Mikael Ylivaara

gRPC-ohjelmistokehys Android-sovelluksessa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

16.4.2023

Tiivistelmä

Tekijä:	Mikael Ylivaara
Otsikko:	gRPC-ohjelmistokehys Android-sovelluksessa
Sivumäärä:	27 sivua
Aika:	16.4.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile Solutions
Ohjaaja:	Lehtori Toni Spännäri

Insinöörityön tavoitteena oli tutkia gRPC-yhteyden käyttämistä mobiilisovelluksessa ja sen vaikutusta tiedonsiirtonopeuksiin verrattuna perinteiseen REST-arkkitehtuurimalliin. Työssä selvitettiin myös gRPC:n käyttöönoton haasteita suhteessa REST-arkkitehtuurimalliin. Työn alussa tutkittiin gRPC-yhteyden käyttöönottoa ja toimintojen lisäämistä, suunniteltiin tarvittavat toiminnot ja odotettavissa olevat tulokset toimivalta yhteydeltä. Lopuksi suunniteltiin käyttöliittymä, joka vastaa ohjelman tarpeita. Työssä tarkasteltiin myös erilaisia mobiilisovellusten kehitystapoja ja erilaisia tietoliikenneratkaisuja, erityisesti REST- ja gRPC-protokollia.

Insinöörityössä kehitettiin kaksi mobiilisovellusta, joissa molemmissa käytettiin sekä gRPC- että REST-rajapintoja, ja palvelinsovellus, joka palveli molempia sovelluksia. Työn tulokset osoittivat, että gRPC-protokollan käyttö mobiilisovelluksessa paransi tiedonsiirron suorituskykyä verrattuna perinteiseen REST-arkkitehtuuriin. gRPC:n käyttöönotto vaatii enemmän opettelua, mutta mahdollistaa helpon koodin generoinnin ja skaalautuvuuden, mikä helpotti sovelluksen kehittämistä ja ylläpitoa. Työ tarjoaa tietoa kehittäjille, jotka suunnittelevat mobiilisovelluksen toteutusta ja haluavat hyödyntää uusimpia teknologioita ja menetelmiä.

Avainsanat: gRPC, REST, mobiilisovellus, Kotlin, Java, Nodejs

Abstract

Author:	Mikael Ylivaara
Title:	gRPC in android application
Number of Pages:	27
Date:	16.4.2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Mobile Solutions
Supervisor:	Toni Spännäri, Senior Lecturer

The aim of this thesis was to investigate how using a gRPC connection in a mobile application affects data transfer speeds compared to the traditional REST architectural model. Additionally, the study explored the challenges of implementing gRPC compared to the REST architectural model. Initially, the investigative part of the work focused on the adoption of the gRPC connection and the addition of functions. Next, the functions to be included in the work were planned, and the kind of results that could be expected from a functioning connection were examined. Finally, a user interface was designed to meet the program's needs.

The work also addressed the development of different mobile applications and various data communication solutions, especially REST and gRPC protocols. In the study, two mobile applications were developed, both of which used both gRPC and REST APIs, and a server application that served both applications. The results of the study showed that using the gRPC protocol in a mobile application improved data transfer performance compared to the traditional REST architecture. Implementing gRPC requires a steeper learning curve, but allows for easy code generation and scalability, which made application development and maintenance easier. The work provides valuable information for developers who are planning to implement a mobile application and want to utilize newer technologies and methods.

Keywords: gRPC, REST, mobile application, Kotlin, Java, Nodejs

Sisällys

1 Johdanto 1

2 Mobiilisovellusten kehitys 1

2.1 Natiivisovellukset 2
2.2 Progressiiviset web-sovellukset 4
2.3 Hybridisovellukset 5

3 Tietoliikennearkkitehtuurit 6

3.1 REST-suunnittelumalli 8
3.2 gRPC-ohjelmistokehys 9

4 Arkkitehtuurien vertailu sovelluksen avulla 12

4.1 Kehitysympäristö 13
4.2 Kehitetyt mobiilisovellukset 14
4.3 Palvelinsovellukset 15

5 Nopeustestien tulokset 18

5.1 JSON- ja Protobuf-tiedonvaihtomuotojen vertailu 18
5.2 Tiedon hakeminen 19
5.3 Tiedon lähetys 21

6 Johtopäätökset 23

7 Yhteenveto 25

Lähteet 26

Lyhenteet

AVD: Android Virtual Device. Emulaattori, joka simuloi Android-laitteen toimintaa.

API: Application programming interface. Ohjelmointirajapinta, määritelmä, jonka avulla eri ohjelma voivat tehdä pyyntöjä toisilleen ja keskustella toistensa kanssa.

Deserialization: Prosessi, jossa muutetaan tallennetun tai siirretyn datan sarjallistettu muoto takaisin ohjelman sisäiseen tietorakenteeseen.

Fly.io: Pilvipalvelu, joka mahdollistaa sovellusten nopean ja helpon käyttöönoton ja hallinnan.

Gradle: Ohjelmistokehityksen automaatio- ja rakennustyökalu, jota käytetään usein Java- ja Kotlin-projektien rakentamiseen ja hallintaan.

HTTP: Hypertext Transfer Protocol on protokolla, jota käytetään tiedonsiirtoon World Wide Webissä.

ICMP: Internet Control Message Protocol on Internet-protokolla, jota käytetään yleensä virheiden ilmoittamiseen ja vianmääritykseen verkkoliikenteessä.

IP: Internet Protocol on Internetin keskeinen verkkokerrosprotokolla, joka mahdollistaa tietokoneiden ja muiden laitteiden välisen tietoliikenteen.

IPsec: Internet Protocol Security on verkkotietoturvan protokolla, joka tarjoaa tietoliikenteen suojausta Internetissä.

JSON: JavaScript Object Notation on kevyt tiedonsiirtoformaatti, joka on helppo luettava sekä ihmiselle että koneelle.

Kotlin: Ohjelmointikieli, joka kehitettiin JetBrains-ohjelmistoyhtiössä. Se on suunniteltu olemaan yhteensopiva Java-alustan kanssa.

Node.js: Avoimen lähdekoodin JavaScript-pohjainen suoritusympäristö, joka mahdollistaa palvelinpuolen sovellusten kehittämisen.

Protocol Buffers: Googlen kehittämä binäärimuotoinen tiedonsiirtoformaatti, joka tunnetaan myös nimellä Protobuf.

RPC: Remote Procedure Call on kommunikointiprotokolla, joka mahdollistaa prosessien tai ohjelmien välisen kommunikoinnin etäältä.

SDK: Software Development Kit on kehitystyökalupaketti, joka sisältää ohjelmistokehittäjille tarvittavat työkalut ja ohjelmistot erilaisten sovellusten ja ohjelmistojen kehittämiseen.

SSL: Secure Sockets Layer on kryptausprotokolla, jota käytetään yleisesti tietoturvan parantamiseen verkkosivustojen välisessä viestinnässä.

Serialization: Prosessi, jossa tietorakenne tai objekti muunnetaan sellaiseen muotoon, joka mahdollistaa sen tallentamisen tai siirtämisen eri järjestelmien välillä.

TCP: Transmission Control Protocol on tietoverkkoprotokolla, joka hallitsee tietojen lähettämistä ja vastaanottamista tietokoneiden välillä luotettavasti.

UDP: User Datagram Protocol on tietoverkkoprotokolla, joka mahdollistaa nopean tiedonsiirron tietokoneiden välillä, mutta ei takaa tietojen toimitusta tai virheiden korjausta.

XML: Extensible Markup Language on merkintäkieli, jota käytetään tietojen tallentamiseen, siirtämiseen ja jakamiseen tietokonejärjestelmissä.

1 Johdanto

Insinöörityössä tutkitaan gRPC-menetelmän käyttöä mobiilisovellusten kehityksessä. Mobiilisovellusten suosion kasvun myötä on noussut esiin tarve nopeammille ja tehokkaammille tietoliikenneprotokollille, joita voidaan käyttää sovellusten palvelinpuolen ja asiakaspuolen välisessä kommunikaatiossa. Työssä tarkastellaan kolmea mobiilisovellusten kehitysmenetelmää: natiivia, progressiivisia web-sovelluksia (PWA) ja hybridimallia sekä kahta tietoliikennearkkitehtuuria: REST ja gRPC.

