovat GET, joka palauttaa resurssin, POST, jolla luodaan resurssi, PUT, jolla päivitetään resurssia, sekä DELETE, jolla poistetaan resurssi. (Sandoval 2009)

3.2 JSON

JavaScript Object Notation eli JSON on avoin standardi tiedostomuodolle, jota käytetään datan välittämisessä. Se käyttää ihmisluettavaa tekstimuotoa, jossa dataobjektit koostuvat attribuutti-arvo-pareista sekä taulukkomuotoisista tietotyypeistä. Nimensä mukaisesti JSON on saanut alkunsa JavaScript-ohjelmointikielestä, mutta sitä voidaan käyttää kaikkien ohjelmointi- ja luonnollisten kielten kanssa. Useissa ohjelmointikielissä on valmiina toimintoja JSON-muotoisen datan luomiseen ja parsimiseen. (“Introducing JSON” 2020)

Seuraavassa listauksessa on esimerkki JSON-muotoisesta datasta, jossa esitellään henkilötietoja ja perhesuhteita:

```
{
  "etunimi": "Matti",
  "sukunimi": "Meikäläinen",
  "elossa": true,
  "ikä": 27,
  "osoite": {
    "lähiosoite": "Isokatu 1",
```

```
    "postinumero": "00001",
    "postitoimipaikka": "Isojoki"
  },
  "puhelinnumerot": [
    {
      "tyyppi": "koti",
      "numero": "049-132456"
    },
    {
      "tyyppi": "työ",
      "numero": "048-968574"
    }
  ],
  "lapset": [],
  "puoliso": null
}
```

Yhdessä REST-arkkitehtuurityylin ja HTTP-protokollan kanssa JSON on muodostunut de facto -standardiksi mikropalvelujen tiedonsiirtomenetelmänä. JSON-muodon etuna on helppo luettavuus ihmiselle. Toisaalta mikropalvelujen välisessä tiedonsiirrossa se voi olla tehoton, koska JSON-muoto vaatii varsinaisen siirrettävän datan ympärille paljon kontekstia. (Indrasiri ja Kuruppu 2020)

3.3 gRPC

gRPC (gRPC Remote Procedure Calls) on Googlen kehittämä avointa lähdekoodia käyttävä tekniikka. Sen avulla on mahdollista suorittaa etäproseduurikutsuja RPC-tyylin mukaisesti. RPC on mekanismi, jota käytetään monissa hajautetuissa järjestelmissä prosessien väliseen kommunikaatioon. RPC-termi on ollut käytössä ainakin vuodesta 1981 (Nelson 1981) alkaen. Sitä pidetään mekanismina, joka tarjoaa prosessien välisen viestinvälityksen helpolla ja läpinäkyvällä tavalla niin, että kuormitus ei kasva merkittävästi. Koska RPC:tä kutsuva proses-

si pysähtyy ja jää odottamaan vastausta etäproseduurilta, lasketaan RPC ja sitä myöten gRPC synkroniseksi kommunikaatioksi. Duplex streaming -ominaisuus kuitenkin mahdollistaa myös viestien asynkronisen lähettämisen pysyvän yhteyden kautta. (Indrasiri ja Kuruppu 2020)

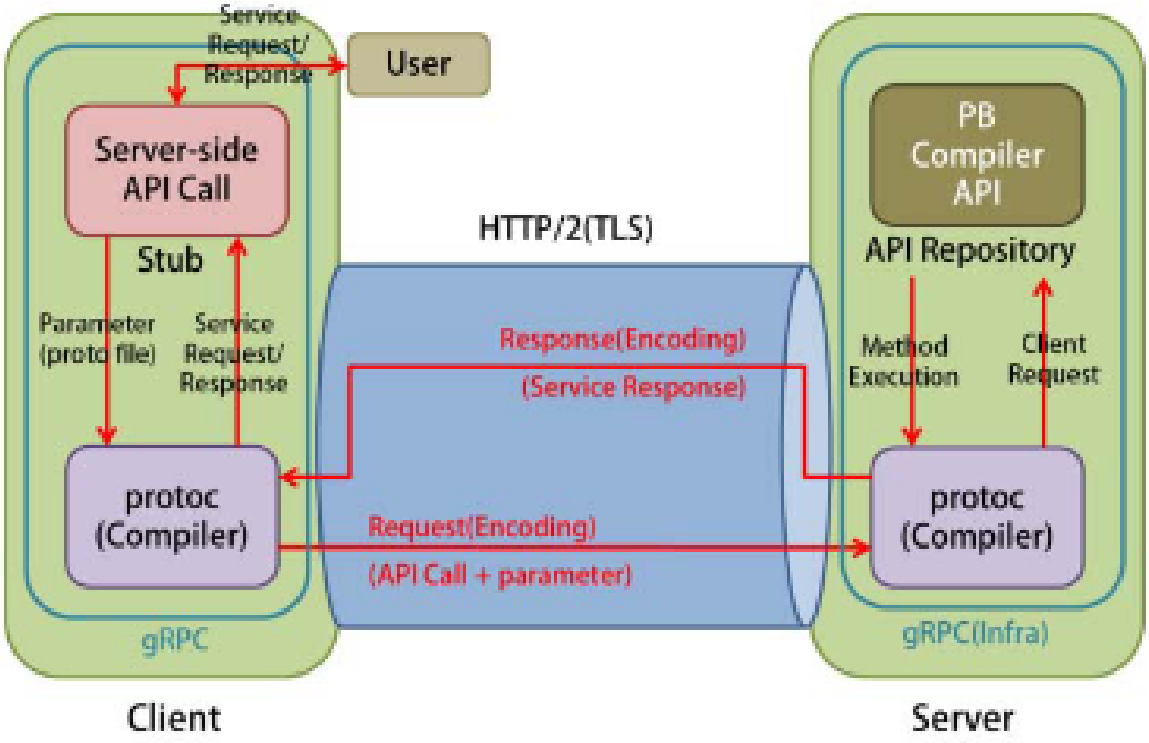
Koska gRPC käyttää HTTP/2-protokollaa, tarjoaa se myös mahdollisuuden protokollan virtaominaisuuden (englanniksi streaming) käyttämiseen. Sitä käyttäen niin asiakkaan lähettämässä viestissä kuin palvelimen palauttamassa vastauksessakin voidaan välittää saman yhteyden kautta useampi viesti. Virta voi olla joko vain toiseen suuntaan tai kaksisuuntainen. HTTP/2 tarjoaa myös etuja suorituskykyyn ja tietoturvaan liittyen. (Du, Lee ja Kim 2018)

gRPC-sovelluksen kehittäminen alkaa palvelurajapinnan kuvaamisella. Palvelurajapinta sisältää tietoa esimerkiksi siitä, kuinka kuluttajat voivat käyttää palvelua, mitä metodeja kuluttajat voivat kutsua etäältä ja mitä parametreja sekä viestimuotoja metodien kutsumisessa käytetään. Palvelurajapintojen kuvaamiseen käytetään erityistä rajapintakuvauskieltä. Useimmiten kielenä on käytössä Protocol Buffers, mutta muidenkin käyttö on mahdollista. (ks. “gRPC Concepts” 2020).

Palvelukuvausta käyttämällä on mahdollista generoida palvelinpuolen ohjelmakoodi eli palvelinluuranko. Tämä helpottaa palvelinlogiikkaa tarjoamalla abstraktiot matalan tason kommunikaatiolle. Samoin on mahdollista generoida niin sanottu tynkä asiakasohjelmalle, mikä helpottaa kommunikaatiota tarjoamalla abstraktiot matalan tason kommunikaatiolle eri ohjelmointikielissä. Käytetty ohjelmointikieli voidaan valita vapaasti, eikä palvelinta ja asiakasta tarvitse kirjoittaa samalla kielellä. (Indrasiri ja Kuruppu 2020)

Palvelinrajapinnassa määriteltyjen metodien etäkutsuminen asiakasohjelmasta on tällöin yhtä helppoa kuin lokaalien funktioiden kutsuminen. Pohjana oleva gRPC-kehys vastaa monista muutoin haastavista asioista kuten datan serialisoinnista, verkkoliikenteestä, autentikoinnista ja pääsynhallinnasta. (Indrasiri ja Kuruppu 2020)

gRPC-kutsun suorittamiseen liittyy oleellisena osana serialisointi ja deserialisointi, joista käytetään englanninkielisiä sanoja marshaling ja unmarshaling. Serialisoinnissa parametrit ja etäfunktio pakataan viestipakettiin, joka voidaan lähettää verkon yli. Deserialisoinnissa viestipaketti puretaan vastaavaan metodikutsuun. gRPC:n toimintaa selventää kuvio 2. (Du, Lee ja Kim 2018)



Kuvio 2. gRPC:n operaatioprosessi asiakkaan ja palvelimen välillä (Du, Lee ja Kim 2018)

Tutkimuksessa (Roohitavaf ym. 2019) käytettiin gRPC:n virtaominaisuutta LogPlayer-nimisessä komponentissa. Se on tarkoitettu käytettäväksi hajautettujen tietokantojen WAL-ominaisuuden, eli lokin etukäteen kirjoittamisen (englanniksi Write-Ahead Logging), yhteydessä. LogPlayer varmistaa, että kaikki lokiin kirjoitetut tapahtumat välitetään jokaiselle tietokantanoodille tasan yhden kerran ja oikeassa järjestyksessä. Tutkimuksessa laadittiin järjestelmä, joka käyttää LogPlayeria gRPC:n kautta. Vertailun vuoksi vastaava järjestelmä laadittiin myös niin, että virta toteutettiin Apache Kafka -viestiväylän avulla. Tutkimukessa osoitettiin TLA+-kieltä apuna käyttäen, että järjestelmä täyttää vaatimukset viestien oikean järjestyksen ja täsmälleen kerran välittämisen suhteen. Lisäksi tutkimuksessa todettiin, että gRPC:tä käyttävä järjestelmä suoriutui Apache Kafka -versiota nopeammin ja oli keskimäärin yli neljä kertaa nopeampi.

Teoksessa (Indrasiri ja Kuruppu 2020) gRPC:n eduiksi mainitaan seuraavat asiat:

- se on tehokas prosessien väliseen viestintään,

- sillä on yksinkertaiset ja hyvin kuvatut palvelurajapinnat ja skeema,
- se on vahvasti tyypitetty,
- sitä ei ole sidottu muihin tekniikoihin, kuten käyttöjärjestelmään tai ohjelmointikieleen,
- siinä on duplex streaming -ominaisuus, joka mahdollistaa palvelimen ja asiakasohjelman lähettää viestejä asynkronisesti pysyvän yhteyden kautta,
- siihen on rakennettu monia hyödyttäviä ominaisuuksia, kuten autentikointi, salaus, aikarajat, pakkaus, kuorman jakaminen ja palvelun löytäminen,
- se integroituu hyvin pilvinatiiveihin ekosysteemeihin ollen osa CNCF:ää (Cloud Native Computing Foundation) ja
- se on kypsä ja laajalti käytössä.

Teos (Indrasiri ja Kuruppu 2020) listaa gRPC:n haittoja seuraavasti:

- se voi olla huono valinta ulospäin tarjottaville palveluille,
- isot muutokset palvelukuvauksessa vaativat monimutkaisen kehitysprosessin ja
- ekosysteemi ei ole kovin laaja.

3.4 Protocol Buffers

Protocol Buffers on Googlen kehittämä avoin tiedon binäärimuotoinen serialisointimenetelmä. Se on tapa serialisoida rakenteista dataa ja käyttökelpoinen sellaisten ohjelmien kehittämiseen, jotka kommunikoivat toistensa kanssa verkon yli tai tallentavat dataa. (Du, Lee ja Kim 2018)

Tekniikkaan kuuluvat rajapintakuvauskieli (interface definition language, IDL), jolla kuvataan jonkin datan rakenne, sekä ohjelma, jolla voidaan generoida kuvauksen perusteella ohjelmakoodi datan muodostamiseen ja parsimiseen. Protocol Buffers lyhennetään usein muotoon ProtoBuf, jota tässäkin tutkielmassa joskus käytetään suomen kielen taivutuksen helpottamiseksi.

ProtoBufin datarakenteita kutsutaan viesteiksi ja sekä ne että palvelut kuvataan .proto-päätteisessä tiedostossa. Proto-tiedosto voidaan kääntää protoc-ohjelmalla kohteena olevalle ohjelmointikielelle ja tuloksena on ohjelmakoodia, jota datarakenteiden lähettäjä tai vastaanottaja voi kutsua. (ks. Du, Lee ja Kim 2018)

ProtoBuf on yleisimmin käytetty kieli gRPC:n palvelukuvauksen tekemiseen. Alla on esimerkki gRPC-palvelukuvauksesta Protocol Buffers -muodossa:

```
// ProductCatalogue.proto
syntax = "proto3";
package eshop;

service ProductCatalogue {
  rpc AddProduct(Product) returns (ProductGUID);
  rpc GetProduct(ProductGUID) returns (Product);
}

message Product {
  string guid = 1;
  string name = 2;
  string description = 3;
  int32 year = 4;
}

message ProductGUID {
  string guid = 1;
}
```

Listauksessa kuvataan verkkokauppaan liittyvä palvelu ProductCatalogue. Alussa kerrotaan käytössä oleva ProtoBuf-versio (proto3) ja kerrotaan pakettinimi (eshop), jolla voidaan estää nimien yhteentörmäykset eri viestityyppien kesken ja joka voi myös vaikuttaa generoitavaan koodiin ohjelmointikielestä riippuen.

Seuraavana kuvataan etäkutsut AddProduct ja GetProduct tuotteiden lisäämiseen ja hakemiseen. Lopuksi kuvataan proseduurien käyttämät viestit Product ja ProductGUID. Viestissä Product on kolme merkkijonomuotoista kenttää ja yksi kokonaisluku. Kenttien kuvauksessa kerrotaan myös kenttänumerot, joiden tulee olla yksilöllisiä. Niitä käytetään kenttien identifioimiseen viestin binäärimuodossa.

Viestin loppuun on mahdollista lisätä kenttiä ilman, että yhteensopivuus vanhaan viestimuotoon särkyy. Tällöin vanhaa viestimuotoa käyttävät palvelut jättävät niille tuntemattomat kentät pois mutta ovat muutoin käyttökelpoisia. (Indrasiri ja Kuruppu 2020)

gRPC:n virtaominaisuus voidaan määritellä proto-tiedostossa näin, kun käytössä on pelkästään palvelimen puolen virta:

rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);

Pelkästään asiakasohjelman käyttämä virta määritellään proto-tiedostossa näin:

rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse);

Kaksisuuntainen virta määritellään proto-tiedostossa näin:

rpc BidiHello(stream HelloRequest) returns (stream HelloResponse);

Proto-tiedosto voidaan kääntää protoc-komennolla halutulle ohjelmointikielelle. Tuloksena saadaan pohjat asiakas- ja palvelinohjelmistolle. Palvelimessa tulee toteuttaa palvelun logiikka ylikirjoittamalla palveluluuranko. Tämän jälkeen gRPC-palvelimen voi käynnistää, jolloin se on valmis kuuntelemaan asiakasohjelmilta tulevia kutsuja ja palauttamaan palveluvastaukset. (Indrasiri ja Kuruppu 2020)

Myös asiakasohjelmaan generoituvat tyngät, joilla palvelua voi kutsua. Asiakastynkä muuttaa kutsut verkon yli tehtäviksi etäkutsuiksi, jotka päätyvät palvelimelle. Koska palvelinkuvaukset ovat riippumattomia ohjelmointikielistä, voidaan asiakas- ja palvelinohjelmat toteuttaa eri ohjelmointikielillä. (Indrasiri ja Kuruppu 2020)

Seuraavissa luvuissa luodaan katsaus olemassa olevaan tutkimukseen, jossa selvitetään tapoja tehostaa JSON- ja REST -menetelmiä sekä vertaillaan gRPC- ja Protocol Buffers -tekniikoita JSON- ja REST-tekniikoihin sekä joihinkin muihin tekniikoihin eri käyttötilanteissa.

