TAMPEREEN TEKNILLINEN YLIOPISTO

ESA VÄNTSI

TIEDON SYNKRONOINTI PÄÄTELAITTEIDEN VÄLILLÄ

Diplomityö

Tarkastaja: Professori Kari Systä
Tieto- ja sähkötekniikan
koulutusvaradekaani hyväksyi
tarkastajan ja aiheen
4.1.2017

TIIVISTELMÄ

VÄNTSI, ESA: Tiedon synkronointi päätelaitteiden välillä
Tampereen teknillinen yliopisto
Diplomityö, 51 sivua
Joulukuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastaja: Professori Kari Systä
Avainsanat: synkronointi, mobiilisovellus

Koska kaikkialla ei välttämättä ole saatavilla toimivaa matkapuhelinverkkoa, on käyttäjäkokemuksen kannalta tärkeää että älypuhelinsovellusta voidaan käyttää verkottomassa tilassa. Jotta älypuhelinsovelluksen käyttö verkottomassa tilassa olisi mahdollista, voidaan joutua lataamaan koko käyttäjän sisältö älypuhelimelle verkon ollessa saatavilla. Usein on riittävää, että käyttäjän tekemät muutokset tietosisältöön välittyvät taustajärjestelmälle viiveellä.

Tämä työ on tehty osana ohjelmistokehitysprojektia. Projektin vaatimuksiin kuului, että sovellusta on voitava käyttää ilman verkkoyhteyttä mahdollisimman samalla tavalla kuin verkkoyhteyden ollessa käytettävissä. Lisäksi edellytettiin, että sovelluksen käyttäjän on voitava käyttää sovellusta useammalta älypuhelimella samanaikaisesti siten, että tehdyt muutokset välittyvät kaikille laitteille.

Tässä diplomityössä esitellään menetelmä, jolla toteutettiin tietosisällön synkronointi sovelluksessa. Synkronoinnilla voidaan ratkaista edellä mainitut vaatimukset: verkoton käyttö ja muutosten välittyminen kaikille käyttäjän laitteille.

Kehitetyssä menetelmässä synkronoitavilla objekteilla on numeerinen ominaisuus, joka ilmaisee viimeisen muutoksen ajankohtaa. Tämän ominaisuuden avulla objekteja voidaan järjestää muokkausajan mukaan. Numeerinen ominaisuus mahdollistaa sen, että voidaan tarkastella mitä muutoksia on tapahtunut tietyn ajankohdan jälkeen.

Kehitetyn menetelmän avulla projektissa onnistuttiin ratkaisemaan sille asetetut vaatimukset. Mahdollisena ongelmana toteutustavassa nähdään se, että kerralla siirrettävän tiedon määrä voi kasvaa liian suureksi.

ABSTRACT

VÄNTSI, ESA: Data Synchronization Between Client Devices
Tampere University of Technology
Master of Science Thesis, 51 pages
December 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Professor Kari Systä
Keywords: synchronization, mobile application

Because a working mobile network connection is not available everywhere, it is important for the user experience that a mobile application is usable without a network connection. In order for a mobile application to work offline, it may be required to download all of the user's content while the network connection is available.

This thesis is a part of a software development project. It was required that the user experience without a network connection should be as much alike as possible than it is with a network connection. Furthermore, a user must be able to use the software with multiple devices at the same time so that changes would be propagated between the devices. This thesis introduces an algorithm which was used in the mobile application to implement synchronization. Synchronization made it possible to solve the requirements: offline use and propagation of changes between devices.

In the designed algorithm every synchronizable object has a numeric property which indicates when the object has been updated last time. This property makes it possible to sort the objects by the modification time and therefore it's possible to determinate which modifications have happened after a specific time.

The requirements were successfully fulfilled by using the invented algorithm. However, it is known that the data transfer might be problematic if too much information is tried to be transferred at a time.