Insinöörityön tavoitteena on selvittää, miten gRPC-menetelmä toimii mobiilisovelluksessa ja millaisia etuja sillä on muihin protokolliin verrattuna. Tutkielmassa käytetään empiiristä tutkimusmenetelmää, jossa kehitetään kaksi eri mobiilisovellutusta, joissa molemmissa on käytössä gRPC-menetelmä. Tulosten analysoinnissa tarkastellaan muun muassa tiedon hakemisen ja lähettämisen nopeutta ja suorituskykyä sekä vertaillaan JSON- ja Protobuf-formaatteja.

Työn tulokset voivat auttaa mobiilisovellusten kehittäjiä valitsemaan parhaiten soveltuvan tietoliikennearkkitehtuurin ja kehittämään tehokkaampia sovelluksia.

2 Mobiilisovellusten kehitys

Mobiilisovellukset ovat ohjelmia, jotka on suunniteltu toimimaan mobiililaitteilla, kuten älypuhelimilla ja tableteilla. Mobiilisovelluksia voidaan kehittää eri käyttöjärjestelmille, kuten Androidille ja iOS:lle. Mobiilisovellusten kehittäminen vaatii erityistä osaamista, sillä mobiililaitteet ovat rajallisia resurssien suhteen, kuten muistin ja prosessoritehon suhteen. Tämän vuoksi mobiilisovellusten suunnittelussa ja kehittämisessä on otettava huomioon laitteen rajoitukset. (1.)

Mobiilisovellukset ovat tärkeitä monissa eri käyttötapauksissa, kuten verkkokaupan ja sosiaalisen median sovelluksissa. Ne mahdollistavat käyttäjille helpon pääsyn palveluihin ja ovat usein keskeinen osa yrityksen liiketoimintastrategiaa.

Mobiilisovellusten suosion kasvaessa myös niiden kehitys on muuttunut nopeammaksi ja monipuolisemmaksi.

Mobiilisovellusten kehittäminen edellyttää tiivistä yhteistyötä palvelinpuolen kehittäjien kanssa, sillä sovellus tarvitsee usein palvelimelta tulevaa tietoa ja toiminnallisuuksia. Tämän vuoksi hyvin suunniteltu ja toteutettu kommunikaatio mobiilisovelluksen ja palvelimen välillä on tärkeä osa sovelluksen toimivuutta ja käyttökokemusta.

Mobiilisovellukset jaetaan joskus sen mukaan, ovatko ne selainpohjaisia (PWA) vai käyttöjärjestelmän omaan alustaan erityisesti suunniteltuja (natiivi). Kolmas luokka, hybridisovellukset, yhdistää selainpohjaisten ja alustakohtaisten sovellusten elementtejä. (1.)

2.1 Natiivisovellukset

Natiivikehityksessä sovellus ohjelmoidaan käyttöjärjestelmän (esimerkiksi iOS tai Android) omalla kielellä, mikä mahdollistaa sen hyödyntämään käyttöjärjestelmän ominaisuuksia täysimääräisesti. Tämä tarkoittaa sitä, että sovellus toimii saumattomasti käyttöjärjestelmän kanssa, mikä parantaa sen suorituskykyä ja käyttökokemusta.

Natiivikehitys vaatii kuitenkin erityistä osaamista ja resursseja. Jokaisen käyttöjärjestelmän koodi täytyy kirjoittaa erikseen, mikä tarkoittaa sitä, että kehittäjien täytyy olla perehtyneitä kaikkiin käyttöjärjestelmiin, joille sovellus halutaan julkaista. Lisäksi kehittäjät tarvitsevat erityistä työkaluja ja ohjelmistoja sovelluksen kehittämiseen.

Natiivikehitys voi olla kuitenkin erittäin tehokas tapa kehittää laadukkaita mobiilisovelluksia, ja se tarjoaa monia etuja verrattuna muihin kehitysmenetelmiin. Yksi etu on suorituskyky. Natiivisovellukset käyttävät laitteen suorituskykyä täysimääräisesti, mikä mahdollistaa nopean toiminnan ja sulavan käyttökokemuksen. (2.)

Toinen etu on parempi pääsy laitteen ominaisuuksiin. Natiivisovellukset pääsevät helposti laitteen ominaisuuksiin, kuten kameraan, Bluetoothiin ja NFC:hen, mikä mahdollistaa monipuolisemmat toiminnot. Natiivisovelluksissa on myös helpompi käsitellä monimutkaisia toimintoja, kuten tietokantoihin tallennettujen tietojen hakeminen ja käsittely.

Natiivisovellusten kehityksessä on kuitenkin myös haasteita. Yksi suurimmista haasteista on kehityksen monimutkaisuus. Jokaiselle alustalle on käytettävä erillisiä kehitystyökaluja, kuten Xcodea iOS:lle ja Android Studioa Androidille. Tämä tarkoittaa, että kehittäjän on opittava useita eri ohjelmointikieliä ja -alustoja, jotta hän voi kehittää natiivisovelluksia useille eri alustoille.

Toinen haaste on kehityksen kustannukset. Koska natiivisovellusten kehittäminen vaatii erilaisia työkaluja ja taitoja useille alustoille, kehitys voi olla kalliimpaa kuin hybridisovellusten kehittäminen. Natiivisovellusten kehitys vaatii myös enemmän aikaa, koska jokainen alusta on kehitettävä erikseen.

Lisäksi natiivisovellusten käyttöönotto voi olla haastavaa. Sovelluksen on läpäistävä sovelluskauppojen tarkastukset, ennen kuin se voidaan julkaista. Jos sovellus ei läpäise tarkastuksia, sen julkaiseminen voi kestää kauemmin tai sitä ei ehkä julkaista ollenkaan.

Kuitenkin, kun otetaan huomioon natiivikehityksen hyödyt, on selvää, miksi monet yritykset edelleen käyttävät natiivikehitystä mobiilisovellusten kehittämiseen. Natiivikehitys antaa kehittäjille enemmän kontrollia sovelluksen ulkonäköön ja toiminnallisuuksiin, mikä johtaa parempaan suorituskykyyn ja parempaan käyttökokemukseen käyttäjille. Lisäksi, jos sovellus vaatii erittäin monimutkaisia toiminnallisuuksia, kuten käyttäjän sijaintitietojen tai kameraominaisuuksien käyttöä, natiivikehitys voi olla parempi vaihtoehto. (2.)

Natiivikehityksessä sovelluksen suorituskyky ja käyttäjäkokemus ovat tärkeitä tekijöitä, jotka vaikuttavat sovelluksen menestykseen. Jos sovellus ei toimi sujuvasti tai käyttäjäkokemus on huono, käyttäjät todennäköisesti hylkäävät

sovelluksen nopeasti. Natiivikehitys voi auttaa varmistamaan, että sovellus toimii hyvin kaikilla alustoilla ja tarjoaa paremman käyttökokemuksen käyttäjille. (2.)

2.2 Progressiiviset web-sovellukset

Progressiiviset web-sovellukset (PWA) ovat nousseet suosioon viime vuosina, koska ne tarjoavat käyttäjille monia hyötyjä perinteisiin natiivisovelluksiin verrattuna. PWA:iden kehittäminen on myös edullisempaa ja nopeampaa kuin natiivikehitys.

Progressiivinen web-sovellus kehitetään käyttämällä web-teknologioita, kuten HTML, CSS ja JavaScript. PWA on käyttöliittymältään ja toiminnaltaan samanlainen kuin natiivisovellus, mutta se ei vaadi erillistä lataamista sovelluskaupasta. Sen sijaan se toimii selaimessa ja voidaan lisätä kotinäyttöön kuvakkeena, jolloin se näyttäytyy samanlaisena kuin natiivisovellus.

PWA:iden etuja ovat niiden nopea kehitysaika ja kustannustehokkuus. PWA:n kehittäminen on huomattavasti nopeampaa kuin natiivikehitys, sillä PWA käyttää web-teknologioita, jotka ovat laajalti käytössä. PWA:iden kehittäminen on myös edullisempaa, sillä se ei vaadi erillistä kehitystyötä eri alustoille.