3.5 OpenAPI-määritys

Nykyaikaisten, laajojen tietojärjestelmien käyttämien rajapintojen kuvaaminen on tärkeää, jotta käyttäjät voivat löytää tarpeita vastaavat palvelut. Tätä tarvetta varten on kehitetty eri-

laisia metadataformaatteja, jotka pyrkivät tarjoamaan käyttäjän huomioivan ja helppokäyttöisen tavan luoda ja hyödyntää rajapinnan kuvausta. REST-rajapinnoille tällaisia ovat esimerkiksi Open API Specification (OAS), RAML, API Blueprint ja Hydra. Näistä tässä esitellään ensimmäistä. (Golovko 2020)

Lähde Golovko (2020) pitää OAS:ää kaikista lupaavimpana REST-rajapinnan kuvausstandardina, koska se on yksinkertainen muoto kuvausten määrittämiseen ja koska tarjolla on laaja joukko aktiivisen yhteisön kehittämiä toimittajariippumattomia rajapintatyökaluja. Työkalut auttavat rajapintojen kehittämistä ja testaamista liki kaikilla nykyisin käytössä olevilla ohjelmointikielillä. Lisäksi Open API -hanke on avoimen lähdekoodin projekti, jota ovat tukemassa esimerkiksi Google, IBM ja Microsoft. OpenAPI-määritys pohjautuu SmarBear-nimisen yrityksen kehittämään Swagger-ohjelmakehykseen ja -määritykseen, joista OpenAPI eriytyi omaksi Linux Foundationin ylläpitämäksi projektiksi vuonna 2016.

OpenAPI-määrityksellä voidaan ilmaista REST-muotoisen WWW-palvelun rajapinnat sekä ihmis- että koneluettavassa muodossa. Määritystä voidaan käyttää rajapinnan kuvaamiseen, tuottamiseen, käyttämiseen sekä visualisointiin. Monille ohjelmointikielille on tarjolla työkaluja, joilla rajapinnan OpenAPI-määrityksen perusteella voidaan generoida koodipohja rajapinnan toteutukselle niin palvelimen kuin asiakasohjelman puolelle. Kun rajapintojen toteutukset on generoitu määrityksestä, voidaan olla varmoja, että ne ovat määrityksen mukaisia. (Golovko 2020)

OpenAPI-määritys on otettu tähän opinnäytteeseen mukaan tarjoamaan vertailukohta Protocol Buffersin tarjoamalle mahdollisuudelle kuvata palvelujen ja rajapinnan toimintaa. Määrityksiä on mahdollista hyödyntää myös erilaisten rajapintojen yhteistoimintaan, kuten on tehty opinnäytteessä Speth (2017). Siinä on luotu sovelluskehys, josta käytetään nimitystä CLARA (Compose Lovely APIs based on Reusable API-Adapters). Kehykseen on mahdollista liittää rajapintasovittimia. Työssä on esimerkkinä kehitetty sovitin gRPC:n ja REST:in välille niin, että gRPC-rajapintaa käyttävään sovellukseen liitetään REST-sovitin, jolloin sovellus voi hyödyntää toisen sovelluksen REST-muotoista rajapintaa. Tarvittavien ohjelmakoodien generoinnissa hyödynnetään OpenAPI- ja Protocol Buffers -määrityksiä.

OpenAPI:n mukainen rajapinnan kuvaus voidaan tehdä joko JSON- tai YAML-notaatiolla.

Alla on esimerkkilistaus verkkokauppasovelluksen tilauksen hakevan GET-pyynnön määrittelystä YAML-notaatiolla kuvattuna.

```
swagger: ’2.0’
info:
title: Webshop REST API
description: Webshop Microservice REST API.
version: "1.0.0"
basePath: /
paths:
/orders/{id}:
get:
summary: Get order with ID id
description: Returns a specific order with given id.
parameters:
- name: id
in: path
description: order id given in path
required: true
type: integer
format: int32
responses:
200:
description: A specific order
schema:
$ref: ’#/definitions/Order’
default:
description: Unexpected error
schema:
$ref: ’#/definitions/Error’
```

3.6 Viestijonot

Monesti eri lähteistä tulevaa tietoa pitää saada koostettua. Esimerkiksi pankkimaailmassa välittäjä tarvitsee tietoja osake- ja valuuttakurssien muutoksista. Perinteisesti tiedot kerätään tekemällä kyselyjä aina kun tieto on voinut muuttua. Tässä piilee ongelmia: Jos samalla tiedolla on monta tarvitsijaa, jokaisen pitää kysyä tieto erikseen. Toisaalta mahdollisesti jatkuvasti muuttuvaa tietoa pitää kysellä jatkuvasti, jotta muutokset saadaan tuoreeltaan kiinni. (Patana 2014)

Vaihtoehdon tarjoaa tuottaja-kuluttaja-malli, jossa tuottaja laittaa tiedon tarjolle heti kun se on olemassa ja kuluttaja eli tilaaja saa sen saman tien käyttöönsä. Yksi tapa toteuttaa tällainen viestintä on viestiväylän avulla. Advanced Message Queuing Protocol (AMQP) on avoin standardi viestiväylän toteuttamiseen ja RabbitMQ yksi sen toteutus. Tekniikoita on käytetty esimerkiksi Jyväskylän yliopiston Korppi-järjestelmän Payments-verkkomaksuissa (Patana 2014) sekä siitä kehitetyssä JYSOA-ympäristössä (Rinnesalo 2019). Toinen esimerkki viestijonosta on Apache Kafka (Roohitavaf ym. 2019).

Viestijonojen toimintaa hoitaa välittäjä (engl. Message Broker). Se huolehtii viestinvälityksestä siitä kiinnostuneille palveluille eli kuluttajille. AMQP:n jonomallin muut tärkeimmät käsitteet ovat vaihde (exchange), jono (queue) ja liitos (binding). Vaihde on paikka, johon tuottaja julkaisee viestin. Jono taas on paikka, jonne viesti välitetään kuluttajan saataville. Liitos kertoo, mihin jonoon tai jonoihin viesti vaihteesta välitetään. Viestijonoilla toteutettu kommunikaatio on yleensä asynkronista, joskin esimerkiksi RabbitMQ:lla ja sen kirjastolla on mahdollista toteuttaa myös etäproseduurikutsuja, jotka toimivat synkronisesti ja keskeyttävästi. (Videla ja Williams 2012)

Viestijonojen avulla voidaan jakaa järjestelmää pienempiin, riippumattomiin osiin tai jakaa kuormaa usean eri käsittelijän kesken. Haittapuolena viestijonon käyttäminen tekee järjestelmän integraatiotestauksesta haastavampaa. (Patana 2014)

4 Kirjallisuuskatsaus

Kirjallisuuskatsauksen muodostaminen alkoi hakemalla Jyväskylän yliopiston kirjaston JYX-haulla sekä Google Scholar -palvelulla esimerkiksi hakutekstillä ”restful web service performance improvement” sekä yhdistämällä tähän sanoja ”json”, ”microservice” ja”gzip”. Hakutuloksina tuli varsin paljon XML- ja JSON-muotojen nopeusvertailuita. Jonkin verran löytyi myös tutkimuksia, joissa oli sovellettu tekstimuotoisten menetelmien lisäksi Protocol Buffers -serialisointimenetelmää johonkin sovellusalaan. Varsin vähän oli tarjolla tutkimustietoa siitä, miten itse JSON ja REST -menetelmää voisi nopeuttaa vaihtamatta tilalle jotakin muuta standardia tai tekniikkaa.

Kirjallisuuskatsauksen alussa esitellään tapoja nopeuttaa JSON ja REST -menetelmää itsessään. Sitten esitellään tutkimuksia, joissa on mukana myös muita tekniikoita, joista erityisen mielenkiinnon kohteena ovat Protocol Buffers ja gRPC. Yhdessä tutkimuksessa myös selvitetään viestijonotekniikan soveltuvuutta mikropalveluihin.

4.1 JSON ja REST -menetelmän nopeuttaminen

Golovko (2020) on tutkinut JSON-serialisoinnin eri vaihtoehtoja Go-kielessä. Vertailussa oli viisi eri kirjastoa. Näistä encoding/json on eräs kielen peruspaketeista. Muut vertailtavat olivat ffjson, fastjson, easyjson ja json-iterator/go.

Kirjastoista encoding/json ja json-iterator/go käyttävät reflektiota serialisointiin ja deserialisointiin. Kirjastot easyjson ja ffjson käyttivät staattista koodin generointia. Viimeinen kirjasto, fastjson, käytti parsintamenetelmää. Tämä erosi muista merkittävästi siinä, että sillä ei saanut deserialisoinnin tuloksena Go-kielen struct-rakennetta vaan se vain tarjoaa tavan päästä käsiksi JSON-viestin eri osiin. Näin ollen se ei myöskään sisällä mahdollisuutta serialisoida viestiä.

Kokeita tehtiin kolmen eri kokoluokan objekteilla: pieniä alle 512 kilotavun kokoisia, isoja 1-10 megatavun kokoisia sekä todella suuria yli kymmenen megatavun kokoisia. Reflektiokirjastoista standardikirjaston encoding/json oli hieman nopeampi serialisoinnissa kuin json-

iterator/go. Jälkimmäinen puolestaan oli noin neljä kertaa nopeampi deserialisoinnissa.

Staattisen koodigeneroinnin kirjastoista easyjson oli puolestatoista kolmeen kertaan nopeampi kuin ffjson niin serialisoinnissa kuin deserialisoinnissa. Kirjasto ffjson oli deserialisoinnissa nopeudeltaan reflektiokirjastojen puolivälissä mutta serialisoinnissa näitä nopeampi. Isojen objektien serialisoinnissa erityisen hyvin toimi easyjson, kun serialisointia suoritettiin rinnakkaisissa prosesseissa.

Kirjasto fastjson oli pienten objektien tapauksessa deserialisoinnissa - tai kirjaston tapauksessa tarkemmin sanottuna dekoodauksessa - jopa 3600 kertaa nopeampi kuin muut tekniikat. Tekstin parsinta -menetelmä kuitenkin hidastui selvästi isojen objektien tapauksessa: niiden kanssa fastjson oli 2-3 kertaa hitaampi kuin muut kirjastot.

Kokeessa selvitettiin myös kirjastojen varaaman keskusmuistin määrää. Tässä easyjson vei todella suurten objektien serialisoinnissa 2-3 kertaa enemmän muistia kuin muut kirjastot. Kirjasto ffjson puolestaan vei noin kaksinkertaisen määrän muistia todella suurten objektien deserialisoinnissa. Muutoin erot olivat pieniä.

Breje ym. (2018) tutkivat PHP-kielellä ja Lavarel-sovelluskehyksellä toteutetun REST-rajapinnan toimintaa. Vertailussa olivat JSON- ja XML-koodausta käyttävät versiot. Rajapinta sisälsi seuraavia HTTP-pyyntöjä: GET (tiedon pyytäminen), POST (uuden datan lähettäminen), PUT (tiedon päivittäminen) ja DELETE (tiedon poistaminen).

Kokeessa tutkittiin sekä lähetysnopeutta että siirretyn tiedon määrää. DELETE-pyyntöjä lukuun ottamatta JSON-muoto osoittautui sekä nopeammaksi että vähemmän tiedonsiirtokapasiteettia vaativaksi tavaksi ja joissain tapauksissa se oli 30-40% parempi.

Kokeita tehtiin myös käyttäen Gzip-pakkausta. Pakkaaminen pienensi lähetettävän tiedon määrää noin viidennekseen ja siirtonopeus kasvoi 20-40%. Tutkimuksessa suositellaankin pakkauksen käyttöä REST-rajapinnoissa, mikäli palvelin mahdollistaa tämän.

4.2 Serialisointimenetelmien vertailua

Aihkisalo ja Paaso (2011) selvittivät WWW-palvelun objektien serialisointimenetelmien suorituskykyä. Tutkitut tekniikat edustivat JAXB-standardia noudattavia tekniikoita, muita tekstimuotoisia serialisoinnin ja deserialisoinnin menetelmiä sekä joitakin binäärimuodon tekniikoita.

Kokeissa käytettiin erityistä delayTool-nimistä työkalua, joka mahdollisti mittaukset nanosekunnin tarkkuudella. Mittauksissa jätettiin käynnistykseen liittyvät operaatiot huomiotta ja mitattiin vain varsinaiseen serialisointiin ja deserialisointiin kuluva aika.

Käytössä oli kolmen eri tyypin objekteja, joista tyyppi A sisälsi tekstimuotoista ja tyyppi B binäärimuotoista dataa sekä tyyppi C taulukon. Jokaiseen objektiin kuului lisäksi neljä tekstikenttää, joissa jokaisessa oli 10 tavua tekstimuotoista dataa. Tyyppien A ja B objektien koot vaihtelivat 13 eri kokoluokassa välillä 512 tavua - 4 megatavua. Taulukon koko vaihteli kahden potensseina välillä 1–512.

Testejä toistettiin 500 kertaa jokaista objektia ja jokaista kokoluokkaa kohti. Mittausten välissä pidettiin aina sekunnin tauko. Tutkittujen tekniikoiden ja testitapausten suuren määrän vuoksi jokaiselle tekniikalle laskettiin keskiarvo kuvaamaan tehokkuutta eri viestityypeillä.

Testilaitteena oli kannettava, jossa oli kaksiytiminen Intel Core 2 Due 2,53GHz -suoritin ja kaksi gigatavua keskusmuistia. Käyttöjärjestelmänä oli Linux-kernelin versiota 2.6.35 käyttävä Ubuntu ja ohjelmointiympäristönä Javan versio 1.6.0.

Viestityypin A tuloksissa useimmilla tekniikoilla serialisointinopeus oli parempi kuin deserialisointi. Binäärimuotoinen ProtoBuf oli merkittävin poikkeus. Sen deserialisointinopeus oli noin 100 megatavua sekunnissa, mikä oli suurin vertailtavista. Serialisoinnin nopeus ProtoBufilla oli noin 50 megatavua sekunnissa.

Nopein serialisoija oli myös binäärimuotoa käyttävä JAXB FastInfoset, jonka nopeus oli noin 90 megatavua sekunnissa. Sen deserialiserialisointinopeus oli noin 30 megatavua sekunnissa.

Myös muut binääritekniikat olivat varsin nopeita. XML-tyypin serialisoijista paras oli JAXB:n

referenssitoteutus 60 Mt/s -nopeudellaan ja paras deserialisoija JAXB Axiom 39 Mt/s -nopeudellaan. JSON-tekniikoista nopein oli JAXB JacksonJSON, jonka serialisointinopeus oli 35 Mt/s ja deserialisointinopeus 20 Mt/s.