ALKUSANAT

Tämä diplomityö on syntynyt osana Vincit Development Oy:n asiakasprojektityötä. Kiitokseni sekä asiakasyritykselle että Vincitille, jotka mahdollistivat tämän työn kirjoituksen.

Lisäksi haluan erityisesti kiittää diplomityön tarkastajaa professori Kari Systää hänen antamastaan arvokkaasta rakentavasta palautteesta sekä työpaikkani puolesta työn ohjaajana toiminutta DI Juha Simolaa, joka ohjasi ja oikoluki työtäni läpi koko sen tekoprosessin. Kiitos myös vanhemmilleni Heli ja Risto Väntsille sekä avopuolisolleni Vilma Salorannalle heidän avustaan työn viimeistelyssä.

Helsingissä 18. marraskuuta 2017

Esa Väntsi

SISÄLLYS

TERMIT JA NIIDEN MÄÄRITELMÄT

Klusteri	Järjestelmä, jossa tehtäviä jaetaan useiden laitteiden kesken.
Push-viesti	Palvelimen lähettämä viesti mobiilisovellukselle.
Shard	Yksi tietokantaklusterin osa tietokantaratkaisussa, jossa tietoa on ositettu useamman tietokantapalvelimen kesken.
Tebitavu	2^{40} tavua.
UUID4	128-bittisen yksilöivän tunnisteen neljäs versio. Esitysmuoto yleisesti muotoa *ee1e58c6-6db1-4121-a3b0-702d4f42a94e*.

1. JOHDANTO

Mobiilisovelluksen käyttökokemuksen ei pitäisi heiketä oleellisesti tilanteissa, joissa verkkoyhteyttä ei ole saatavilla tai sen toimintavarmuus on heikko. Tällaisia tilanteita voi esiintyä erityisesti mobiiliverkkojen katvealueilla tai suurkaupungeissa.

Sovelluksesta riippuen voi olla, että tiedon välittömällä siirtymisellä palvelimelle ei ole merkitystä käytön kannalta. Erityisesti sovelluksissa, joissa tietosisällön muutoksia tehdään vain käyttäjää itseään varten, tiedon ei tarvitse päätyä palvelimelle ennen kuin päivittynyttä tietoa tarvitsee käsitellä tai lukea toiselta päätelaitteelta. Tiedon synkronointi mobiilisovelluksessa mahdollistaa sovelluksen käyttämisen ilman verkkoyhteyttä.

Synkronoinnin avulla voidaan viivästyttää sekä muilla laitteilla tehtyjen muutosten lataus laitteelle, että laitteella tehtyjen muutosten lähetys palvelimelle. Käyttäjän tekemiä muutoksia voidaan puskuroida laitteelle hetkinä, jolloin verkkoyhteyttä ei ole saatavilla. Kun verkkoyhteys on saatavilla, voidaan lähettää tehdyt muutokset palvelimelle. Palvelin voi tässä yhteydessä myös lähettää päätelaitteelle edellisen synkronoinnin jälkeen tapahtuneet muutokset.

Palvelimen tehtävänä synkronoinnin aikana on ratkaista mahdollisesti muodostuneet ristiriidat tietosisällössä. Mikäli esimerkiksi kaksi päätelaitetta ovat muokanneet samaa tietuetta, täytyy selvittää näiden muutosten vaikutus tietueeseen. Käyttäjäkokemuksen kannalta on oleellista, että ristiriita ratkaistaan mahdollisimman selkeästi.

Synkronointi mahdollistaa myös useamman päätelaitteen käyttämisen tilanteissa, joissa laitteet toimivat samassa kontekstissa. Samassa kontekstissa toimiminen voi tapahtua esimerkiksi siten, että palveluun on kirjauduttu samalla käyttäjätunnuksella usealta laitteelta. Synkronoinnin tarkoituksena on, että mikäli useampi laite on yhteydessä verkkoon, on niiden tietosisältö samassa tilassa ennen pitkää.