Toinen PWA:iden etu on niiden helppo päivitettävyys. Koska PWA toimii selaimessa, sen päivittäminen on helppoa ja nopeaa ja päivitykset ovat heti saatavilla kaikille käyttäjille.

PWA:iden heikkoutena ovat niiden rajoitukset. PWA:illa ei ole täyttä pääsyä laitteen ominaisuuksiin, kuten esimerkiksi kameraan ja mikrofoniin. Tämä rajoittaa sovelluksen toiminnallisuuksia, ja joissakin tapauksissa se voi olla merkittävä haitta. (3.)

Toinen heikkous on PWA:iden suorituskyvyn rajoitukset. Vaikka PWA:iden suorituskyky on parantunut huomattavasti viime vuosina, se ei vielä yllä täysin

natiivisovelluksen suorituskykyyn. Tämä voi olla ongelma sovelluksissa, jotka vaativat erittäin nopeaa suorituskykyä, kuten esimerkiksi pelit. (3.)

PWA:iden kehittäminen on edullisempaa ja nopeampaa kuin natiivikehitys. PWA:n helppo päivitettävyys on myös etu verrattuna natiivisovelluksiin. Kuitenkin PWA:iden rajoitukset, kuten pääsy laitteen ominaisuuksiin ja suorituskyvyn rajoitukset, voivat olla merkittäviä haittoja joillekin sovelluksille. Ennen PWA:iden valitsemista sovelluksen kehittämiseen onkin tärkeää tarkastella sovelluksen vaatimuksia ja tarpeita ja arvioida, ovatko PWA:n rajoitukset este sovelluksen toiminnallisuuden kannalta.

PWA:iden suosio on kasvanut viime vuosina, ja ne ovat tulleet vakavaksi kilpailijaksi perinteisille natiivisovelluksille. PWA:iden edut, kuten nopea kehitysaika ja kustannustehokkuus sekä helppo päivitettävyys, tekevät niistä houkuttelevan vaihtoehdon sovelluksen kehittämisessä. Kuitenkin PWA:iden rajoitukset on myös tärkeää huomioida ennen valinnan tekemistä. (3.)

2.3 Hybridisovellukset

Hybridisovellukset ovat toinen vaihtoehto mobiilisovelluksen kehittämiseen. Hybridisovellukset yhdistävät natiivisovellusten ja web-sovellusten parhaat puolet, ja ne toimivat sekä selaimessa että laitteen käyttöjärjestelmässä. Hybridisovellus kehitetään käyttämällä web-teknologioita, kuten HTML, CSS ja JavaScript, ja ne voidaan paketoida natiivisovellukseksi käyttämällä alustakohtaisia työkaluja.

Hybridisovellusten etu on niiden kyky toimia usealla eri alustalla, mikä vähentää kehityskustannuksia ja nopeuttaa sovelluksen julkaisua. Lisäksi hybridisovellukset ovat yleensä helppoja päivittää ja ne voivat hyödyntää useita laitteen ominaisuuksia, kuten kameraa ja GPS-paikannusta. (3.)

Hybridisovellusten suorituskyky on parantunut huomattavasti viime vuosina, mutta ne eivät yllä täysin natiivisovellusten suorituskykyyn. Tämä voi olla

merkittävä haitta sovelluksille, jotka vaativat erittäin nopeaa suorituskykyä, kuten pelit. Lisäksi hybridisovelluksilla voi olla yhteensopivuusongelmia eri laitteiden ja käyttöjärjestelmien kanssa, mikä voi johtaa sovelluksen rajoituksiin. (3.)

Toinen hybridisovellusten heikkous on niiden suuri tiedostokoko. Koska hybridisovellus sisältää sekä web-sovelluksen että natiivisovelluksen komponentit, sen tiedostokoko voi olla suurempi kuin pelkän natiivisovelluksen tai pelkän web-sovelluksen.

Hybridisovellukset ovat houkutteleva vaihtoehto mobiilisovelluksen kehittämiseen, sillä ne yhdistävät natiivisovellusten ja web-sovellusten parhaat puolet. Hybridisovellukset toimivat usealla eri alustalla, mikä vähentää kehityskustannuksia ja nopeuttaa sovelluksen julkaisua. Kuitenkin hybridisovellusten suorituskyvyn ja yhteensopivuusongelmien rajoitukset on otettava huomioon sovelluksen kehityksessä. (3.)

3 Tietoliikennearkkitehtuurit

Tietoliikennearkkitehtuuri viittaa tietokoneverkon kokonaisuuden suunnitteluun ja rakenteeseen. Se on tärkeä osa mobiiliviestintää ja laitteiden välistä yhteydenpitoa. Viime vuosina älypuhelinten ja muiden mobiililaitteiden myötä verkkoarkkitehtuurista on tullut entistä tärkeämpi modernin maailman sujuvan toiminnan kannalta.

Tietoliikennearkkitehtuuri kuten myös mobiiliverkkoarkkitehtuuri voidaan jakaa useisiin kerroksiin, joilla on omat erityistehtävänsä. Kerrokset sisältävät fyysisen kerroksen, tiedonsiirtokerroksen, verkkokerroksen, kuljetuskerroksen ja sovelluskerroksen. Jokainen kerros suorittaa tietyn toiminnon ja vastaa siitä, että data siirretään tehokkaasti ja turvallisesti. (4.)

Fyysinen kerros on alin kerros mobiiliverkkoarkkitehtuurissa ja vastaa datan lähettämisestä fyysisessä mediassa. Se sisältää verkon laitteistokomponentit, kuten reitittimet, kytkimet ja kaapelit. Tiedonsiirtokerros vastaa siitä, että data

siirretään luotettavasti ja virheettömästi laitteiden välillä. Tähän kerrokseen kuuluvat sellaiset protokollat kuin Ethernet, Wi-Fi ja Bluetooth. (4.)

Verkkokerros vastaa datan reitittämisestä eri verkkojen välillä. Se käyttää protokollina IP:tä (Internet Protocol) ja ICMP:tä (Internet Control Message Protocol), jotta data toimitetaan oikeaan kohteeseen. Kuljetuskerros vastaa siitä, että data siirretään luotettavasti ja turvallisesti laitteiden välillä. Se sisältää protokollat TCP (Transmission Control Protocol) ja UDP (User Datagram Protocol).

Lopuksi sovelluskerros vastaa käyttäjille tarjottavista palveluista, kuten sähköpostista, verkkoselaamisesta ja sosiaalisesta mediasta. Se käyttää protokollina HTTP:tä (Hypertext Transfer Protocol) ja SMTP:tä (Simple Mail Transfer Protocol) näiden palveluiden tarjoamiseen. (4.)

Yksi keskeinen haaste mobiiliverkkoarkkitehtuurissa on varmistaa, että data siirretään turvallisesti. Mobiililaitteet ovat erityisen alttiita tietoturvaloukkauksille niiden liikuteltavuuden ja helpon katoamisen tai varastamisen vuoksi. Yksi tapa ratkaista tämä haaste on käyttää virtuaalisia erillisverkkoja (VPN) datan salaukseen laitteiden välillä. VPN käyttää protokollina IPsec:ä (Internet Protocol Security) ja SSL:a (Secure Sockets Layer) tarjoamaan turvallisen tunnelin laitteiden välillä.

Toinen haaste mobiiliverkkoarkkitehtuurille on skaalautuvuus ja kapasiteetti. Mobiiliverkoissa on useita käyttäjiä, jotka käyttävät verkkopalveluita ja sovelluksia samanaikaisesti. Tämä aiheuttaa suurta kuormitusta verkkoarkkitehtuurille ja vaatii kykyä skaalautua ja mukautua suurempiin käyttäjämääriin.

Yksi tapa ratkaista skaalautuvuusongelmaa on käyttää pilvipalveluja, jotka voivat tarjota lisää kapasiteettia ja resursseja mobiiliverkolle. Pilvipalvelut voivat myös tarjota parempaa tietoturvaa, koska ne voivat tarjota suojaa eri tasoilla, mukaan lukien verkko- ja sovelluskerros. (5.)