Viestityypillä B ProtoBuf oli selvästi nopein: sen serialisointinopeus oli 180 Mt/s ja deserialisointinopeus 120 Mt/s. Binäärimuotoinen Hessian pääsi serialisoinnissa tasolle 90 Mt/s ja deserialisoinnissa tasolle 30 Mt/s.

Muista binärityyppisistä tekniikoista JAXB FastInfoset ja JAXB JacksonSMILE pääsivät noin 30 megatavun sekuntivauhtiin, mikä oli parhaiden tekstimuotoisten tekniikoiden, kuten JAXB XML:n, tasoa.

Taulukkomuotoisella viestityypillä C binärityyppisten nopeus oli pääosin välillä 1–4 megatavua sekunnissa. Useat XML-tekniikat olivat binärityyppisiä parempia. Nopein oli JiBX XPP, jonka deserialisointinopeus oli 17 Mt/s ja serialisointinopeus 6 Mt/s. JSON-tekniikat jäivät tasolle 1–2 megatavua sekunnissa.

Tutkimuksessa todetaan, että pelkkä serialisoinnin ja deserialisoinnin nopeus ei etenkään WWW-palvelun tapauksessa ole aina ratkaisevaa. Suuri vaikutus voi olla myös serialisoinnin tuloksena tulevan datan määrällä, sillä tämä data pitää siirtää verkkoyhteyden kautta palvelimen ja asiakasohjelman välillä. Useimmiten binärityypin tekniikoissa siirrettävän datan määrä on pienin. JSON puolestaan säästää tilaa XML:ään verrattuna.

4.3 Mobiililaitteet

Sumaray ja Makki (2012) tutkivat datan serialisointimenetelmien suorituskykyä mobiililaitteissa, joissa resurssit ovat vähäiset ja tiedonsiirtokapasiteetti rajallinen. Tutkimuksessa mukana oli neljä serialisointimenetelmää: tekstimuotoiset XML ja JSON sekä binäärimuotoiset Thrift ja ProtoBuf. Tutkimus kattoi kolme näkökulmaa: datan koko, serialisoinnin nopeus ja menetelmän käytön helppous.

Tutkimuksessa keskityttiin Android-käyttöjärjestelmään ja toteutettiin sovellus, jolla pystyttiin testaamaan serialisoinnin ja deserialisoinnin nopeutta. Lisäksi sovellus kykeni tallentamaan datan ja tulokset ja sillä saattoi helposti toistaa testejä useita kertaa.

Sovelluksessa datamallina oli kaksi Java-luokkaa. Toinen edusti kirjaa ja koostui useasta tekstikentästä sekä numeerisista ja totuusarvotyyppisistä kentistä. Toinen luokka edusti videota ja siinä oli vähän tekstimuotoisia kenttiä sekä paljon numeerista dataa.

XML-serialisointiin käytettiin Simple-kehystä ja JSON:in kanssa Streaming API:a. ProtoBufista käytössä oli versio 2.4.1 ja Thriftistä versio 0.6.1.

Datan koot kirjaobjektilla vaihtelivat välillä 687–873 tavua. Binäärimuotoiset tekniikat olivat hieman tekstimuotoisia parempia ja järjestys tiiveimmästä eniten tilaa vievään oli: ProtoBuf, Thrift, JSON, XML.

Video-objektin kanssa suhteelliset erot olivat suurempia. Järjestys oli sama, ja ProtoBuf vei 59 tavua ja JSON 139 tavua. Thrift ja XML vaativat liki kaksinkertaisen määrän tilaa luokan toiseen tekniikkaan verrattuna.

Nopeudeltaan XML osoittautui selvästi heikoimmaksi. Kirjaobjektin serialisointi vei binäärimuotoisilla tekniikoilla noin 2,3 millisekuntia ja JSON:illa liki kaksinkertaisen ajan. XML vaati kymmenkertaisen ajan. Deserialisoinnissa ProtoBuf vei noin 0,3 ms, Thrift noin kaksinkertaisen, JSON noin nelinkertaisen ja XML yli 20-kertaisen ajan.

Video-objektin nopeusvertailussa järjestys oli sama. JSON:ia lukuun ottamatta numeerisen datan serialisointiin kului noin 20% lyhempi aika kuin tekstimuotoisen kirjaobjektin. Molempien viestityyppien serialisointi JSON:illa vei suunnilleen saman ajan. Deserialisoinnissa ProtoBuf vei 0,2 ms, Thift noin 1,5-kertaisen, JSON noin 6,5-kertaisen ja XML yli 20-kertaisen ajan.

Käytettävyyden osalta tutkimuksessa tehtiin muutama tekstimuotoisia XML- ja JSON-tekniikoita puoltava huomio: Ne ovat ihmisen luettavissa, joten niitä voi usein olla helpompi käsitellä. Toisaalta useimmat ohjelmointiympäristöt tarjoavat suoraan tuen XML- ja JSON-muodoille. Tutkitut binääritekniikat vaativat erilliset viestin skeeman kuvaavat tiedostot: ProtoBufissa .proto ja Thriftissä .thrift. Tätä myötä ne vaativat myös erityisen tuen ohjelmointikieleltä tai kirjaston käyttöä.

4.4 Esineiden Internet

Popić ym. (2016) tutkivat tiedonsiirron toteutusta esineiden Internetissä eli IoT-kommunikaatiossa JSON-, BSON- ja ProtoBuf-tekniikoita käyttäen. Tutkimuskohteena oli ajoneuvojen seurantalaite, joka käytti suljettua lähdekoodia ja jonka tarkkaa toimintaa ei artikkelissa voitu kertoa.

Järjestelmän omat viestit olivat binäärimuotoisia. Viestejä oli neljää tyyppiä: pyyntöjä, joilla haetaan dataa laitteesta, komentoja, joilla tallennetaan tai päivitetään dataa laitteeseen, vastauksia ja statusviestejä, joita laite palauttaa, sekä hyväksynnästä ja hylkäämisestä kertovia ja yhteyden ylläpidossa käytettäviä viestejä.

Kokeessa luettiin yli 50 000 oikeista seurantalaitteista tullutta viestiä. 46% viesteistä oli hyväksynnästä kertovia ack-viestejä. Pienen kokonsa vuoksi nämä veivät kuitenkin vain 11% kaikkien viestien vaatimasta tallennustilasta. Vastausviestejä oli määrällisesti 48%, mutta kokonaistilasta nämä veivät 86%.

Viestit tallennettiin lokitiedostoon ja muutettiin sitten kolmeen testissä olleeseen viestimuotoon. Sekä muunnetut että alkuperäiset viestit tallennettiin muistinvaraiseen tietokantaan, josta saatiin kokojen mittausten ja yhteenvedon jälkeen tulostaulukko. Alkuperäisessä binäärimuodossa viestit veivät 2,75 Mt, ProtoBuf-muodossa 6,84 Mt, BSON-muodossa 34,61 Mt ja JSON-muodossa 40,49 Mt.

Komentotyyppisissä viesteissä oli mahdollista käyttää ProtoBufin enum-ominaisuutta, mikä pienensi tällaisten ProtoBuf-viestien kokoa merkittävästi BSON-viesteihin verrattuna. Alkuperäisessä binäärimuodossa komentoviesteissä käytettiin vakiopituisia merkkijonoja, joissa saattoi olla redundanssia. Tämän ansiosta ProtoBuf-muotoiset komentoviestit eivät vieneet juurikaan enempää tilaa kuin alkuperäiset binääriviestit.

Johtopäätöksenä tutkimuksessa todettiin, että JSON- ja BSON-muotoihin verrattuna ProtoBuf on selvästi tehokkaampi viestimuoto. Toisaalta alkuperäinen binäärimuoto oli yli puolet tiiviimpi. Niinpä tehokkuuden kannalta tilanteeseen sovitettu binäärimuoto voi olla ProtoBufia parempi, mutta tällöin eri laitteiden yhteistoiminta on hankalampi toteuttaa. Tutkimuksessa ei selvitetty eri tekniikoiden serialisoinnin aikavaativuutta.

4.5 Twitter

Wibowo (2011) tutki, voisiko ProtoBuf nopeuttaa Twitter-mikrobloggauspalvelun rajapinto-ja. Tutkimuksessa selvitettiin tiedonsiirtoa palvelimen ja Twitterin Internet-selaimessa toimivan asiakasohjelman välillä.

Tarkastelu rajoitettiin vain yhteen toimintoon: palvelimen vastaukseen, kun asiakasohjelma pyytää tilatietojen päivitystä. Saatava vastausviesti on JSON-muotoinen ja koostuu kahdesta osasta: käyttäjätiedoista sekä varsinaisista tilatiedoista. Tutkimuksessa luotiin JSON-viestiä vastaava ProtoBuf-formaatti ja tutkittiin tiedon serialisoinnin nopeutta sekä tarvittavan tallennustilan määrää.

Vertailu tehtiin järjestelmässä, jossa oli Intel Atom N280 1.66 GHz -suoritin, gigatavu keskusmuistia ja Ubuntu-käyttöjärjestelmä. Käytössä oli Java-ohjelmointikieli ja OpenJDK 1.6 -virtuaalikone. Kummallakin tekniikalla serialisoitiin miljoona Twitter-viestiä niin, että ne tallennettiin tiedostoihin. Operaatiosta laskettiin sekä käytetty aika että tarvittu tallennuskapasiteetti.

Tulosten perusteella yhden viestin serialisointiin kului JSON-tekniikalla hieman yli 10 sekuntia ja ProtoBuf-tekniikalla hieman alle 7 sekuntia. Tallennustilaa JSON-viesti vei noin 1800 tavua ja ProtoBuf-viesti noin 600.

4.6 Akka.NET

Littorin, Reijnst ja Sörman Lundgren (2015) tutkivat Akka.NET-sovelluskehyksen serialisointimenetelmiä. Vertailussa olivat Json.NET (NewtonSoftJsonSerializer), TypeSerializer ja ProtoBuf. Tutkimuksessa verrattiin serialisointi- ja deserialisointiaikoja sekä tiedonsiirron nopeutta ja sen vaatimaa kapasiteettia kaikkiaan kahdessakymmenessäneljässä eri testissä.

Näistä ProtoBuf todettiin parhaaksi kahdessakymmenessäkahdessa eri testissä. Se tarjosi myös kaikissa kolmessa testitapauksessa parhaan pakkaussuhteen. TypeSerializer pärjäsi huonommin kuin tutkijat olivat odottaneet ja se oli monissa tapauksissa hitaampi kuin NewtonSoftJsonSerializer.

Sekä lähettävä että vastaanottava ohjelma oli tehty Akka.NET-kehyksellä. Käytössä oli kolme erilaista viestiä. Ensimmäinen viesti oli pieni merkkijono-objekti, jossa oli Akka.NET:in yhteydenmuodostuksessa käytettävä komento. Toinen viesti koostui objektista, jonka sisällä oli kaksi merkkijono-, kokonaisluku-, liukuluku- ja päivämäärätyypin dataa sisältävää objektia. Kolmas viesti koostui myös useista tietotyypeistä mutta sisälsi enemmän merkkijonomuotoista dataa.

Kokeissa kaikki viestit lähetettiin neljään otteeseen niin, että ensimmäisellä kerralla viestit lähetettiin sata kertaa, toisella tuhat, kolmannella 10 000 ja viimeisellä 100 000 kertaa. Viestien lähettämiseen kuluneet ajat ja niiden vaatima tallennustila kirjattiin ylös tekstitiedostoon.

Testit toteutettiin niin, että lähetys ja vastaanotto olivat eri tietokoneilla. Koneet olivat samassa lähiverkossa yhdistettyinä reitittimen ja gigabitin sekuntinopeuteen pystyvän CAT6-johdon välityksellä. Koneissa oli Windows 8.1 -käyttöjärjestelmät, Intel i5 -suorittimet ja toisessa kahdeksan ja toisessa neljä gigatavua keskusmuistia. Molemmissa oli nopeat SSD-levyt.

Kokeiden perusteella kaikilla viestityypeillä ProtoBuf vei vähiten tallennustilaa, Typeserializer toiseksi vähiten ja Json.NET eniten. Viestityypeillä 1 ja 3 erot olivat pieniä, mutta viestityypillä 2 Typeserializer-viestit veivät noin kaksinkertaisen ja Json.NET 2,5-kertaisen tilan ProtoBufiin verrattuna.

Serialisoinnin aikavaatimuksen osalta ProtoBuf oli useimmissa testeissä nopein ja vei noin kolmasosan muiden vaatimasta ajasta. Kaikista suurimmalla viestimäärällä ja tyypin 1 viesteillä Typeserializer kuitenkin oli ProtoBufia noin 25% nopeampi.

Deserialisoinnissa tulokset olivat saman suuntaisia, ja ProtoBuf vei useissa kokeissa noin puolet tai kolmasosan muiden tekniikoiden vaatimasta ajasta. Viestityypillä 1 erot olivat kuitenkin pienempiä isojen viestimäärien tapauksessa. 10 000 viestin kohdalla Typeserializer oli hieman ProtoBufia nopeampi mutta määrän kymmenkertaistuessa ProtoBuf oli taas nopeampi.

Erillisten serialisointiin ja deserialisointiin kuluvien aikojen lisäksi kokeessa mitattiin koko tiedonsiirtoon kulunut aika koneiden välillä. Tämä kattoi serialisoinnin, lähetyksen tietoverkossa ja deserialisoinnin.

Koko tiedonsiirtoon kuluneen ajan mittaustulos oli yllättävä, sillä ProtoBuf oli merkittävästi muita nopeampi vain kaikkein isoimmalla viestimäärällä viestityyppien 2 ja 3 kanssa. Joissain testeissä ProtoBuf oli aavistuksen hitaampi kuin muut.

Tutkimuksessa ei osattu varmasti selittää ilmiötä, mutta tutkijat löysivät kaksi mahdollista selittävää tekijää: Ensinnäkin käytetyt tietokoneet eivät olleet identtisiä vaan deserialisoinnista vastannut kone oli hitaampi. Toisaalta tutkijat arvelivat, että joko Akka.NET-kehys tai jokin muu koeympäristön komponentti voisi rajoittaa viestien lähettämisen suorituskykyä asiakasohjelmien välillä.

4.7 RabbitMQ

Hong, Sik Yang ja Kim (2018) toteuttivat kaksi versiota keskustelupalstana toimivasta verkkosovelluksesta. Ensimmäisessä versiossa käytössä olivat REST-rajapinnat ja rajapintayhdyskäytävä, jonka tehtävänä oli välittää viestit relevanteille mikropalveluille. Toisessa versiossa HTML-käyttöliittymästä tulevat pyynnöt päätyivät yhdyskäytävän sijasta AMQP-pohjaiselle RabbitMQ-viestijonolle, joka välitti viestit RPC-tyylisesti mikropalveluille ja paluuviestit takaisin kutsujalle. Viestiväylä tarjosi viesteille puskurin siltä varalta, että pyyntöjä tulee kerralla paljon eikä kaikkiin ehditä heti vastaamaan.