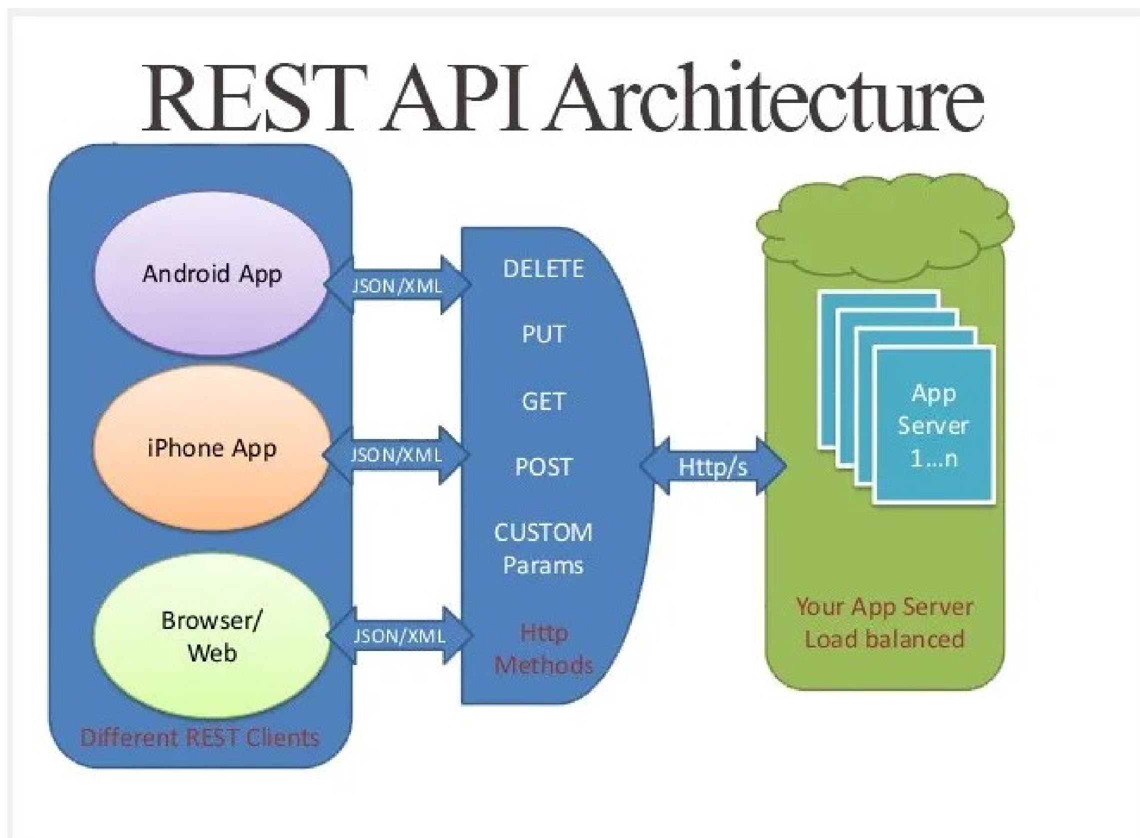
Lisäksi mobiiliverkkoarkkitehtuurissa on otettava huomioon myös liikkuvuus. Mobiililaitteet voivat liikkua eri paikoissa, jolloin ne saattavat vaihtaa verkon

kantavuusaluetta ja siirtyä uuteen tukiasemaan. Tämä vaatii mobiiliverkkoarkkitehtuurilta kykyä tukea langattoman yhteyden siirtoa (handover), joka mahdollistaa laitteen siirtymisen saumattomasti uuteen verkkoympäristöön.

Mobiiliverkkoarkkitehtuuri on tärkeä osa mobiiliviestintää ja laitteiden välistä yhteydenpitoa, mutta tässä työssä keskitytään kahteen yleisimpään: REST-arkkitehtuurimalli ja gRPC-ohjelmistokehys, joka käyttää tietoliikennearkkitehtuurina HTTP/2-protokollaa. (5.)

3.1 REST-suunnittelumalli

REST (Representational State Transfer) on suunnittelumalli, jota käytetään verkkopalveluiden rakentamiseen. Se perustuu HTTP-protokollaan, joka mahdollistaa resurssien käsittelyn erilaisilla HTTP-verbimetodeilla, kuten GET, POST, PUT ja DELETE (kuva 1).



Kuva 1. REST-arkkitehtuurin CRUD-malli (create, read, update, delete) (10).

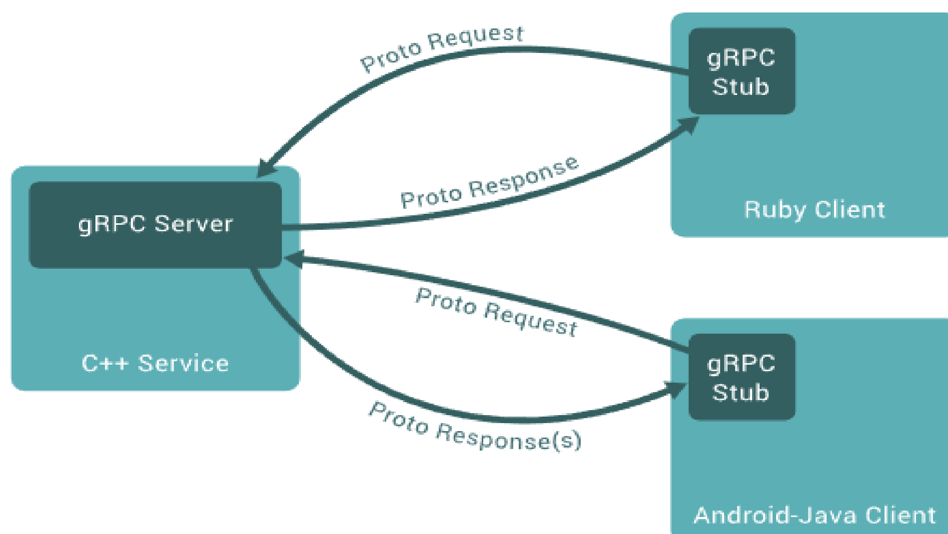
REST-arkkitehtuuri noudattaa joitain periaatteita, jotka auttavat suunnittelemaan skaalautuvia ja helposti ylläpidettäviä verkkopalveluita. Nämä periaatteet ovat

1. resurssien tunnistaminen yksilöllisellä URI:lla
2. HTTP-verbin käyttäminen tietyn toiminnon suorittamiseen resurssilla
3. resurssin tilan siirtäminen esittämällä se JSON- tai XML-muodossa
4. hyperlinkkien käyttäminen resurssien löytämiseen ja yhteyden muodostamiseen.

REST-arkkitehtuuria käytetään laajasti nykyaikaisissa verkkopalveluissa, kuten sosiaalisissa medioissa, verkkokaupoissa ja mobiilisovelluksissa. Se on erityisen hyödyllinen palveluiden välisessä kommunikaatiossa, sillä se mahdollistaa resurssien jakamisen eri järjestelmien välillä helposti. (6.)

3.2 gRPC-ohjelmistokehys

gRPC on Googlen kehittämä avoimen lähdekoodin ohjelmistokehys, joka mahdollistaa tehokkaan etäkutsujen toteuttamisen eri ohjelmointikielillä. gRPC:n avulla voidaan toteuttaa yhteydet eri järjestelmien välille ja siirtää tietoa niiden välillä nopeasti ja tehokkaasti (kuva 2). gRPC perustuu protokolla-puskuriarkkitehtuuriin, joka mahdollistaa tietojen siirtämisen eri ohjelmointikielten välillä.