Kokeessa todettiin, että kun pyyntöjen määrä oli pieni, REST-mallinen keskustelupalsta toimi nopeammin. Kun yhtäaikaisten viestien määrä kasvoi, REST-versio alkoi hidastua ja pyyntöjen määrän kasvaessa se saattoi kokonaan lakata vastaamasta. RabbitMQ-versio oli pienillä viestimäärillä hitaampi, mutta määrän kasvaessa sen nopeus nousi REST-versiota paremmaksi. Merkittävin etu viestijonoa hyödyntävässä versiossa oli se, että suurillakaan viestimäärillä järjestelmä ei mennyt täysin toimimattomaan tilaan vaan viestijono toimii puskurina.

Shafabakhsh (2020) puolestaan vertaili kolmea mikropalveluista koostuvaa verkkopalvelua, joissa mobiililaitteella käytetään verkkokauppaa. Yhteys mobiililaitteesta rajapintayhdyskäytävään tapahtui aina REST-menetelmällä ja JSON-muotoa käyttäen. Rajapintayhdyskäytävästä tiedot välittyivät mikropalveluille. Mikropalvelujen väliset yhteydet oli toteutettu kolmella eri tekniikalla: REST, gRPC ja RabbitMQ. REST ja gRPC olivat synkroninen tiedonsiirtotapa, kun taas RabbitMQ asynkroninen, jossa viestit kulkivat välittäjän (broker)

kautta. Kaikki palvelut oli kehitetty NodeJs-ohjelmointikielellä ja palvelut oli laitettu käyttöön Microsoftin Azure Kubernetes Cluster Service -pilvipalveluun Kubernetes-työkalun avulla.

Kokeessa selvitettiin eri tekniikoiden nopeutta ja saatavuutta erilaisilla käyttäjämäärillä. Testityökaluna käytössä oli Apache JMeter, jolla on mahdollista luoda virtuaalisia käyttäjiä, jotka aiheuttavat kuormaa palveluille. Kokeita tehtiin kolmessa eri mittaluokassa niin, että ensin yhtäaikaisia käyttäjiä oli 50, sitten sata ja lopulta 200. Jokaisella testikerralla ajoa jatkettiin kolmen minuutin ajan.

Kun yhtäaikaisia käyttäjiä oli 50, nopein oli gRPC, jossa keskimääräinen vastausaika oli 0,22 sekuntia. REST jäi tästä 0,01 sekuntia ja RabbitMQ 0,04 sekuntia. Myös sadalla käyttäjällä gRPC oli nopein ja keskimääräinen vastausaika oli 3,3 sekuntia. Sekä REST että RabbitMQ olivat 0,2 sekuntia hitaampia. Kun käyttäjiä oli 200, RabbitMQ oli nopein, 6,4 sekuntia, gRPC toiseksi nopein, 7,2 sekuntia, ja REST hitain, 7,8 sekuntia.

Saatavuuden laskemisessa oli käytetty artikkelista Johnson ym. (2014) otettua kaavaa 3:

$$Availability = \frac{MTTF}{MTTF + MTTR}$$

Kuvio 3. Saatavuus (availability) = MTTF/(MTTF+MTTR) Johnson ym. (2014)

Kaavassa MTTF tulee englannin sanoista ”Mean Time to Failure” ja MTR sanoista ”Mean Time to Recovery”. Ensimmäinen kertoo keskimääräisen ajan ilmenneiden häiriöiden välillä ja jälkimmäinen keskimääräisen ajan, jossa järjestelmä palautuu. Saatavuutta testattaessa JMeter-työkalulla simuloitiin kahtasataa yhtäaikaista käyttäjää. Tässä kokeessa ei ollut aikarajaa vaan odotettiin virhetilanteen ilmaantumista. Kokeen perusteella paras MTTF-arvo oli RabbitMQ:lla, 430 sekuntia, toiseksi paras gRPC:llä, 290 sekuntia, ja huonoin REST:illä, 275 sekuntia. Virheen jälkeen Kubernetes käynnisti palvelun uudelleen ja näin saatiin MTTR-arvo. Siinä REST oli paras, 19 sekuntia ja gRPC toinen, 21 sekuntia. Koska RabbitMQ-versiossa pitää käynnistää uudelleen myös RabbitMQ:n välittäjäpalvelu, oli se hitain 29 sekunnilla. Kaavalla 3 lasketut saatavuusarvot olivat: RabbitMQ 0,95 ja sekä gRPC että REST 0,93.

Tämän jälkeen arvioitiin laadullisina suureina skaalautuvuutta ja kompleksisuutta. Kuten

nopeustesteistäkin nähtiin, RabbitMQ vaikutti toimivan hyvin, kun kuorma kasvaa. Asynkronisena tekniikkana RabbitMQ:n etuna on lisäksi mahdollisuus ajaa viestinvälittäjää klusterissa, jossa on useampi noodi. Näin kuormaa voidaan tarvittaessa jakaa. Synkronisten tekniikoiden gRPC ja REST skaalaaminen on hankalampaa. Tämä johtuu siitä, että palvelut ovat suoraan yhteydessä toisiinsa ja skaalautuminen tarkoittaa jokaisen palvelun monistamista. Kompleksisuutta arvioitiin koodirivien ja toimintopisteiden määrällä sekä testaamisen haastavuudella. REST-tekniikalla rivi- ja toimintopistemäärät olivat pienimmät. gRPC-käytti Proto-Buf-viestimuotoa, joka tarvitsee skeeman .proto-tiedostossa, mikä lisää kompleksisuutta. Testattavuuden kannalta gRPC ja RabbitMQ tuovat haasteita, sillä niissä pitää muodostaa yhteys. RabbitMQ on vielä hieman hankalampi, sillä siinä viestit eivät kulje suoraan palvelulta toiselle vaan välittäjän kautta, mikä lisää yhden lisäelementin testaamiseen.

5 Käytännönosuus

Tutkielman käytännönosuudessa laaditaan Go-kielellä sekä REST ja JSON että gRPC ja Protocol Buffers -tekniikoilla mikropalveluympäristöt OPC UA -protokollalla luetun IoT-datan tiedonsiirtoon. Tämän jälkeen kerätään joukko OPC UA -viestejä, jotka lähetetään kumpaakin tekniikkaa käyttäen, ja mitataan lähetykseen kuluva aika. Lähettämistä tutkitaan sekä samalla koneella toimivien mikropalvelujen tapauksessa että verkon yli kommunikoivien mikropalvelujen tapauksessa.

5.1 Tutkimusasetelma

Kokeessa testattiin OPC UA -protokollalla luetun IoT-datan siirtoa gRPC- ja JSON-menetelmillä. Open Platform Communications Unified Architecture (OPC UA) on koneiden välisen kommunikaation protokolla jota käytetään teollisuusautomaatioon. Sen on kehittänyt OPC Foundation. Protokollaa voidaan käyttää esimerkiksi mittalaitteilta tulevan sensoridatan lukemiseen. KEPServerEX puolestaan on eräs kaupallinen tuote, joka osaa hyödyntää OPC UA -protokollaa. Se tarjoaa yhden lähteen teollisuusautomaation datalle, jota saatetaan lukea usealta eri laitteelta. (“KEPServerEX - Product Overview” 2020)

Testin alussa luettiin tietoa OPC UA -protokollalla KEPServerEX-palvelimelta kahteen Go-kielen kanavaan niin, että molempiin kanaviin tuli sama data. Tämän jälkeen viestit lähetettiin testikertojen välillä vaihtelevasti joko ensin gRPC-rajapintaan ja seuraavana JSON-rajapintaan tai päinvastaisessa järjestyksessä.

Lähetykseen kuului myös datan serialisointivaihe. Vastaanottavat JSON- ja gRPC-rajapinnat deserialisoivat viestit ja tulostivat niiden tietosisällön ruudulle sekä tämän jälkeen lähettivät vastausviestinä kuittauksen.

Koko lähetysoperaatiosta mitattiin aika Go-kielen time-kirjaston avulla niin, että mittaus alkoi siitä hetkestä, kun ensimmäinen viesti luettiin Go:n kanavasta, ja päättyi, kun viimeisenkin kanavassa olleen viestin lähetyskuittaus saatiin. Lähetys olisi voinut myös epäonnistua, mutta kokeissa ei ilmennyt yhtään virhetilannetta.

Viestit luettiin KEPServerEX-ohjelmasta tekemällä tilaus 25 simulaationoodiin. Alla on listaus noodeista sekä niiden tietotyypeistä:

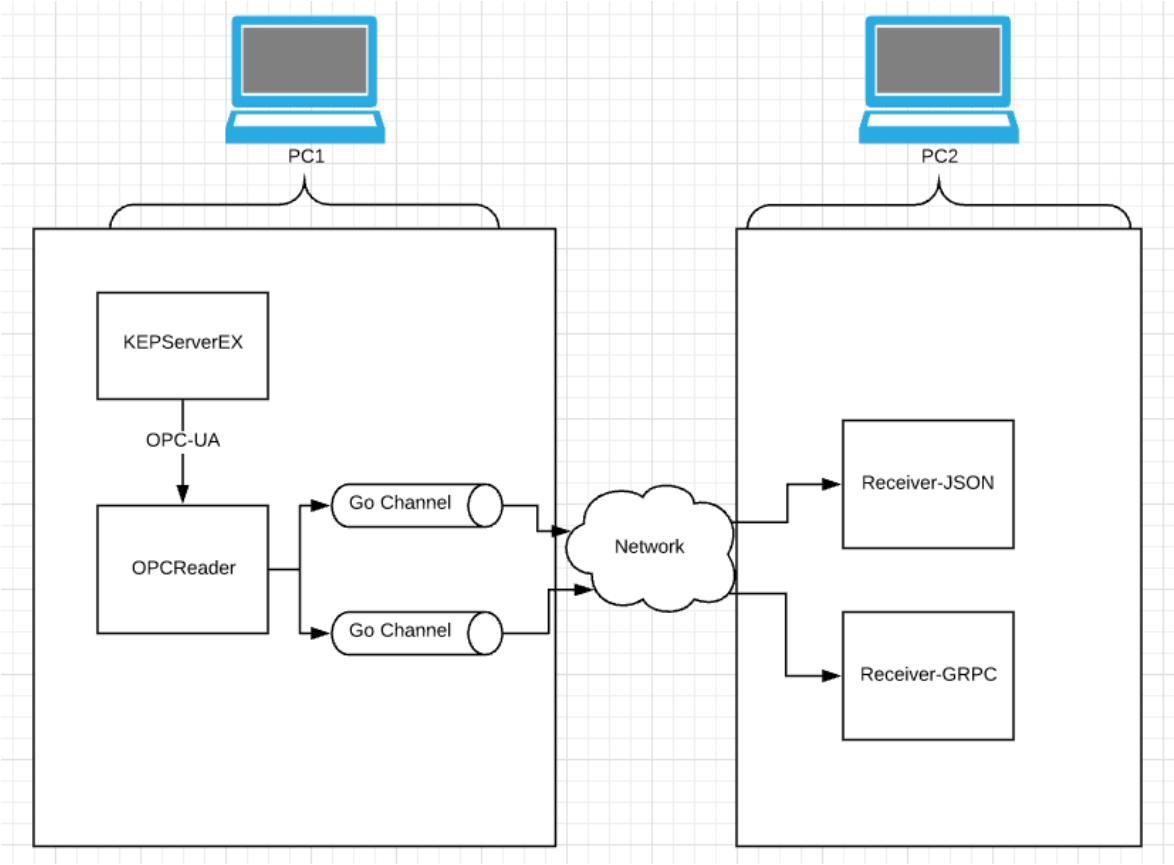
```
Ramp1 (Long)
Ramp2 (Float)
Ramp3 (Word)
Ramp4 (Long)
Ramp5 (LLong)
Ramp6 (QWord)
Ramp7 (LLong)
Ramp8 (Double)
Random1 (Short)
Random2 (Word)
Random3 (Long)
Random4 (Long)
Random5 (Long)
Random6 (LLong)
Random7 (QWord)
Random8 (QWord)
Sine1 (Float)
Sine2 (Float)
Sine3 (Float)
Sine4 (Float)
KEPTimestamp (Date)
User1 (String)
User2 (Float)
User3 (Boolean)
User4 (String)
```

Useimmat noodit olivat numeerisia. Tämä kuvaa hyvin todellista tilannetta, jossa luetaan mittausarvoja antureista. Mukana oli myös kaksi tekstimuotoista noodia sekä yksi aikaleiman sisältävä noodi.

Simulaationoodeihin KEPServerEX-ohjelma generoi joko satunnaista tai annetun funktion mukaisesti vaihtelevaa dataa tietyin väliajoin. Kun tilaus luettiin, palautti palvelin kaikki ne noodit arvoineen, jotka olivat muuttuneet edellisen lukukerran jälkeen.

Tilaukset luettiin testeissä sata kertaa. Keskimäärin yhdellä kerralla luettiin noin kolmetoista eri noodia, eli sadalla lukukerralla luettiin yhteensä noin 1300 eri arvoa. Kokeissa sadan tilauskerran viestien lähetykset tehtiin molemmilla metodeilla yhteensä sata kertaa, jolloin kaikkiaan luettavaksi tuli noin 130 000 viestiä.

Lähetystapojen järjestystä vaihdeltiin jatkuvasti. Viestien lukemisen jälkeen pidettiin aina viiden sekunnin tauko, joka varmisti, että lukemiseen käytetyt resurssit olivat vapautuneet. Tämän jälkeen viestit lähetettiin ensin metodilla 1 ja sitten metodilla 2. Eri lähetysmuotojen välissä pidettiin viiden sekunnin tauko varmistamassa, että kaikki edellisen lähetyksen resurssit olivat vapautuneet. Koejärjestelmän arkkitehtuuria selventää kuvio 4.



Kuvio 4. Kuva arkkitehtuurista

Ensimmäisellä koekerralla kaikki ohjelmat toimivat samalla tietokoneella (PC1). Toisella koekerralla Windowsissa toimiva KEPServerEX-ohjelma sekä Go-kielellä tehty OPCReader-ohjelma toimivat tietokoneella PC1 ja Go-kielellä tehdyt vastaanottavat rajapinnat Receiver-JSON ja Receiver-GRPC toisella tietokoneella PC2. Koneet oli yhdistetty langattomalla ad-hoc-verkolla niin, että langattoman verkon muodosti Jolla-matkapuhelin.

Yhteystapa oli varsin epätavallinen. Tällaiset järjestelmät ovat kuitenkin usein huonojen verkkoyhteyksien takana ja välillä pitkiä aikoja kokonaan katveessa. Niinpä viestien varastointi ja sitten niiden lähettäminen isona joukkona tarjolla olevan rajoitetun yhteyden kautta voivat olla hyvin tavallinen tilanne.

Mainittakoon myös, että viestit voitaisiin välittää myös suoraan OPC UA -protokollaa käyttäen. Esimerkiksi pilvipalvelussa toimiva mikropalvelu voisi olla suoraan yhteydessä KEPServerEX-ohjelman kanssa. Järjestelmä on kuitenkin suunniteltu tilanteeseen, jossa verkkoyhteyttä ei kaiken aikaa ole. Tällöin on parempi lukea viestit OPC UA -protokollalla ja tallentaa ne koneelle lokaalisti esimerkiksi dokumenttikantaan. Kokeessa viestit tallennettiin lyhytaikaisesti Go-kielen kanavaan. Kun viestejä on luettu ja kun verkkoyhteys on olemassa, viestit lähetetään vastaanottavalle palvelulle jotakin muuta protokollaa ja viestinvälitysmekanismia käyttäen.