Kuva 2. gRPC-malli eri ohjelmointikielien välillä (9).

gRPC:hen sisältyy neljä tärkeää komponenttia:

1. Protobuf (Protocol Buffers)

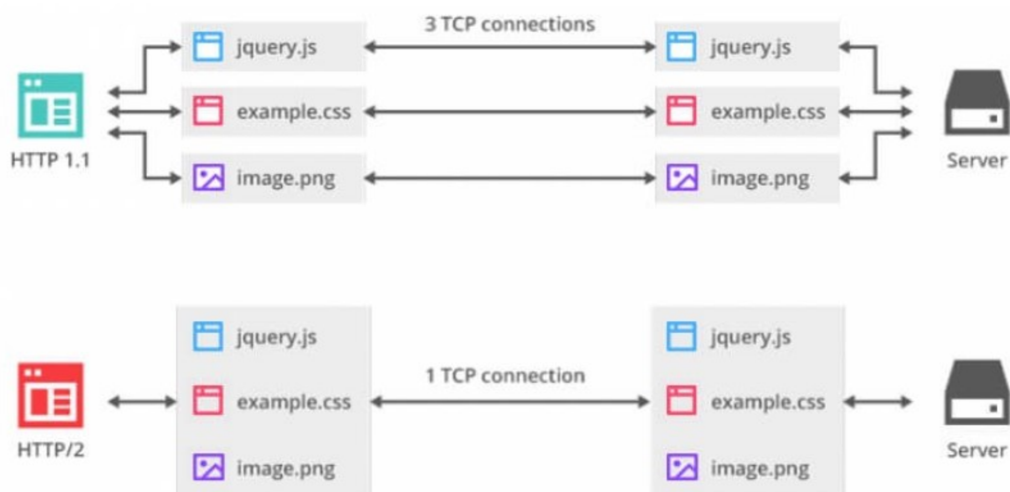
2. RPC (Remote Procedure Call)

3. kanavat (Channels)

4. koodigeneraattorit (Code Generators).

Protobuf on binäärimuotoinen tiedonsiirtoprotokolla, jolla on kyky määrittää tietomalli (.proto) kerran ja sen jälkeen generoida automaattisesti koodia eri ohjelmointikielille, jotta voidaan käyttää samaa tietomallia eri järjestelmissä. Protobufin käyttö mahdollistaa myös nopean tiedonsiirron, sillä Protobuf-muotoisen datan koko on yleensä pienempi kuin esimerkiksi XML- tai JSON-muotoisen datan koko (6).

RPC on etäkutsuprotokolla, jonka avulla voidaan kutsua toisen järjestelmän funktioita etänä. gRPC käyttää HTTP/2-protokollaa (kuva 3), joka mahdollistaa monipuolisen ja tehokkaan tiedonsiirron.



Kuva 3. HTTP1.1:n ja HTTP/2:n ero liittymien määrässä yhden kuvan hakemisessa (11).

Kanavat ovat yhteyksiä kahden järjestelmän välillä. Ne mahdollistavat yhteyksien luomisen ja hallinnoinnin, ja niiden avulla voidaan siirtää tietoa järjestelmien välillä.

Koodigeneraattorit ovat gRPC:n tärkeä komponentti, koska niiden avulla voidaan automaattisesti generoida koodia eri ohjelmointikielille. Koodigeneraattoreiden avulla voidaan säästää aikaa ja minimoida inhimillisten virheiden määrää. (7.)

Tärkeimmät erot gRPC:n ja REST:n välillä näkyvät kuvassa 4.

Feature	gRPC	REST
Protocol	HTTP/2 (Fast)	HTTP / 1.1 (slow)
Payload	Protobuf (binary, surf)	JSON (text, large)
API Contract	Strict, required (.proto)	Loose optional (Open API)
Code Generation	Built-in (protoc)	Third-party tools (swagger)
Security	TLS/SSL	TLS/SSL
Streaming	Bidirectional streaming	Client server request only
Browser Support	Limited (require gRPC-web	Yes

Kuva 4. gRPC:n ominaisuudet verrattuna REST:n vastaaviin ominaisuuksiin (10).

4 Arkkitehtuurien vertailu sovelluksen avulla

Insinöörityössä haluttiin selvittää gRPC-protokollan vaikutusta mobiilisovelluksen suorituskykyyn verrattuna REST-rajapintaan. Lisäksi työssä tarkasteltiin näiden kahden protokollan eroja ohjelmoijan näkökulmasta.

Työ toteutettiin vertailemalla gRPC- ja REST-protokollien suorituskykyä ja helppokäyttöisyyttä mobiilisovelluksen näkökulmasta. Suorituskykyä mitattiin lähettämällä erikokoisia viestejä ja tarkkailemalla niiden lähetysaikaa ja vastausaikaa. Helppokäyttöisyyttä arvioitiin käyttämällä protokollia käytännössä mobiilisovelluksen toteutuksessa ja tarkastelemalla sen vaatimaa koodin määrää ja vaikeusastetta.

4.1 Kehitysympäristö

Asiakaspuolen tutkimuksessa käytettiin Android Studio -kehitysympäristöä, joka on suosittu integroitu kehitysympäristö Android-sovellusten kehittämiseen. Kehitysympäristö sisältää kaikki tarvittavat työkalut, kuten Android SDK:n (Software Development Kit), AVD:n (Android Virtual Device) ja emulaattorin, joiden avulla sovelluksia voidaan kehittää, testata ja simuloida eri laitteissa.

Lisäksi työssä käytettiin gRPC-tekniikkaa, joka on avoimen lähdekoodin projekti, joka tarjoaa tehokkaan tavan rakentaa hajautettuja sovelluksia. gRPC käyttää Protobufia (Protocol Buffers) viestien koodaamiseen ja siirtämiseen verkossa, mikä tekee siitä nopeamman ja tehokkaamman kuin perinteiset tekniikat, kuten XML tai JSON.

Palvelinpuolen toteutuksessa käytettiin Node.js-ympäristöä, joka on suosittu kehitysympäristö JavaScript-ohjelmointikielelle. Node.js mahdollistaa nopean ja skaalautuvan palvelinpuolen toteutuksen, ja sen avulla voidaan rakentaa monipuolisia ja tehokkaita web-sovelluksia.

gRPC-protokollan käyttö palvelinpuolella toteutettiin Node.js:n grpc-pakettia käyttäen. Paketti tarjoaa käyttöön gRPC:n tärkeimmät toiminnallisuudet, kuten palvelimen ja asiakkaan luonnin sekä viestien koodauksen ja dekoodauksen Protobuf-muodossa. Node.js:n vahva tuki asynkroniselle ohjelmoinnille mahdollistaa gRPC-palvelimien nopean ja tehokkaan toteutuksen.

Työssä verrattiin myös REST-rajapinnan toteutusta Node.js:llä. REST-rajapinnan toteutus Node.js:llä on yleinen käytäntö, ja sen toteutus onnistuu esimerkiksi express.js-pakettia käyttäen. REST-rajapinnan toteutus on verrattain yksinkertaista, ja se mahdollistaa resurssien luomisen, päivittämisen, hakemisen ja poistamisen HTTP-protokollan avulla. (9.)

Node.js:n käyttö palvelinpuolen toteutuksessa mahdollistaa nopean ja tehokkaan toteutuksen sekä gRPC- että REST-rajapinnoille. Kummankin teknologian

käyttöön löytyy laajalti dokumentaatiota ja valmiita ratkaisuja, mikä tekee niiden käytöstä suhteellisen helppoa ja nopeaa.

4.2 Kehitetyt mobiilisovellukset

Insinöörityötä varten kehitettiin kaksi yksinkertaista mobiilisovellusta. Ne suunniteltiin Android-käyttöjärjestelmälle, ja toinen toteutettiin Kotlin-ohjelmointikielellä ja toinen JAVA-ohjelmointikielellä. Java valittiin siksi että Kotlinin gRPC:n ohjelmalliset riippuvuudet eivät päivittyneet, koska kehityskoneella Kotlin Gradle ei pysty päivittämään sisäisten ongelmien vuoksi.

Mobiilisovellukset koostuvat yhdestä näkymästä, joka sisältää kolme tekstikenttää, joihin käyttäjä voi syöttää palvelimen IP-osoitteen, porttinumeron ja objektien määrän testauksessa. Lisäksi näkymässä on neljä painiketta: ”Read”, ”Write”, ”Stream” ja ”JSON and Proto object Serialization & Deserialization and size”. Kotlin-versio (kuva 5) sisältää myös kaksi keltaista aluetta, joista ylempi on piirtoalusta ja alempi on palvelimelta tuleva kuva-alue.

Kun käyttäjä on syöttänyt oikean IP-osoitteen, porttinumeron ja objektien määrän, voi käyttäjä halutessaan testata REST- ja gRPC-kirjoitus- ja lukutoimintoja vastaavilla painikkeilla. Tulokset ilmoitetaan listalla, joka sisältää siirretyn datan koon (size) ja datan siirtoon käytetyn ajan. Painike ”Stream” esittää vain gRPC:n virtaavan datan siirron palvelimelta. Painike ”JSON and Proto object Serializtion & Deserialization and size” esittää JSON- ja Protobuf-formaattien sarjallistamista ja palauttamista sarjallistamisesta nopeuden sekä tiedon koon laskemista: kuinka suuri data on, kun se on sarjallistettu JSON- ja Protobuf-formaateissa.

Kuva 5. Emulaattorin esittämä sovelluksen näkymä.

4.3 Palvelinsovellukset

Palvelinpuolen kaksi sovellusta kehitettiin aluksi paikallisella palvelimella testausta varten ja myöhemmin ne nostettiin fly.io-palvelimelle. REST:ä käyttävä palvelin rakennettiin Node.js- ja Express.js-liitännällä (esimerkkikoodi 1) vastaamaan asiakaspuolen kutsuihin ja esittämään viestin käsittelyyn käyttämä aika.

```
{
"dependencies": {
     "async": "^1.5.2",
     "body-parser": "^1.20.2",
     "exectimer": "^2.2.2",
     "express": "^4.18.2"
     }
}
```

Esimerkkikoodi 1. REST-palvelinliitännäiset.

gRPC-rajapinta luotiin käyttäen Node.js:ää ja gRPC-liitännäistä yhdessä Google Protobuf- ja Google Protoloader -kirjastojen kanssa (esimerkkikoodi 2). Google Protobuf on tehokas binääriprotokolla, joka mahdollistaa monimutkaisten

tietomallien määrittelyn ja käytön. Protoloader-kirjasto taas auttaa lataamaan Protobuf-tiedostoja dynaamisesti käytön aikana, mikä mahdollistaa joustavan ja tehokkaan tietojen käsittelyn gRPC-rajapinnan kautta. (19.)

```
{
"dependencies": {
     "@grpc/grpc-js": "^1.1.0",
     "@grpc/proto-loader": "^0.5.0",
     "google-protobuf": "^3.0.0"
     }
}
```

Esimerkkikoodi 2. gRPC-palvelimen mahdollistavat liitännät.

REST-palvelimen siirtäminen fly.io:lle on yksinkertaista: tarvitaan fly.io-tili ja flyctl-komentorivisovellus ja kirjautuminen fly.io-palvelimille. Komento ”flyctl launch” esittää muutaman kysymyksen liittyen sovelluksen esitysnimeen ja julkaistavan alueen sijaintiin, minkä jälkeen flyctl määrittää fly.toml-tiedoston (esimerkkikoodi 3). Tämä tiedosto määrittää julkaisun ja voi siirtää palvelimen komennolla ”flyctl deploy”, joka siirtää palvelimen tutkittavaksi ja testattavaksi. Jos palvelin läpäisee kaiken, se myös käynnistetään.

```
[experimental]
  auto_rollback = true

[[services]]
  http_checks = []
  internal_port = 8080
  processes = ["app"]
  protocol = "tcp"
  script_checks = []
  [services.concurrency]
    hard_limit = 25
    soft_limit = 20
    type = "connections"

  [[services.ports]]
    force_https = true
    handlers = ["http"]
    port = 80

  [[services.ports]]
    handlers = ["tls", "http"]
    port = 443

  [[services.tcp_checks]]
    grace_period = "1s"
    interval = "15s"
    restart_limit = 0
    timeout = "2s"
```

Esimerkkikoodi 3. Komennon ”flyctl launch” muodostama fly.toml-tiedosto.

gRPC-palvelimen siirtäminen fly.io:lle muodostuu samalla tavalla kuin REST-palvelimen, mutta fly.toml-tiedosto pitää muuttaa käyttämään HTTP/2:ta. Esimerkkikoodissa 4 on tehdyt muutokset, jotka ottavat HTTP/2:n käyttöön.

```
[experimental]
  auto_rollback = true

[[services]]
  http_checks = []
  internal_port = 8080
  processes = ["app"]
  protocol = "tcp"
  script_checks = []
  [services.concurrency]
    hard_limit = 25
    soft_limit = 20
    type = "connections"

  [[services.ports]]
    handlers = []
    port = 80

  [[services.ports]]
    handlers = ["tls"]
    port = 443

   [services.ports.tls_options]
    alpn = ["h2"]
```

Esimerkkikoodi 4. Modifioitu fly.toml-tiedosto HTTP/2:n käyttöön.

5 Nopeustestien tulokset

5.1 JSON- ja Protobuf-tiedonvaihtomuotojen vertailu

Insinöörityössä tehtiin nopeusvertailuja tietojen muuttamisesta, lähettämisestä ja vastaanottamisesta. Testejä suoritettiin yhteensä kaksikymmentä kappaletta. Ensimmäisessä testissä suoritetaan ensin olion serialisointi ja sen jälkeen suoritetaan olion deserialisointi ja esitetään aika, joka kului koko testin suorittamiseen. Testin luonteenomainen nopeus johti siihen, että esimerkkikoodin 5 oliota jouduttiin monistamaan useita kertoja.

```
{
     “firstName”: “Olli”,
     “lastName”: “Ohjelmoija”,
     “age”: 30
}
```

Esimerkkikoodi 5. Objekti, joka on JSON-muodossa.

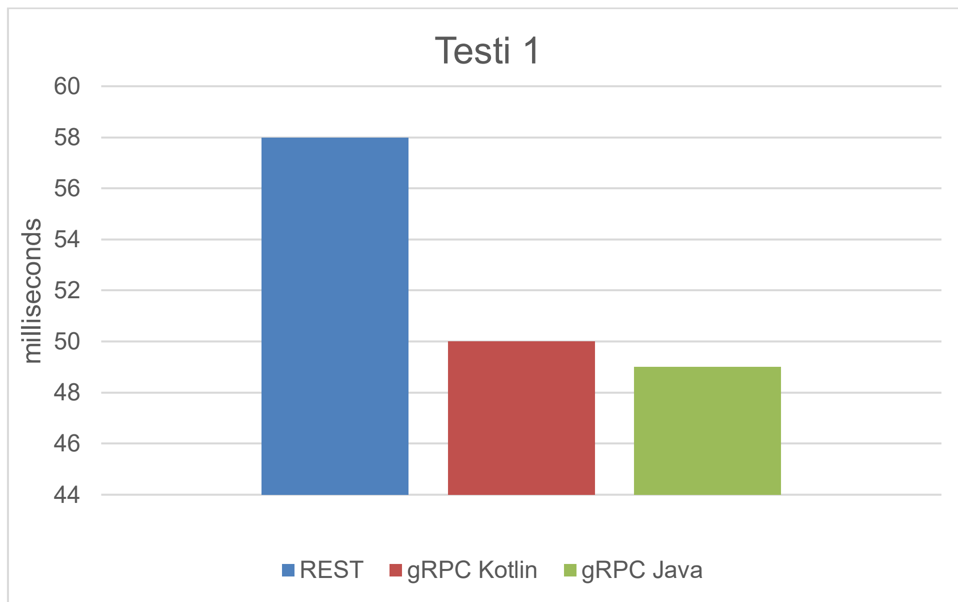
Taulukkoon 1 on koottu JSON- ja Proto-olioiden määrä ja koko lisäksi suoritusaika tiedon muunnossa.

Taulukko 1. JSON-olion koko ja mitatut suoritusajat.