Go-kielellä tehdyt ohjelmat oli käännetty PC2-koneella. Windows-koneeseen käännettiin ajettava EXE-tiedosto antamalla go build -komennolle ympäristömuuttuja GOOS=windows. Käytetty Go-versio oli 1.13.1. Go-kielen Protocol Buffer -moduulista käytössä oli versio v1.4.0-rc.4.

Käytetyt tietokoneet olivat seuraavat:

PC1

- Lenovo C19006W -kannettava
- Windows 10 Enterprise 64-bit -käyttöjärjestelmä
- (build 18262.720)
- Intel Core i7-4810MQ -suoritin (4 ydintä)
- 32 Gt keskusmuistia

- SSD-levy
- WLAN

PC2

- Lenovo P50 -kannettava
- Ubuntu 18.04.4 LTS -käyttöjärjestelmä
- (Linux-kernel 4.15.0-96)
- Intel Core i7-6700HQ -suoritin (4 ydintä)
- 32 Gt keskusmuistia
- SSD-levy
- WLAN

Protobuf-viesteissä käytettiin seuraavaa protomääritystä:

```
message OPCMessage {
    string guid = 1;
    enum ValueType {
        UNKNOWN = 0;
        TYPES_BOOLEAN = 1;
        TYPES_INT = 2;
        TYPES_FLOAT = 3;
        TYPES_DATETIME = 4;
        TYPES_STRING = 5;
    }
    ValueType value_type = 2;
    string node_id = 3;
    bool value_boolean = 4;
    int32 value_int = 5;
    float value_float = 6;
    google.protobuf.Timestamp value_datetime = 7;
    string value_string = 8;
}
```

JSON-muotoisessa lähetyksessä käytetyt viestit olivat tällaista muotoa:

```
{
  "guid": "guid123",
  "valuetype": 3,
  "nodeid": "ns=2;s=Simulation Examples.Functions.Ramp1",
  "valueboolean": false,
  "valueint": 0,
  "valuefloat": 23.4,
  "valuedatetime": null,
  "valuestring": ""
}
```

Rakenteissa oli huomioitu se, että luettava data sisältää eri tietotyyppejä. Eri tietotyypeille on rakenteessa omat kenttänsä. Lisäksi on kenttä, joka ilmoittaa viestissä olevan arvon tietotyypin. Näin samalla viestirakenteella voidaan välittää eri tietotyyppiä olevia arvoja.

REST-versiota varten laadittiin myös OpenAPI-määrityksen mukainen kuvaus YAML-kielellä. Sen avulla sekä valittuun ohjelmointikieleen soveltuvalla työkalulla on mahdollista generoida tyngät palvelin- ja asiakasohjelmille ja näin varmistaa, että rajapinnat vastaavat määritystä. Go-kielelle OpenAPI:n eli entiseltä nimeltään Swaggerin ominaisuudet tarjoaa go-swagger-niminen kirjasto. Kirjasto on kuitenkin vielä kehitysvaiheessa ja sillä generoitu asiakasohjelma ei täysin toiminut. Niinpä kokeessa käytetyt REST-rajapinnat on kirjoitettu ilman koodigeneraattorin apua. Alla on listaus OpenAPI-rajapintakuvauksista YAML-notaatiolla.

```
consumes:
- application/opc-message.v1+json
definitions:
  opcmessage:
    type: object
    required:
      - guid
```

```
    - valueType
  properties:
    guid:
      type: string
      minLength: 1
    valueType:
      type: string
      enum: [UNKNOWN, TYPES_BOOLEAN, TYPES_INT,
             TYPES_FLOAT, TYPES_DATETIME, TYPES_STRING]
    nodeID:
      type: string
      minLength: 1
    valueBoolean:
      type: boolean
    valueInt:
      type: integer
      format: int64
    valueFloat:
      type: number
      format: double
    valueDateTime:
      type: string
      format: date-time
    valueString:
      type: string
opcresponse:
  type: object
  required:
    - guid
    - status
  properties:
```

```
      guid:
        type: string
        minLength: 1
      status:
        type: string
        enum: [TRANSFER_OK, TRANSFER_ERROR]
info:
  description: Receiver for OPC UA messages converted to JSON
  title: OPC Receiver
  version: 1.0.0
paths:
  /:
    post:
      tags:
        - opcmessage
      operationId: storeOne
      parameters:
        - name: body
          in: body
          schema:
            $ref: "#/definitions/opcmessage"
      responses:
        201:
          description: StoreResponse
          schema:
            $ref: "#/definitions/opcresponse"
produces:
- application/opc-response.v1+json
schemes:
- http
swagger: "2.0"
```

5.2 Tulokset

Ensimmäisessä kokeessa kaikki ohjelmat toimivat samassa Windows-tietokoneessa. KEPServerEX-ohjelmasta luettiin tilausta sata kertaa ja kaikkiaan luettiin 129 895 viestiä.

Viestien lähetys vei JSON-menetelmällä kaikkiaan 37,076 sekuntia. gRPC-menetelmällä aikaa kului 39,784 sekuntia. Samalla koneella toimivaan rajapintaan lähetyksessä erot olivat pieniä ja lähetys tapahtui hyvin nopeasti. gRPC osoittautui keskimäärin 7,3% hitaammaksi tavaksi. Yhden viestin keskimääräiseksi lähetysajaksi tuli JSON-tekniikalla noin 285 mikrosekuntia ja gRPC-tekniikalla noin 306 mikrosekuntia.

JSON oli nopeampi lähes jokaisella testiajolla. Vain kuudella ajolla gRPC oli nopeampi ja yhdellä molempien ajaksi mitattiin sama aika 479 millisekuntia. Isoin absoluuttinen ero tekniikoiden välillä samalla ajokerralla oli JSON-tekniikan eduksi 138 mikrosekuntia viestiä kohden laskettuna. Kerroilla, joilla gRPC oli nopeampi, isoin ero oli 24 mikrosekuntia viestiä kohden.

Toisessa kokeessa vain KEPServerEX-palvelin sekä viestien luennasta ja lähetyksestä vastannut OPCReader-ohjelma toimivat Windows-tietokoneella. JSON- ja gRPC-rajapinnat olivat toisella tietokoneella, joka käytti Ubuntu-käyttöjärjestelmää. Koneet olivat samassa langattomassa ad-hoc-verkossa, joka muodostettiin Jolla-matkapuhelimella. Yhteys koneelta toiselle kulki yksityisverkon IP-osoitteen perusteella. KEPServerEX-ohjelmasta luettiin tilausta sata kertaa ja kaikkiaan luettiin 129 020 viestiä.

Viestien lähetys vei JSON-menetelmällä kaikkiaan 19 minuuttia 57 sekuntia. gRPC-menetelmällä aikaa kului 10 minuuttia 2 sekuntia. Kun ohjelmat toimivat eri laitteilla, osoittautui gRPC 49,7% nopeammaksi menetelmäksi. Yhden viestin keskimääräiseksi lähetysajaksi tuli JSON-tekniikalla 9,28 millisekuntia ja gRPC-tekniikalla 4,67 millisekuntia.

gRPC oli nopein jokaisella testiajolla. Yksittäisellä ajokerralla pienin ero oli 1,99 millisekuntia viestiä kohden ja suurin ero 8,71 millisekuntia viestiä kohden.

Tulosten perusteella voidaan sanoa, että gRPC-tekniikka toimii hyvin OPC UA -protokollalla luettujen IoT-viestien lähettämiseen eri koneilla toimivien prosessien välillä. Sen sijaan JSON osoittautui aavistuksen nopeammaksi tapauksessa, jossa mikropalvelut toimivat samassa ko-

neessa. Paikallisten prosessien osalta tulos oli hieman yllättävä, sillä ennakkoarviona oli, että gRPC olisi nopeampi myös tällöin. gRPC:n huono menestys tässä tilanteessa voisi olla mahdollinen aihe jatkotutkimukselle.

Kokeissa ei käytetty pakkaustekniikoita HTTP-yhteydessä. Tämä olisi saattanut nopeuttaa JSON-menetelmää. Toisaalta gRPC-tekniikka mahdollistaa batch-muotoisen lähetyksen, jossa useampi viesti pakataan samaan lähetykseen, mikä voi parantaa lähetysnopeutta. Myös näiden tekniikoiden tutkiminen on mahdollinen aihe jatkotutkimukselle.

Lisäksi kiinnostava tutkimusaihe olisi ottaa mukaan vertailuun viestijonoihin perustuvia tekniikoita. Näitä voisivat olla esimerkiksi RabbitMQ sekä AWS:n (Amazon Web Services) tarjoamat SQS (Simple Queue Service) - ja SNS (Simple Notification Service) -palvelut. Suoritetussa kokeessa vastaanotettuja viestejä ei tallennettu vaan vain tulostettiin ruudulle, mikä oli nopeaa. Jos tallentamisessa voisi kestää pidempi aika, voisi lähettämiseen kuluva aika kasvaa. Tällöin voisi lähettävän mikropalvelun kannalta olla hyvä, että lähetys tehtäisiinkin viestijonoon eikä suoraan tallentavaan palveluun. Viestijonossa viesti on tallessa ja se tallennetaan lopulliseen tietorakenteeseen sitten, kun kapasiteettia on. Lähettävän palvelun kannalta viesti on kuitenkin jo saatu perille.

Tarkat mittaustulokset on esitetty taulukossa 5. Sarake ”Run” kertoo ajokerran. Sarakkeessa ”Msgs” on ajokerralla käsiteltyjen viestien määrä. Sarakkeet ”JSON (ms)” sekä ”gRPC (ms)” ilmoittavat eri tekniikoilla käytetyn ajan millisekunteina. Sarakkeet ”JSON average” ja ”gRPC average” ilmoittavat viestiä kohden keskimäärin kuluneen ajan millisekunteina. Taulukon osiossa ”Local” ilmoitetaan yhdellä koneella tapahtuneet mittaukset ja osiossa ”Remote” kahden langattoman lähiverkon kautta yhteydessä olleen koneen mittaukset.

5.3 Yhteenveto kokeesta

Käytännönosuudessa tutkittiin tiedonsiirtoon kuluvaa aikaa, kun lähetetään OPC UA -protokollalla luettua anturidataa mikropalvelusta toiseen. Ensin tutkimuksessa luotiin OpenAPI- ja Protocol Buffers -kielillä määritelmät rajapinnoille, joita käytetään IoT-datan tiedonsiirtoon. Protocol Buffers -määritelmästä generoitiin Go-ohjelmointikielinen pohja gRPC-protokollaa ja ProtoBuf-muotoisia viestejä käyttäville mikropalveluille, joista toinen lähettää

viestejä ja toinen vastaanottaa niitä. OpenAPI-määrityksen perusteella ohjelmoitiin vastaavat REST-rajapinnat sekä JSON-viestimuodot. Tämän jälkeen mikropalveluille laadittiin yksinkertaiset toteutukset.

Kokeissa luettiin KEPServerEX-ohjelman tuottamaa simuloitua anturidataa OPC UA -protokollalla Go-kielen kanavaan. Kun haluttu määrä viestejä oli luettu, viestit lähetettiin vaihtelevassa järjestyksessä sekä gRPC ja Protocol Buffers että REST ja JSON -tekniikoilla. Luentakertoja oli sata ja jokaisella kerralla luettiin keskimäärin 1300 viestiä.

Lähetystä tehtiin kahdella eri tavalla: niin, että lähettävät ja vastaanottavat mikropalvelut sijaitsivat samalla tietokoneella, ja niin, että ne sijaitsivat eri koneilla, jotka oli yhdistetty Jolla-matkapuhelimesta luodulla ad-hoc-verkolla. Kaikki mikropalvelut oli toteutettu Go-kielellä. Verkon yli lähettäessä lähettävät mikropalvelut toimivat Windows-käyttöjärjestelmässä ja vastaanottavat mikropalvelut Linux-käyttöjärjestelmässä.

Tuloksista kävi ilmi, että samalla tietokoneella toimiessaan REST ja JSON -mallia käyttävät mikropalvelut toimivat hieman nopeammin. Kun lähettävät ja vastaanottavat mikropalvelut olivat erillisillä tietokoneilla ja viestit piti välittää verkkoyhteyden yli, oli gRPC ja Protocol Buffers merkittävästi nopeampi tekniikka.

Ennakkoarviona oli, että REST ja JSON olisivat kummassakin tilanteessa hitaampi tekniikka. Tutkimuksessa ei etsitty selitystä siihen, miksi tulos poikkesi tältä osin ennakkoarviosta. Selityksen etsiminen on mahdollinen jatkotutkimusaihe. Tutkimuksessa ei myöskään hyödynnetty REST-tekniikan yhteydessä pakkausta eikä gRPC-tekniikan yhteydessä usean viestin yhtäaikaisen lähettämisen mahdollistavaa batch-ominaisuutta. Näiden mahdollinen nopeusvaikutus on toinen mahdollinen jatkotutkimusaihe. Kokeen mukaiseen tilanteeseen voisi myös sopia viestijonoihin perustuva tekniikka, jolloin lähettäjän kannalta riittää, että viesti on jonossa vaikka sitä ei kenties vielä ole lopullisesti käsitelty. Viestijonojen ottaminen mukaan vertailuun on kolmas mahdollinen aihe jatkotutkimukselle.

measurements

Local						Remote					
Run	Msgs	JSON (ms)	GRPC (ms)	JSON Average	gRPC Average	Run	Msgs	JSON (ms)	GRPC (ms)	JSON Average	gRPC Average
1	1215	589	578	0.4848	0.4757	1	1209	10819	5383	8.9487	4.4524
2	1208	583	554	0.4826	0.4586	2	1203	10604	7741	8.8146	6.4347
3	1218	566	557	0.4647	0.4573	3	1201	11286	526	9.3972	0.4380
4	1217	560	554	0.4601	0.4552	4	1205	10965	5339	9.0996	4.4307
5	1431	676	656	0.4724	0.4584	5	1385	13002	6407	9.3877	4.6260
6	1210	554	557	0.4579	0.4603	6	1213	12975	6657	10.6966	5.4880
7	1215	554	548	0.4560	0.4510	7	1512	14091	8077	9.3194	5.3419
8	1205	323	348	0.2680	0.2888	8	1208	11862	6094	9.8195	5.0447
9	1205	322	342	0.2672	0.2838	9	1424	13063	6684	9.1735	4.6938
10	1212	325	344	0.2682	0.2838	10	1212	10921	5629	9.0107	4.6444
11	1366	377	394	0.2760	0.2884	11	1204	10907	5032	9.0590	4.1794
12	1218	327	352	0.2685	0.2890	12	1212	10924	6467	9.0132	5.3358
13	1211	325	349	0.2684	0.2882	13	1214	11169	6906	9.2002	5.6886
14	1206	325	351	0.2695	0.2910	14	1214	10958	7571	9.0264	6.2364
15	1433	385	425	0.2687	0.2966	15	1300	11867	7808	9.1285	6.0062
16	1376	367	399	0.2667	0.2900	16	1206	11044	5084	9.1575	4.2156
17	1213	325	351	0.2679	0.2894	17	1212	15854	5292	13.0809	4.3663
18	1215	327	355	0.2691	0.2922	18	1202	10845	5157	9.0225	4.2903
19	1216	323	354	0.2656	0.2911	19	1212	10916	5217	9.0066	4.3045
20	1423	383	409	0.2691	0.2874	20	1205	10847	5133	9.0017	4.2598
21	1212	321	352	0.2649	0.2904	21	1213	11005	5343	9.0725	4.4048
22	1215	326	360	0.2683	0.2963	22	1302	117	5557	0.0899	4.2680
23	1440	388	421	0.2694	0.2924	23	1300	11417	5496	8.7823	4.2277
24	1205	332	344	0.2755	0.2855	24	1213	10985	519	9.0561	0.4279
25	1214	326	345	0.2685	0.2842	25	1210	10776	5138	8.9058	4.2463
26	1217	327	348	0.2687	0.2859	26	1359	12789	6293	9.4106	4.6306
27	1206	323	346	0.2678	0.2869	27	1209	10986	5063	9.0868	4.1878
28	1216	328	353	0.2697	0.2903	28	1451	13959	8733	9.6203	6.0186
29	1350	364	387	0.2696	0.2867	29	1213	12101	574	9.9761	0.4732
30	1217	324	350	0.2662	0.2876	30	1279	12401	6075	9.6959	4.7498
31	1201	323	348	0.2689	0.2898	31	1198	11033	6293	9.2095	5.2529
32	1216	330	348	0.2714	0.2862	32	1312	11601	5654	8.8422	4.3095
33	1408	375	405	0.2663	0.2876	33	1181	11209	8789	9.4911	7.4420
34	1214	325	350	0.2677	0.2883	34	1213	11374	5812	9.3768	4.7914
35	1363	368	389	0.2700	0.2854	35	1464	15701	7138	10.7247	4.8757
36	1212	322	354	0.2657	0.2921	36	1415	13077	6969	9.2417	4.9251
37	1204	323	343	0.2683	0.2849	37	1274	13898	5584	10.9089	4.3830
38	1352	363	393	0.2685	0.2907	38	1434	13514	9867	9.4240	6.8808
39	1216	323	351	0.2656	0.2887	39	1212	10953	5459	9.0371	4.5041
40	1308	368	382	0.2813	0.2920	40	1494	13098	6486	8.7671	4.3414
41	1328	359	393	0.2703	0.2959	41	1217	15907	5337	13.0707	4.3854
42	1216	328	368	0.2697	0.3026	42	1416	12746	6188	9.0014	4.3701
43	1204	328	356	0.2724	0.2957	43	1370	16994	6073	12.4044	4.4328
44	1215	326	347	0.2683	0.2856	44	1443	13162	8979	9.1213	6.2225
45	1213	335	351	0.2762	0.2894	45	1271	11674	9143	9.1849	7.1935
46	1216	333	350	0.2738	0.2878	46	1371	12616	9228	9.2020	6.7309
47	1213	330	349	0.2721	0.2877	47	1401	1283	706	0.9158	0.5039
48	1215	327	358	0.2691	0.2947	48	1286	11748	5569	9.1353	4.3305
49	1366	369	395	0.2701	0.2892	49	1206	11384	5882	9.4395	4.8773

Kuvio 5. Mittaustulokset

measurements

50	1217	328	357	0.2695	0.2933	50	1212	11389	5689	9.3969	4.6939
51	1205	327	383	0.2714	0.3178	51	1346	12287	7533	9.1285	5.5966
52	1386	396	498	0.2857	0.3593	52	1214	11152	6004	9.1862	4.9456
53	1344	382	539	0.2842	0.4010	53	1340	12171	5729	9.0828	4.2754
54	1214	347	348	0.2858	0.2867	54	1322	12052	5485	9.1165	4.1490
55	1440	385	456	0.2674	0.3167	55	1213	10875	5144	8.9654	4.2407
56	1334	360	380	0.2699	0.2849	56	1216	10956	5099	9.0099	4.1933
57	1259	338	362	0.2685	0.2875	57	1524	14003	649	9.1883	0.4259
58	1448	403	410	0.2783	0.2831	58	1210	11562	5324	9.5554	4.4000
59	1441	389	412	0.2700	0.2859	59	1213	10968	5033	9.0420	4.1492
60	1432	386	414	0.2696	0.2891	60	1406	15484	6005	11.0128	4.2710
61	1212	326	355	0.2690	0.2929	61	1214	10741	5631	8.8476	4.6384
62	1365	372	396	0.2725	0.2901	62	1213	1086	6148	0.8953	5.0684
63	1320	357	483	0.2705	0.3659	63	1216	10622	5341	8.7352	4.3923
64	1214	332	350	0.2735	0.2883	64	1307	11425	5408	8.7414	4.1377
65	1369	372	408	0.2717	0.2980	65	1216	10599	4967	8.7163	4.0847
66	1366	367	398	0.2687	0.2914	66	1211	10721	6862	8.8530	5.6664
67	1217	335	503	0.2753	0.4133	67	1432	12697	832	8.8666	0.5810
68	1211	335	346	0.2766	0.2857	68	1388	1234	5677	0.8890	4.0901
69	1422	384	411	0.2700	0.2890	69	1290	1189	5245	0.9217	4.0659
70	1432	386	424	0.2696	0.2961	70	1211	10958	5409	9.0487	4.4666
71	1212	327	347	0.2698	0.2863	71	1334	12001	7664	8.9963	5.7451
72	1200	325	343	0.2708	0.2858	72	1397	12705	5746	9.0945	4.1131
73	1372	372	399	0.2711	0.2908	73	1354	12081	5662	8.9225	4.1817
74	1333	359	386	0.2693	0.2896	74	1314	11738	5436	8.9330	4.1370
75	1323	363	379	0.2744	0.2865	75	1368	15286	575	11.1740	0.4203
76	1357	366	387	0.2697	0.2852	76	1364	12225	5662	8.9626	4.1510
77	1334	372	386	0.2789	0.2894	77	1345	12002	5684	8.9234	4.2260
78	1205	327	344	0.2714	0.2855	78	1409	12293	5658	8.7246	4.0156
79	1482	403	438	0.2719	0.2955	79	1321	11329	5431	8.5761	4.1113
80	1412	388	408	0.2748	0.2890	80	1428	12592	6028	8.8179	4.2213
81	1431	392	408	0.2739	0.2851	81	1226	10699	5259	8.7268	4.2896
82	1502	404	430	0.2690	0.2863	82	1216	10815	5089	8.8939	4.1850
83	1436	393	414	0.2737	0.2883	83	1442	1273	6101	0.8828	4.2309
84	1216	328	358	0.2697	0.2944	84	1215	10802	5014	8.8905	4.1267
85	1216	331	367	0.2722	0.3018	85	1217	1096	5021	0.9006	4.1257
86	1349	365	382	0.2706	0.2832	86	1216	10771	5107	8.8577	4.1998
87	1210	338	350	0.2793	0.2893	87	1333	11945	5536	8.9610	4.1530
88	1376	370	393	0.2689	0.2856	88	1421	12829	6021	9.0281	4.2372
89	1431	395	411	0.2760	0.2872	89	1213	11125	5215	9.1715	4.2993
90	1214	336	350	0.2768	0.2883	90	1309	11888	5987	9.0817	4.5737
91	1214	326	355	0.2685	0.2924	91	1335	12162	5624	9.1101	4.2127
92	1322	363	391	0.2746	0.2958	92	1214	11	5173	0.0091	4.2611
93	1211	357	398	0.2948	0.3287	93	1209	10842	5276	8.9677	4.3639
94	1280	345	365	0.2695	0.2852	94	1278	1135	5391	0.8881	4.2183
95	1323	399	432	0.3016	0.3265	95	1211	10913	5084	9.0116	4.1982
96	1356	380	417	0.2802	0.3075	96	1213	10906	4977	8.9909	4.1031
97	1547	427	448	0.2760	0.2896	97	1385	12468	5842	9.0022	4.2181
98	1432	391	416	0.2730	0.2905	98	1333	11897	5549	8.9250	4.1628
99	1632	479	479	0.2935	0.2935	99	1312	1179	5348	0.8986	4.0762
100	1534	418	457	0.2725	0.2979	100	1294	11811	5396	9.1275	4.1700
total	129895	37076	39784	0.2854	0.3063	**total**	129020	1089417	562921	8.4438	4.3631

Kuvio 6. Mittaustulokset, sivu 2

6 Pohdinta

Mikropalvelujen välisen tiedonsiirron tehostamiseen on olemassa erilaisia tapoja. Perinteiseen JSON- ja REST-menetelmään on mahdollista saada nopeutusta muutamin keinoin.

Golovko (2020) selvitti, että JSON-serialisointia ja -deserialisointia voi saada 2–4 kertaa nopeammaksi Go-kielessä valitsemalla tilalle jonkin muun kirjaston standardikirjaton sijaan. Eri kirjastojen hyöty kuitenkin vaihtelee objektien koon mukaan. Joissain tilanteissa JSON-rakenteen lukeminen voi nopeutua jopa monituhatkertaiseksi käyttämällä kirjastoa, joka parsii suoraan rakennetta sen sijaan että muodostaisi siitä Go-kielisen rakenteen. Tällaista ei kuitenkaan voida käyttää serialisointiin.

Toisaalta Aihkisalo ja Paaso (2011) totesi, että WWW-palvelujen tapauksessa ei useinkaan ole suurta merkitystä sillä, kuin nopea on serialisointiprosessi. Isompi merkitys on sillä, kuin suureksi verkkoyhteyden kautta tuleva datamäärä muodostuu. Tällöin nopein lopputulos yleensä saadaan käyttämällä binäärimuotoista protokollaa JSON-muodon sijaan.

Mikäli käytössä on tekstiä oleva viestimuoto kuten XML tai JSON, on REST-rajapinnassa mahdollista käyttää Gzip-pakkausta. (Breje ym. 2018) suosittelevatkin pakkauksen käyttämistä aina, kun se on mahdollista. Näin lähetettävän tiedon määrä voi pudota noin viidennekseen ja siirtonopeus kasvaa 20-40%.

Popić ym. (2016) totesivat, että kaikista nopein tiedonsiirtotapa saattaa olla varta vasten järjestelmälle tehty tiedonsiirtomuoto. Tämän haittapuolena on, että järjestelmää on hankala integroida muiden järjestelmien ja laitteiden kanssa. Tutkimuksessa selvitettiin tiedonsiirtoa suljetun lähdekoodin IoT-järjestelmässä, jossa oman protokollan käyttö on joissain tilanteissa mahdollista.

Myös Sumaray ja Makki (2012) totesivat binäärimuotoiset Thrift- ja ProtoBuf-tekniikat tekstimuotoisia nopeammiksi, kun asiaa tutkittiin mobiililaitteiden näkökulmasta. Tutkimuksessa kuitenkin huomautettiin, että tekstimuotoiset tekniikat, kuten XML ja JSON, voivat tuoda etuja sen vuoksi, että niitä on mahdollista ihmisen lukea ja toisaalta siksi, että niiden tuki on lähes kaikissa ohjelmointikielissä valmiina kun taas esimerkiksi ProtoBuf voi vaatia erillisen

kirjaston käyttöä.

Golovko (2020) totesi ProtoBufin hyväksi valinnaksi Akka.NET-ohjelmakehyksen serialisointimenetelmäksi. Kaikissa tilanteissa ProtoBuf ei kuitenkaan toiminut odotetun nopeasti. Tutkijat pohtivat, että tiedonsiirtonopeutta saattaisi rajoittaa jokin Akka.NET-kehyksen rakenteessa oleva. Wibowo (2011) puolestaan totesi Protocol Buffers -muodon soveliaaksi Twitter-mikroblogipalvelun viestimuodoksi. Suurin hyöty tulisi viestien vaatimassa tilassa, joka putoaisi noin kolmasosaan.

Hong, Sik Yang ja Kim (2018) selvittivät RabbitMQ-viestijonon toimintaa WWW-palvelun kanssa. Se todettiin pienellä kuormalla tehottomaksi, mutta kun kuormitus kasvoi, se nousi REST-versiota nopeammaksi. Erityisen paljon viestijono auttoi, kun kuormitus oli palvelun kapasiteettiin nähden aivan liian suurta, jolloin jono toimi puskurina, joka estää palvelua joutumasta kokonaan toimimattomaan tilaan.

Tutkimuksessa Shafabakhsh (2020) saatiin tietoa REST-, gRPC- ja RabbitMQ-tekniikoista, kun niitä sovellettiin mobiililaitteella käytettävään verkkokauppaan. Järjestelmässä mobiililaite otti yhteyden rajapintayhdyskäytävään REST-rajapinnan kautta käyttäen JSON-viestimuotoa. Yhdyskäytävästä pyyntö välitettiin mikropalveluille, jotka kommunikoivat eri tekniikoiden avulla. Tässä kokeessa ei REST- ja gRPC-tekniikoiden välille tullut suurta eroa. Tämä voisi selittyä osittain sillä, että yhteys mobiililaitteeseen oli yhä REST-tekniikalla toteutettu. Sen sijaan RabbitMQ paljastui nopeimmaksi, kun yhtäaikaisten käyttäjien määrä kasvoi. Koska RabbitMQ viestijonona toimii puskurina palveluiden välillä, kuulostaa tulos odotetulta ja on linjassa tutkimuksen Hong, Sik Yang ja Kim (2018) kanssa. RabbitMQ-versio oli saatavuudeltaan paras, koska siinä vikaantumisia ilmeni selvästi harvimmin. Tämä riitti korvaamaan ympäristön uudelleenkäynnistyksen vaatiman pidemmän ajan. Skaalautuvuudeltaan RabbitMQ arvioitiin myös parhaaksi, koska se sietää kuormitusta ja koska kuormaa voidaan tasata ajamalla välittäjää klusterissa. Toisaalta tutkimus huomioi RabbitMQ:n isomman kompleksisuuden, sillä se vaatii enemmän ohjelmakoodia ja sen testaamisessa mukaan tulee uutena elementtinä välittäjä. Tässä suhteessa REST on paras ratkaisu.

Tämän tutkielman käytännönosuudessa tehdyssä kokeessa REST ja JSON sekä Protocol Buffers ja gRPC osoittautuivat suunnilleen yhtä tehokkaiksi tilanteessa, jossa mikropalvelut

toimivat samassa tietokoneessa. Sen sijaan langattoman lähiverkon yli koneelta toiselle tietoa siirrettäessä Protocol Buffers ja gRPC pudottivat siirtoaikaa liki puoleen. Koetilanteessa koetettiin simuloida tapausta, jossa suuri määrä mittausdataa kerätään ja lähetetään huonojen ja välillä puuttuvien verkkoyhteyksien kautta toiseen laitteeseen. Tässä tilanteessa Protocol Buffers ja gRPC olivat nopeita, joten ne soveltuvat hyvin kyseiseen käyttötarkoitukseen vaihtoehdoksi.

7 Yhteenveto

Tutkimuksessa selvitettiin mikropalvelujen tiedonsiirrossa de facto -standardina toimivan JSON-viestejä käyttävän REST-tekniikan toimintaa sekä esiteltiin sille vaihtoehdon tarjoavaa Protocol Buffers -viestimuotoa sekä gRPC-rajapintatekniikkaa. Tutkimuksessa luotiin myös katsaus mikropalveluarkkitehtuurin toimintaan ja historiaan. Lisäksi selvitettiin ohjelmointirajapintojen ja tiedonsiirtomenetelmien toimintaa etenkin mikropalvelujen näkökulmasta. Näin luotiin pohja eri menetelmien suorituskyvyn vertailemiselle.