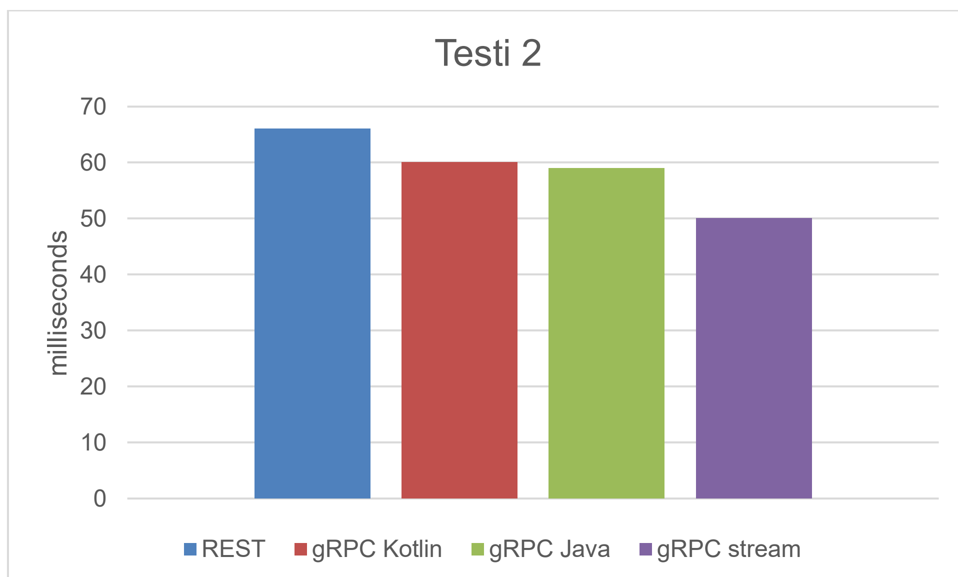
JSON-olioiden lukumäärä	Olion koko	Suoritusaika
1	53 tavua	n. 2 ms
100	5 300 tavua	n. 5 ms
1000	53 000 tavua	n. 28 ms
Proto-olioiden määrä	Olion koko	Suoritusaika
1	20 tavua	n. 0 ms
100	2 000 tavua	n. 0 ms
1000	20 000 tavua	n. 5 ms

5.2 Tiedon hakeminen

Työn toisessa testissä asiakas lähettää vastaanottavien olioiden määrän ja palvelin lähettää esimerkkikoodi 5 mukaisen olion vastaanotetun määrän kerran. Hakuaika alkaa lähetyksen lähettämisestä, jatkuu viestin vastaanottamiseen ja päättyy viestin käsittelyyn, joka sisältää deserialisoinnin tai JSON-parsimisen. Kun olioita otettiin vastaan enemmän, testeihin lisättiin stream palvelu. Tämän lisäksi, kun testiin lisättiin enemmän olioita, käytettiin lisäksi stream-palvelua niiden hakemiseen.



Kuva 6. Hakuajat palvelimelta yhdelle oliolle, jonka koko 20−53 tavua.

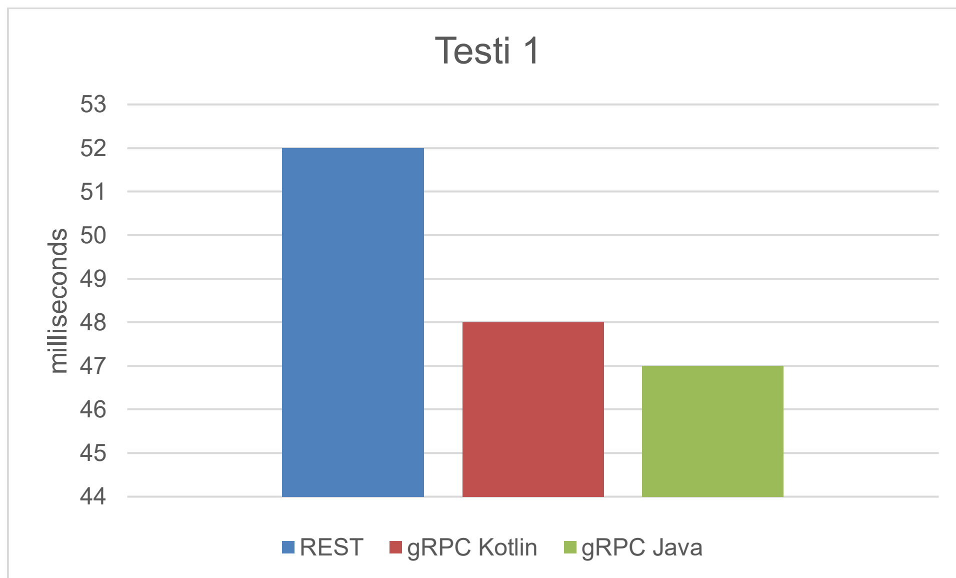


Kuva 7. Hakuajat sadalle oliolle, joiden koko 2000−5200 tavua.

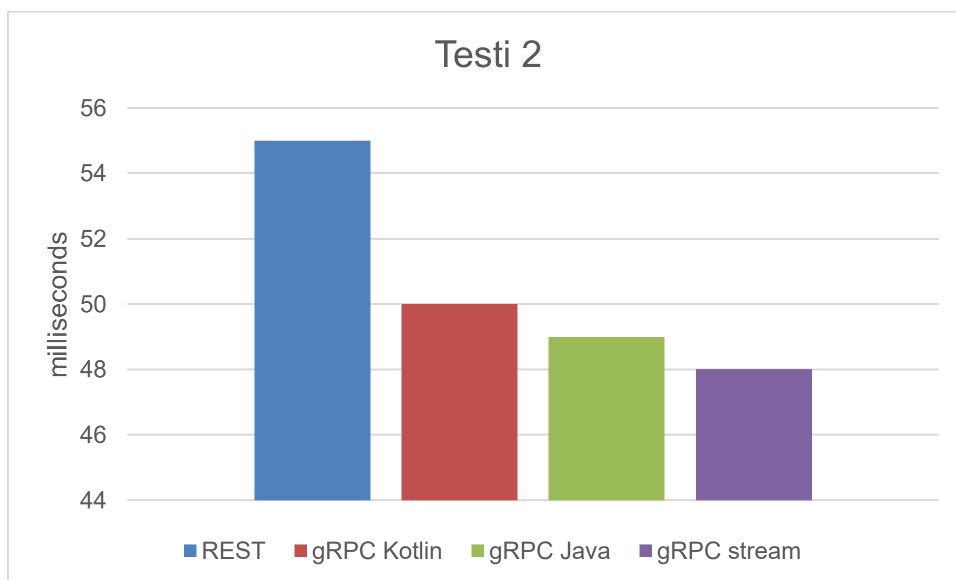
Kuva 8. Hakuajat tuhannelle oliolle, joiden koko 20000−53000 tavua.

5.3 Tiedon lähetys

Työn kolmannessa testissä asiakas lähettää esimerkkikoodin 5 olion palvelimelle testissä 1, ja palvelin kuittaa viestin saaduksi. Myöhemmin olio monistetaan moninkertaiseksi. Lähetysnopeus muodostuu testin lähetyksen ja vastauksen saamisen välisestä ajasta. Tämän lisäksi, kun testiin lisättiin enemmän olioita, käytettiin stream-palvelua niiden lähettämiseen.



Kuva 9. Yhden olion lähetysnopeus, jonka koko 20−53 tavua.



Kuva 10. Sadan olion lähetysnopeus, jonka koko 2000−5300 tavua.

Kuva 5. Tuhannen olion lähetyksen nopeus, jonka koko 20000−53000 tavua.

6 Johtopäätökset

Kaikissa testeissä havaittiin, että gRPC oli hieman nopeampi kuin REST. Javan gRPC-tulokset olivat parempia, koska käytettyjen riippuvuuksien versiot olivat uudemmat. gRPC:n oppimisessa on enemmän työtä ohjelmoijalle, ja sen käyttöönotto vie enemmän aikaa. Toisaalta, kun gRPC on otettu käyttöön, uusien kutsujen luominen on huomattavasti helpompaa. Virhetilanteissa REST on huomattavasti helpompi kuin gRPC, koska REST antaa enemmän tietoa siitä, mikä meni vikaan ja itse viestin lukeminen on helpompaa.

Lisäksi on huomattava, että gRPC:n käyttöönotto edellyttää usein ohjelman arkkitehtuurin uudelleenarviointia, koska gRPC:n käyttöön liittyy uusia käsitteitä, kuten Protobuf-skeemat ja sisäänrakennetut tiedostojen siirrot. Tämä voi vaikuttaa ohjelman suunnitteluun ja kehitysaikaan.