Kirjallisuuskatsauksessa tutustuttiin aihepiirin tutkimukseen, joissa käsiteltiin REST-tekniikan mahdollisia parannuskeinoja sekä tutkittiin eri tekniikoiden tehokkuutta serialisoinnin, mikroblogien, esineiden Internetin sekä Akka.NET-ohjelmistokehyksen näkökulmasta. Mukana oli myös tutkimuksia, joissa oli käytössä viestijonotekniikka. Useimmissa tutkimuksissa gRPC ja Protocol Buffers todettiin varteenotettavaksi vaihtoehdoksi nopeuden sekä sen vaatiman tiedonsiirtokapasiteetin ja viestien viemän tallennustilan osalta.

Toisaalta JSON- ja REST-tekniikoita on mahdollista nopeuttaa käyttämällä tilanteeseen parhaiten sopivaa JSON-serialisointikirjastoa sekä pakkaamalla REST-rajapinnan käyttämä data Gzip-menetelmällä. JSON-muoto voi joskus olla hyödyllinen myös siksi, että se sopii myös ihmisen luettavaksi. Samoin liki kaikissa ohjelmointikielissä on tuki JSONille, kun taas Protocol Buffer voi vaatia erillisen kirjaston käyttämistä.

Joihinkin tilanteisiin tehokkain ratkaisu voi olla järjestelmää varten kehitetty erityinen tiedonsiirtomuoto. Tämä kuitenkin tekee integrointia muiden järjestelmien kanssa paljon hankalammaksi. Myös viestijonoihin perustuva viestintäkanava voi olla perusteltu, etenkin jos kuormahuippuja pitää saada tasattua. Haittapuolena se lisää järjestelmän kompleksisuutta.

Tutkimuksessa laadittiin myös mikropalvelujärjestelmä OPC UA -protokollalla luettavan IoT-datan tiedonsiirtoon. Taustalla oli anturidataa huonojen ja välillä puuttuvien verkkoyhteyksien kautta toiselle koneelle lähettävä järjestelmä. Tähän tilanteeseen Protocol Buffers ja gRPC osoittautuivat hyväksi tiedonsiirtotavaksi.

Ennakkoarvion vastaisesti samalla koneella toimivien mikropalvelujen tapauksessa REST ja

JSON olivat hieman nopeampi tekniikka. Tähän selityksen etsiminen on mahdollinen aihe jatkotutkimukselle. Tutkimuksessa ei hyödynnetty REST-tekniikan yhteydessä pakkausta eikä gRPC-tekniikan yhteydessä usean viestin yhtäaikaisen lähettämisen mahdollistavaa batch-ominaisuutta. Myös näiden mahdollinen nopeusvaikutus on mahdollinen jatkotutkimusaihe. Lisäksi IoT-tiedon lähettäminen voisi nopeutua viestijonon avulla, sillä tällöin lähettävälle mikropalvelulle riittää, että viesti on siirretty viestijonoon vaikka lopullinen käsittely olisikin kenties vielä kesken. Viestijonojen mukaan ottaminen vertailuun on kolmas mahdollinen aihe jatkotutkimukselle.

Lähteet

Aihkisalo, Tommi, ja Tuomas Paaso. 2011. “A Performance Comparison of Web Service Object Marshalling and Unmarshalling Solutions”, 122–129. Elokuu. https://doi.org/10.1109/SERVICES.2011.61.

Alami-Kamouri, Sophia, Ghizlane Orhanou ja Said Elhajji. 2016. “Overview of mobile agents and security”, 1–5. IEEE.

Breje, Anca-Raluca, Robert Gyorödi, Cornelia Gyorödi, Doina Zmaranda ja George Pecherle. 2018. “Comparative Study of Data Sending Methods for XML and JSON Models”. *International Journal of Advanced Computer Science and Applications* 9 (12). https://doi.org/10.14569/IJACSA.2018.091229. http://dx.doi.org/10.14569/IJACSA.2018.091229.

Buyya, Rajkumar. 2010. “Cloud computing: The next revolution in information technology”. Teoksessa *2010 First International Conference On Parallel, Distributed and Grid Computing (PDGC 2010),* 2–3. https://doi.org/10.1109/PDGC.2010.5679963.

“Channels - A Tour of Go”. 2020. Viitattu 11. marraskuuta 2020. https://tour.golang.org/concurrency/2.

“Documentation - The Go Programming Language”. 2020. Viitattu 11. marraskuuta 2020. https://golang.org/doc/.

Du, Sang Gyun, Jong Won Lee ja Keecheon Kim. 2018. “Proposal of GRPC as a New Northbound API for Application Layer Communication Efficiency in SDN”. Teoksessa *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication.* IMCOM ’18. Langkawi, Malaysia: Association for Computing Machinery. ISBN: 9781450363853. https://doi.org/10.1145/3164541.3164563. https://doi-org.ezproxy.jyu.fi/10.1145/3164541.3164563.

Erl, Thomas. 2016. *Service-Oriented Architecture: Analysis and Design for Services and Microservices.* 2nd. USA: Prentice Hall Press. ISBN: 0133858588.

“Extensible Markup Language (XML) - W3C”. 2020. Viitattu 11. marraskuuta 2020. https://www.w3.org/XML/.

Feki, Mohamed, Fahim Kawsar, Mathieu Boussard ja Lieven Trappeniers. 2013. “The Internet of Things: The Next Technological Revolution”. *Computer* 46 (helmikuu): 24–25. https://doi.org/10.1109/MC.2013.63.

Fialli, Joseph, ja Sekhar Vajjhala. 2003. “The Java Architecture for XML Binding (JAXB)”, http://xml.coverpages.org/jaxb-V07spec.pdf.

Fielding, Roy. 2000. “Architectural Styles and the Design of Network-based Software Architectures”. Väitöskirja, University of California.

Fowler, Martin, ja James Lewis. 2014. “Microservices: a definition of this new architectural term”. Viitattu 11. marraskuuta 2020. https://www.martinfowler.com/articles/microservices.html.

Garcia, Cristiano, ja Ramon Abilio. 2017. “Systems Integration Using Web Services, REST and SOAP: A Practical Report”. *Revista de Sistemas de Informação da FSMA* 19 (kesäkuu): 23–30.

Garcia, Joshua, Daniel Popescu, George Edwards ja Nenad Medvidovic. 2009. “Identifying Architectural Bad Smells”. Teoksessa *2009 13th European Conference on Software Maintenance and Reengineering,* 255–258. https://doi.org/10.1109/CSMR.2009.59.

Ghofrani, Javad, ja Daniel Lübke. 2018. “Challenges of Microservices Architecture: A Survey on the State of the Practice”, 1–8.

“GNU Gzip”. 2020. Viitattu 11. marraskuuta 2020. https://www.gnu.org/software/gzip/.

Golovko, Gregory. 2020. “Best Practices for Speeding Up JSON Encoding and Decoding in Go”. Viitattu 11. marraskuuta 2020. https://yalantis.com/blog/speed-up-json-encoding-decoding.

Gouigoux, Jean-Philippe, ja Dalila Tamzalit. 2017. “From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture”, 62–65. Huhtikuu. https://doi.org/10.1109/ICSAW.2017.35.

“gRPC Concepts”. 2020. Viitattu 11. marraskuuta 2020. https://grpc.io/docs/guides/concepts/.

Hong, Xian Jun, Hyun Sik Yang ja Young Han Kim. 2018. “Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application”. Teoksessa *2018 International Conference on Information and Communication Technology Convergence (ICTC),* 257–259. https://doi.org/10.1109/ICTC.2018.8539409.

Indrasiri, Kasun, ja Danesh Kuruppu. 2020. *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes.* OŔeilly Media. ISBN: 978-1492058335.

Indrasiri, Kasun, ja Prabath Siriwardena. 2018. *Microservices for the Enterprise.* Apress. ISBN: 978-1484238578.

“Introducing JSON”. 2020. Viitattu 11. marraskuuta 2020. https://www.json.org/json-en.html.

Johnson, Pontus, Robert Lagerström, Mathias Ekstedt ja Magnus Osterlind. 2014. “It management with enterprise architecture”. *ICUIMC 12: Proceedings of the 6th International Conference on Ubiquitous Information Management.*

“KEPServerEX - Product Overview”. 2020. Viitattu 11. marraskuuta 2020. https://www.kepware.com/en-us/products/kepserverex/.

Lamersdorf, Winfried. 2011. “Paradigms of Distributed Software Systems: Services, Processes and Self-organization”, 33–40. IEEE.

Littorin, Hannes, Johnny Reijnst ja Robert Sörman Lundgren. 2015. “Serialisering i Akka.NET: en jämförande studie av Json.NET, Protobuf och TypeSerializer”. Tutkielma, Handelshögskolan vid Örebro Universitet.

Mumbaikar, Snehal, ja Puja Padiya. 2013. “Web Services Based On SOAP and REST Principles”.

Nelson, Bruce. 1981. “Remote procedure call”, https://doi.org/10.1007/1-4020-8086-7_8.

Newman, Sam. 2015. *Building Microservices: Designing Fine-Grained Systems.* OŔeilly Media. ISBN: 978-1-4919-5035-7.

Patana, Kari. 2014. “Avoin viestijonoprotokolla Korpin verkkomaksuissa”. Bachelor’s Thesis, Jyväskylän yliopisto.

Petrie, Charles. 2016. *Web Service Composition.* Springer, Cham. ISBN: 978-3-319-32833-1.

Popić, Srđan, Dražen Pezer, Bojan Mrazovac ja Nikola Teslic. 2016. “Performance evaluation of using Protocol Buffers in the Internet of Things communication”, 261–265. Lokakuu. https://doi.org/10.1109/SST.2016.7765670.

Rinnesalo, Kirsi. 2019. “Mikropalveluarkkitehtuuri - Sovelluskohteena JYSOA-arkkitehtuuri”. Bachelor’s Thesis, Jyväskylän yliopisto.

Roohitavaf, Mohammad, Kun Ren, Gene Zhang ja Sami Ben-romdhane. 2019. “LogPlayer: Fault-tolerant Exactly-once Delivery using gRPC Asynchronous Streaming”. eprint: 1911.11286.

Salah, Tasneem, Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri ja Yousof Al-Hammadi. 2016. “The evolution of distributed systems towards microservices architecture”, 318–325. IEEE.

Sandoval, Jose. 2009. *RESTful Java Web Services: Master core REST concepts and create RESTful web services in Java.* Packt Publishing. ISBN: 978-1-847196-46-0.

Schermann, Gerald, Jürgen Cito ja Philipp Leitner. 2015. “All the services large and micro: Revisiting industrial practice in services computing”, 36–47. Springer.

Shafabakhsh, Benyamin. 2020. “Research on Interprocess Communication in Microservices Architecture”. Master’s Thesis, KTH Royal Institute Of Technology.

Speth, Sandro. 2017. “Entwicklung von Microservices mit zusammensetzbaren API-Bausteinen”. Bachelor’s Thesis, Universität Stuttgart.

Sumaray, Audie, ja S. Makki. 2012. “A comparison of data serialization formats for optimal efficiency on a mobile platform”. *ICUIMC Í2: Proceedings of the 6th International Conference on Ubiquitous Information Management* (helmikuu): 1–6. https://doi.org/10.1145/2184751.2184810.

Taibi, Davide, ja Valentina Lenarduzzi. 2018. “On the Definition of Microservice Bad Smells”. *IEEE Software* vol 35 (toukokuu). https://doi.org/10.1109/MS.2018.2141031.

Taibi, Davide, Valentina Lenarduzzi ja Claus Pahl. 2017. “Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation”. *IEEE Cloud Computing* 4 (lokakuu). https://doi.org/10.1109/MCC.2017.4250931.

“Unified Architecture - OPC Foundation”. 2020. Viitattu 11. marraskuuta 2020. https://opcfoundation.org/about/opc-technologies/opc-ua/.

Wibowo, Canggih. 2011. “Evaluation of Protocol Buffers as Data Serialization Format for Microblogging Communication”. Marraskuu.

Videla, Alvaro, ja Jason J. W. Williams. 2012. *RabbitMQ in Action.* Manning. ISBN: 9781935182979.

“YAML - The Natural Language for Data”. 2020. Viitattu 11. marraskuuta 2020. https://www.yaml.io.

Yarygina, Tetiana, ja Anya Bagge. 2018. “Overcoming Security Challenges in Microservice Architectures”, 11–20. Maaliskuu. https://doi.org/10.1109/SOSE.2018.00011.

Liitteet

A Ohjelmaesimerkit

```
// main.go - OPCReader
package main

import (
    "context"
    "flag"
    "fmt"
    "net/http"
    "time"
    log "github.com/sirupsen/logrus"
    opcmessage "github.com/rkcb1430/opcmessage"
    "github.com/gopcua/opcua"
    "google.golang.org/grpc"
)

var (
    handle     = 42
    totalcount *int
    kepURL     *string
    backendIP  *string
    grpcFirst  *bool
    debugMode  *bool
)

func main() {
    totalcount = flag.Int("totalcount", 100,
          "Number of times to read subscribed values from KEPServer")
    kepURL = flag.String("kepurl", "opc.tcp://localhost:49320",
          "KEPServer URL")
    backendIP = flag.String("backend", "172.28.172.3",
          "IP of the backend services")
    grpcFirst = flag.Bool("grpcfirst", true,
          "Whether to send first to GRPC and then JSON or vice versa")
    debugMode = flag.Bool("debug", false, "Run in debug mode")

    flag.Parse()

    if *debugMode {
        log.Info("Debug logging enabled")
        log.SetLevel(log.DebugLevel)
```

```
    }

    log.Debugf(
          "Total count: %d, KEP URL: %s, Backend IP: %s, gRPC first: %v",
          *totalcount, *kepURL, *backendIP, *grpcFirst)

    ctx, cancel := context.WithCancel(context.Background())

    nodes := readNodes()

    log.Debug("Preparing gRPC connection")

    var opts []grpc.DialOption
    opts = append(opts, grpc.WithInsecure())
    grpcURL := fmt.Sprintf("%s:50052", *backendIP)
    conn, err := grpc.DialContext(context.Background(), grpcURL, opts...)
    if err != nil {
        log.Debugf("Got an error: %v", err)
        return
    }
    defer conn.Close()

    grpcClient := opcmessage.NewOPCStorageClient(conn)

    log.Debug("gRPC connected")
+
    httpClient := &http.Client{Timeout: time.Second * 10}

    log.Debug("HTTP client created")

    log.Debug("Connecting to OPC server")

    c := opcua.NewClient(*kepURL)
    err = c.Connect(ctx)

    if err != nil {
        log.Debugf("Cannot connect to OPC: %v", err)
        return
    }

    defer c.Close()

    log.Debug("OPC connected")

    incoming := make(chan *opcua.PublishNotificationData, *totalcount)
    opcSubscription, err := c.Subscribe(&opcua.SubscriptionParameters{
        Interval: 100 * time.Millisecond,
```

```
    }, incoming)
    if err != nil {
        log.Debugf("Got an error: %v", err)
        return
    }
    defer opcSubscription.Cancel()

    err = subscribeNodes(opcSubscription, nodes)
    if err != nil {
        return
    }

    log.Debug("Added subs")

    go opcSubscription.Run(ctx)
    count, messagesGrpc, messagesJson := readMessages(ctx, incoming)
    cancel()

    log.Debugf("Read %d messages, sleep awhile and start sending", count)
    time.Sleep(5 * time.Second)

    if *grpcFirst {
        log.Debug("gRPC first")
        sendUsingGrpc(grpcClient, messagesGrpc)
    } else {
        log.Debug("JSON first")
        sendUsingJson(httpClient, messagesJson)
    }

    time.Sleep(5 * time.Second)

    if *grpcFirst {
        sendUsingJson(httpClient, messagesJson)
    } else {
        sendUsingGrpc(grpcClient, messagesGrpc)
    }
}

// opcfunctions.go - OPCReader
package main

import (
    "bufio"
    "context"
    "fmt"
    "os"
    log "github.com/sirupsen/logrus"
```

```
    "github.com/gopcua/opcua"
    "github.com/gopcua/opcua/ua"
)

func readNodes() []string {
    log.Debug("Reading node IDs from file")

    nodes := make([]string, 0)
    file, err := os.Open("nodes")
    if err == nil {
        scanner := bufio.NewScanner(file)
        for scanner.Scan() {
            nodes = append(nodes, scanner.Text())
        }
    }

    log.Debugf("Read %d node IDs", len(nodes))
    return nodes
}

func subscribeNodes(opcSub *opcua.Subscription,
  subscriptions []string) error {
    errorCount := 0
    for _, sub := range subscriptions {
        err := monitorNode(opcSub, sub)
        if err != nil {
            errorCount++
            log.Debugf("Got an error: %v", err)
        } else {
            log.Debugf("Subbed %s", sub)
        }
    }
    if errorCount > 0 {
        return fmt.Errorf("%d errors when subscribing", errorCount)
    }
    return nil
}

func monitorNode(sub *opcua.Subscription, node string) error {
    id, err := ua.ParseNodeID(node)
    if err != nil {
        return err
    }

    // arbitrary client handle for the monitoring item
    handle++
    handle := uint32(handle)
```

```
    miCreateRequest := opcua.NewMonitoredItemCreateRequestWithDefaults(
          id, ua.AttributeIDValue, handle)
    res, err := sub.Monitor(ua.TimestampsToReturnBoth, miCreateRequest)
    if err != nil {
        return err
    }
    if res.Results[0].StatusCode != ua.StatusOK {
        return fmt.Errorf("Error subing: %d", res.Results[0].StatusCode)
    }
    return nil
}

func readMessages(ctx context.Context,
  incoming chan *opcua.PublishNotificationData)
  (count int, messages1, messages2 chan *opcua.PublishNotificationData) {
    messages1 = make(chan *opcua.PublishNotificationData, *totalcount)
    messages2 = make(chan *opcua.PublishNotificationData, *totalcount)

    for {
        select {
        case <-ctx.Done():
            log.Debug("We're done")
            return
        case res := <-incoming:
            if res.Error != nil {
                log.Debugf("Error reading: %v", res.Error)
                continue
            }
            messages1 <- res
            messages2 <- res
            count++
            if count%100 == 0 {
                log.Debugf("Times read: %d/%d",
                                  count, *totalcount)
            }
            if count == *totalcount {
                close(messages1)
                close(messages2)
                return
            }
        }
    }
}


//grpcsender.go - OPCReader
package main
```

```
import (
    "context"
    "fmt"
    "time"
    "github.com/golang/protobuf/ptypes/timestamp"
    "github.com/gopcua/opcua"
    "github.com/gopcua/opcua/ua"
    opcmessage "github.com/rkcb1430/opcmessage"
)

func makeProtoMessage(data interface{}) *opcmessage.OPCMessage {
    msg := &opcmessage.OPCMessage{}
    switch val := data.(type) {
    case int16, uint16, int32, int64, uint32, uint64:
        msg.ValueType = opcmessage.OPCMessage_TYPES_INT
        msg.ValueInt = getIntValue(data)
    case float32, float64:
        msg.ValueType = opcmessage.OPCMessage_TYPES_FLOAT
        msg.ValueFloat = getFloatValue(data)
    case string:
        msg.ValueType = opcmessage.OPCMessage_TYPES_STRING
        msg.ValueString = val
    case time.Time:
        msg.ValueType = opcmessage.OPCMessage_TYPES_DATETIME
        timeValue := val
        msg.ValueDatetime = &timestamp.Timestamp{
          Seconds: timeValue.Unix(),
          Nanos: int32(timeValue.Nanosecond())}
    case bool:
        msg.ValueType = opcmessage.OPCMessage_TYPES_BOOLEAN
        msg.ValueBoolean = val
    default:
        msg.ValueType = opcmessage.OPCMessage_UNKNOWN
    }
    return msg
}

func sendUsingGrpc(client opcmessage.OPCStorageClient,
  cpGrpc chan *opcua.PublishNotificationData) {
    count, errors := 0, 0
    start := time.Now()
    fmt.Printf("Start sending to GRPC at %s\n", start.String())
    for res := range cpGrpc {
        switch x := res.Value.(type) {
        case *ua.DataChangeNotification:
            for _, item := range x.MonitoredItems {
```

```
                data := item.Value.Value.Value()
                msg := makeProtoMessage(data)

                resp, err := client.Store(context.Background(), msg)

                if resp == nil || resp.Status ==
                  opcmessage.StoringReply_TRANSFER_ERROR || err != nil {
                    errors++
                }
                count++
            }

        default:
            fmt.Printf("what's this publish result? %T\n", res.Value)
        }
    }
    duration := time.Since(start)
    fmt.Printf("Stopped sending to GRPC.
      It took %v to process %d messages.
      Failed to send %d messages\n", duration, count, errors)
}

//jsonsender.go - OPCReader
package main

import (
    "bytes"
    "context"
    "encoding/json"
    "fmt"
    "io/ioutil"
    "net/http"
    "time"
    "github.com/gopcua/opcua"
    "github.com/gopcua/opcua/ua"
    "github.com/prometheus/common/log"
)

type OPCMessage_ValueType int32

const (
    OPCMessage_UNKNOWN        OPCMessage_ValueType = 0
    OPCMessage_TYPES_BOOLEAN  OPCMessage_ValueType = 1
    OPCMessage_TYPES_INT      OPCMessage_ValueType = 2
    OPCMessage_TYPES_FLOAT    OPCMessage_ValueType = 3
    OPCMessage_TYPES_DATETIME OPCMessage_ValueType = 4
    OPCMessage_TYPES_STRING   OPCMessage_ValueType = 5
```

```
)

type OPCMessage struct {
    Guid          string               `json:"guid"`
    ValueType     OPCMessage_ValueType `json:"valuetype"`
    NodeId        string               `json:"nodeid"`
    ValueBoolean  bool                 `json:"valueboolean"`
    ValueInt      int32                `json:"valueint"`
    ValueFloat    float32              `json:"valuefloat"`
    ValueDatetime *time.Time           `json:"valuedatetime"`
    ValueString   string               `json:"valuestring"`
}

func (m OPCMessage) String() string {
    switch m.ValueType {
    case OPCMessage_TYPES_STRING:
        return m.ValueString
    case OPCMessage_TYPES_INT:
        return fmt.Sprintf("%d", m.ValueInt)
    case OPCMessage_TYPES_FLOAT:
        return fmt.Sprintf("%.2f", m.ValueFloat)
    case OPCMessage_TYPES_DATETIME:
        return m.ValueDatetime.String()
    case OPCMessage_TYPES_BOOLEAN:
        return fmt.Sprintf("%t", m.ValueBoolean)
    case OPCMessage_UNKNOWN:
        return ""
    default:
        return ""
    }
}

type StoringReply_Status int32

const (
    StoringReply_TRANSFER_OK    StoringReply_Status = 0
    StoringReply_TRANSFER_ERROR StoringReply_Status = 1
)

func makeJSONMessage(data interface{}) *OPCMessage {
    msg := &OPCMessage{}
    switch val := data.(type) {
    case int16, uint16, int32, int64, uint32, uint64:
        msg.ValueType = OPCMessage_TYPES_INT
        msg.ValueInt = getIntValue(data)
    case float32, float64:
        msg.ValueType = OPCMessage_TYPES_FLOAT
```

```
        msg.ValueFloat = getFloatValue(data)
    case string:
        msg.ValueType = OPCMessage_TYPES_STRING
        msg.ValueString = val
    case time.Time:
        msg.ValueType = OPCMessage_TYPES_DATETIME
        timeValue := data.(time.Time)
        t := time.Unix(timeValue.Unix(), int64(timeValue.Nanosecond()))
        msg.ValueDatetime = &t
    case bool:
        msg.ValueType = OPCMessage_TYPES_BOOLEAN
        msg.ValueBoolean = val
    default:
        msg.ValueType = OPCMessage_UNKNOWN
    }
    return msg
}

func post(url string, data *bytes.Buffer, client *http.Client) []byte {
    var body []byte
    log.Debugf("Sending POST request to %s", url)
    resp, err := client.Post(url,
      "application/x-www-form-urlencoded", data)
    if err != nil {
        log.Errorf("Error reading response: %v", err)
        return body
    }
    defer resp.Body.Close()
    body, _ = ioutil.ReadAll(resp.Body)
    return body
}

func store(ctx context.Context, client *http.Client, msg *OPCMessage)
  StoringReply_Status {
    data, err := json.Marshal(msg)
    if err != nil {
        return StoringReply_TRANSFER_ERROR
    }
    b := bytes.NewBuffer(data)
    jsonURL := fmt.Sprintf("http://%s:3001", *backendIP)
    resp := post(jsonURL, b, client)
    switch string(resp) {
    case "200 OK":
        return StoringReply_TRANSFER_OK
    default:
        return StoringReply_TRANSFER_ERROR
    }
```

```
}

func sendUsingJson(client *http.Client,
  cpJson chan *opcua.PublishNotificationData) {
    count, errors := 0, 0
    start := time.Now()
    fmt.Printf("Start sending to JSON at %s\n", start.String())
    for res := range cpJson {
        switch x := res.Value.(type) {
        case *ua.DataChangeNotification:
            for _, item := range x.MonitoredItems {
                data := item.Value.Value.Value()
                msg := makeJSONMessage(data)

                resp := store(context.Background(), client, msg)

                if resp == StoringReply_TRANSFER_ERROR {
                    errors++
                }
                count++
            }

        default:
            fmt.Printf("what's this publish result? %T\n", res.Value)
        }
    }
    duration := time.Since(start)
    fmt.Printf("Stopped sending to JSON.
      It took %v to process %d messages.
      Failed to send %d messages\n", duration, count, errors)
}


//jsosender.go - OPCReader

//main.go - Receiver-GRPC
package main

import (
    "context"
    "fmt"
    "net"
    "os"
    "time"
    opcmessage "gitlab.ip.cinia.fi/opcmessage"
    "google.golang.org/grpc"
)
```

```
type Server struct {
}

func print(m *opcmessage.OPCMessage) string {
    switch m.GetValueType() {
    case opcmessage.OPCMessage_TYPES_STRING:
        return m.GetValueString()
    case opcmessage.OPCMessage_TYPES_INT:
        return fmt.Sprintf("%d", m.GetValueInt())
    case opcmessage.OPCMessage_TYPES_FLOAT:
        return fmt.Sprintf("%.2f", m.GetValueFloat())
    case opcmessage.OPCMessage_TYPES_DATETIME:
        t := time.Unix(m.GetValueDatetime().Seconds,
          int64(m.GetValueDatetime().Nanos))
        return t.String()
    case opcmessage.OPCMessage_TYPES_BOOLEAN:
        return fmt.Sprintf("%t", m.GetValueBoolean())
    case opcmessage.OPCMessage_UNKNOWN:
        return ""
    default:
        return ""
    }
}

func (s Server) Store(ctx context.Context, in *opcmessage.OPCMessage)
  (*opcmessage.StoringReply, error) {
    fmt.Printf("At %s, got message: %v\n", time.Now(), print(in))
    return &opcmessage.StoringReply{Guid: "kisas"}, nil
}

func main() {
    lis, err := net.Listen("tcp", ":50052")
    if err != nil {
        fmt.Printf("Got error: %v\n", err)
        os.Exit(1)
    }
    var server Server
    var opts []grpc.ServerOption
    grpcServer := grpc.NewServer(opts...)
    opcmessage.RegisterOPCStorageServer(grpcServer, server)
    _ = grpcServer.Serve(lis)
}

//main.go - Receiver-JSON
package main

import (
```

```
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "time"
)

type OPCMessage_ValueType int32

const (
    OPCMessage_UNKNOWN        OPCMessage_ValueType = 0
    OPCMessage_TYPES_BOOLEAN  OPCMessage_ValueType = 1
    OPCMessage_TYPES_INT      OPCMessage_ValueType = 2
    OPCMessage_TYPES_FLOAT    OPCMessage_ValueType = 3
    OPCMessage_TYPES_DATETIME OPCMessage_ValueType = 4
    OPCMessage_TYPES_STRING   OPCMessage_ValueType = 5
)

// Enum value maps for OPCMessage_ValueType.
var (
    OPCMessage_ValueType_name = map[int32]string{
        0: "UNKNOWN",
        1: "TYPES_BOOLEAN",
        2: "TYPES_INT",
        3: "TYPES_FLOAT",
        4: "TYPES_DATETIME",
        5: "TYPES_STRING",
    }
    OPCMessage_ValueType_value = map[string]int32{
        "UNKNOWN":        0,
        "TYPES_BOOLEAN":  1,
        "TYPES_INT":      2,
        "TYPES_FLOAT":    3,
        "TYPES_DATETIME": 4,
        "TYPES_STRING":   5,
    }
)

type OPCMessage struct {
    Guid          string                `json:"guid"`
    ValueType     OPCMessage_ValueType  `json:"valuetype"`
    NodeId        string                `json:"nodeid"`
    ValueBoolean  bool                  `json:"valueboolean"`
    ValueInt      int32                 `json:"valueint"`
    ValueFloat    float32               `json:"valuefloat"`
    ValueDatetime *time.Time            `json:"valuedatetime"`
```

```
    ValueString   string               `json:"valuestring"`
}

func (m OPCMessage) String() string {
    switch m.ValueType {
    case OPCMessage_TYPES_STRING:
        return m.ValueString
    case OPCMessage_TYPES_INT:
        return fmt.Sprintf("%d", m.ValueInt)
    case OPCMessage_TYPES_FLOAT:
        return fmt.Sprintf("%.2f", m.ValueFloat)
    case OPCMessage_TYPES_DATETIME:
        return m.ValueDatetime.String()
    case OPCMessage_TYPES_BOOLEAN:
        return fmt.Sprintf("%t", m.ValueBoolean)
    case OPCMessage_UNKNOWN:
        return ""
    default:
        return ""
    }
}

func handle(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)
    if err != nil {
        fmt.Fprintf(w, "ERROR")
        return
    }

    msg := OPCMessage{}
    err = json.Unmarshal(body, &msg)
    if err != nil {
        fmt.Fprintf(w, "ERROR")
        return
    }
    fmt.Printf("At %s, got message: %s\n", time.Now(), msg)
    fmt.Fprintf(w, "200 OK")
}

func main() {
    http.HandleFunc("/", handle)
    log.Fatal(http.ListenAndServe(":3001", nil))
}
```