Toisaalta, gRPC:llä on myös etuja REST:iin verrattuna. Esimerkiksi gRPC käyttää HTTP/2 -protokollaa, joka mahdollistaa usean pyynnön lähettämisen samaan yhteyteen ja tarjoaa paremman suorituskyvyn kuin REST:n yksittäiset pyynnöt. gRPC:n tuki monikielisyydelle ja automaattiselle koodigeneroinnille

tekee myös monimutkaisen järjestelmän rakentamisesta helpompaa ja nopeampaa.

Toisaalta gRPC:n käyttäminen mobiilisovelluksessa, joka ottaa yhteyden palvelimeen vain harvoin, ei ole suositeltavaa. Tämä johtuu siitä, että gRPC-kanavan avaaminen ja sulkeminen on verrattain raskasta toimintaa, mikä voi hidastaa yhteyttä, kun käyttäjä haluaa lähettää tai vastaanottaa dataa. REST on tässä tapauksessa parempi vaihtoehto, koska se on kevyempi ja helpompi käsitellä.

gRPC on erinomainen vaihtoehto monimutkaisiin projekteihin, joissa on suuri määrä liikennettä, mutta tärkeintä on valita tekniikka, joka sopii parhaiten projektin tarpeisiin ja vaatimuksiin.

Yksi mahdollinen jatkotutkimuksen aihe liittyisi gRPC:n ja REST:n vertailuun suurten järjestelmien skaalautuvuudessa ja suorituskyvyssä. Tämä voisi sisältää testejä, joissa mitataan molempien tekniikoiden suorituskykyä suurilla datamäärillä ja eri kuormitusprofiileilla.

Lisäksi olisi mielenkiintoista tutkia, miten gRPC ja REST eroavat monimutkaisten järjestelmien kehitys- ja ylläpitokustannuksissa. Tämä voisi sisältää kyselyn ohjelmistokehittäjille siitä, miten he kokevat näiden kahden tekniikan käytön koodin kehityksessä ja ylläpidossa.

Toinen mielenkiintoinen tutkimusaihe olisi gRPC:n ja RESTin käytön vaikutus turvallisuuteen ja tietoturvaan. Tämä voisi sisältää testejä, joissa mitataan molempien tekniikoiden suojautumista tietoturvaan liittyviä hyökkäyksiä vastaan, kuten tietojen vuotoja ja SQL-injektioita.

Voisi olla hyödyllistä tutkia myös, miten gRPC:n ja REST:n käyttö vaikuttaa sovelluksen yhteensopivuuteen eri käyttöjärjestelmien ja sovelluskehysten kanssa. Tämä voisi sisältää testejä, joissa mitataan molempien tekniikoiden suorituskykyä eri käyttöjärjestelmissä ja sovelluskehyksissä.

7 Yhteenveto

Insinöörityössä tutkittiin gRPC:n käyttöä mobiilisovelluksissa ja verrattiin sitä perinteiseen REST-arkkitehtuuriin. Työn tavoitteena oli selvittää, miten gRPC suoriutuu mobiilisovelluksissa tietoliikenteen ja datan käsittelyn osalta ja verrata sen suorituskykyä REST:iin.

Työssä kehitettiin kaksi mobiilisovellusta käyttäen eri ohjelmointikieliä: Kotlin ja Java. Sovellukselle kehitettiin sekä gRPC- että REST-rajapinnat. Lisäksi kehitettiin palvelinsovellus, joka palveli sovelluksia.

Tulokset osoittivat, että gRPC-protokollalla on useita etuja verrattuna perinteisiin REST-rajapintoihin. gRPC-rajapinta tarjoaa nopeamman tiedonsiirron, pienemmän datan siirtoon tarvittavan kaistanleveyden ja paremman käyttökokemuksen huonolla verkkoyhteydellä. Lisäksi gRPC-rajapinnan käyttö mahdollistaa helpon koodin generoinnin ja skaalautuvuuden.

Insinöörityön tulokset osoittavat gRPC:n olevan lupaava ja tehokas menetelmä mobiilisovelluksissa, ja se voi parantaa merkittävästi sovellusten suorituskykyä ja tehokkuutta etenkin tilanteissa, joissa sovellus lähettää paljon tietoa palvelimelle. Tämä on erityisen tärkeää nykypäivän käyttäjille, jotka vaativat nopeita ja sulavia käyttökokemuksia mobiilisovelluksissa. Työstä on hyötyä kehittäjille, jotka suunnittelevat mobiilisovellusten toteutusta ja haluavat hyödyntää uusimpia teknologioita ja menetelmiä.

Työssä tarkasteltiin myös erilaisia mobiilisovellusten kehitysmenetelmiä, kuten natiivit, PWA:t ja hybridit, sekä tietoliikennearkkitehtuurit REST ja gRPC. Tämä tieto auttaa kehittäjiä valitsemaan sopivimman ratkaisun oman sovelluksensa tarpeisiin.

Lähteet

1 Terrel Hanna, Katie & Wigmore, Ivy. 2023. Mobile app. Verkkoaineisto. TechTarget. < https://www.techtarget.com/whatis/definition/mobile-app>. Luettu 13.3.2023.

2 Shaun Lewis & Mike Dunn. 2019. Native Mobile Development. E-book. O’Reilly Media

3 Chris Griffith. 2017. Mobile App Development with Ionic. Revised Edition. E-kirja. O'Reilly Media.

4 Brian Fling. 2009. Mobile Design and Development. E-kirja. O’Reilly Media.

5 Jerry D. Gibson. 2017. Mobile Communications Handbook. 3rd Edition. E-kirja. CRC Press.

6 Lokesh Gupta. 2022. What is REST. Verkkoaineisto. REST API Tutorial. <https://restfulapi.net/>. Luettu 8.1.2023.

7 Maarek, Stephane. 2019. gRPC [Java] Master Class: Build Modern API and Microservices. Verkkoaineisto. Packt Publishing. <https://learning.oreilly.com/videos/grpc-java-master/9781838558048/>. Katsottu 12.3.2023.

8 Protocol Buffers. Verkkoaineisto. Google Developers. <https://developers.google.com/protocol-buffers>. Luettu 6.1.2023.

9 Kasun Indrasiri & Danesh Kuruppu. 2020. gRPC: Up and Running. E-kirja. O’Reilly Media.

10 Beshkov, Mukhaddin. 2021. What is gRPC Protocol and How it Defines API Architecture. Verkkoaineisto. Wallarm. <https://www.wallarm.com/what/the-concept-of-grpc>. Luettu 8.3.2023.

11 Singh, Vinamra. 2022. REST vs gRPC- Which is the best API Protocol? Verkkoaineisto. Quokka Labs. <https://quokkalabs.com/blog/rest-vs-grpc-which-is-the-best-api-protocol>. Luettu 1.4.2023.

12 gRPC. 2023. Verkkoaineisto. gRPC Authors.<https://grpc.io/>. Luettu 5.1.2023.

13 Muthukumarana, Dilanka. 2019. RESTful API Design Best Practices (Principles). Verkkoaineisto. <https://dilankam.medium.com/restful-api-design-best-practices-principles-ded471f573f3>. Luettu 2.4.2023.

14 The Advantages of HTTP2- Why You Shud Move on to HTTP2. 2023. Verkkoaineisto. Cheap SSL Security. <https://cheapsslsecurity.com/p/the-advantages-of-http2/>. Luettu 7.1.2023.

15 Rodriguez, Alex, 2015. Introduction to RESTful Web services. Verkkoaineisto. IBM Developer. <https://developer.ibm.com/articles/ws-restful/>. Luettu 8.1.2023.

16 Gyori, Laszlo. 2022. gRPC vs REST: Getting Started With the Best API Protocol. Verkkoaineisto. Toptal <https://www.toptal.com/grpc/grpc-vs-rest-api>. Luettu 10.1.2023.

17 Hillpot, Jeremy. 2022. gRPC vs REST: How Does gRPC Compare with Traditionall REST APIs? Verkkoaineisto. DreamFactory. https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/. Luettu 11.1.2023.

18 Spurio, Andrea & Bugaj, Hubert. 2021 Comparing gRPC performance across different technologies. Verkkoaineisto. Nexthink. <https://www.nexthink.com/blog/comparing-grpc-performance>. Luettu 12.1.2023.

19 Protocol Buffers Documentation. Verkkoaineisto. Google. <https://protobuf.dev/>. Luettu 13.1.2023.