Toimintavarman synkronointimenetelmän toiminta edellyttää hajautettujen järjestelmien haasteiden huomioon ottamista. Monet hajautettujen järjestelmien haasteista johtuvat siitä, että järjestelmässä olevat kellot eivät käy samaa aikaa. Usein on kuitenkin syytä saada tapahtumille järjestys. Tämän työn esittelemä synkronointi-

menetelmä käyttää kellonaikojen sijasta versionumerointia. Tietueiden versionumeroiden hallinnointi suoritetaan keskitetysti palvelimella, jolloin päätelaitteiden väliset erot eivät vaikuta versionumeroiden muodostumiseen ja näin ollen tapahtumien järjestykseen.

Tässä työssä kuvataan Vincit Development Oy:n asiakasyritykselle toteutettu menetelmä tiedon synkronointiin. Menetelmä tulee osaksi asiakasyritykselle toimitettavaa matkapuhelinsovellusta ja sen taustajärjestelmää. Toteutuksen tavoitteena projektin kannalta on mahdollistaa sovelluksen käyttö olosuhteissa, joissa verkkoyhteyttä ei ole saatavilla tai sen toiminta on epävarmaa sekä mahdollistaa käyttäjälle useamman päätelaitteen yhtäaikainen käyttö. Projektiin on toteutettu asiakassovellus iOS-käyttöjärjestelmälle sekä taustajärjestelmä. Työssä kuvataan sovelluksen toimintaympäristö, keskeisimmät teknologiat ja toteutustapa. Toteutettu järjestelmä ei käytä valmiita synkronointiratkaisuja, vaan kuvattu menetelmä rakennettiin sovellusta varten.

Työn tavoitteena on kuvata toteutettu synkronointimenetelmä ja selvittää sen soveltuvuus käyttöympäristöönsä. Synkronointimenetelmän lisäksi esitellään ympäristö, jossa menetelmää on tähän työhön liittyvässä projektissa käytetty. Toteutetun järjestelmän lisäksi työssä arvioidaan saatavilla olevien menetelmien ja järjestelmien toimivuutta projektin käyttötarkoitukseen.

Luvussa 2 käsitellään tiedon monistamiseen sekä hajautettuihin järjestelmiin liittyvää teoriaa. Luvussa 3 esitellään joitain olemassa olevia synkronointimenetelmiä ja arvioidaan niiden soveltuvuutta tässä työssä esiteltyyn käyttötapaukseen. Luvussa 4 esitetään toteutettu menetelmä korkealla tasolla. Luvussa 5 käydään läpi toteutusyksityiskohdat liittyen ympäristöön, teknologioihin ja tietomalliin sekä tarkastellaan algoritmin toteutusta kooditasolla. Luvussa 6 arvioidaan toteutetun järjestelmän onnistumista sekä valittujen teknologiaratkaisuiden soveltuvuutta käyttötarkoituksiinsa tässä projektissa. Luvussa esitellään myös menetelmän ja sen toteutusyksityiskohtien kehitysmahdollisuuksia. Luku 7 on tutkielman yhteenveto.

2. TEOREETTISET LÄHTÖKOHDAT

Tiedon monistaminen on oleellinen osa suunniteltua synkronointimenetelmää ja projektin vaatimusten täyttämistä. Se mahdollistaa tiedon saatavuuden myös tilanteissa, joissa verkkoa ei ole saatavilla. Projektin puitteissa muodostuva järjestelmä on hajautettu: se muodostuu useista itsenäisistä osista. Koska kyseessä on hajautettu järjestelmä, ajan käsittely hajautetuissa järjestelmissä vaatii erityistä huomiota verrattuna yksiosaisiin, ei-hajautettuihin kokonaisuuksiin. Ajan käsittelyn lisäksi järjestelmässä on haasteena se, että järjestelmän tiettyjä resursseja ei voi käyttää kuin yksi järjestelmän osa kerrallaan. Poissulkemisella voidaan rajoittaa haluttujen sovelluskoodin osien samanaikaista suorittamista.

2.1 Tiedon monistaminen

Tiedon monistamisella tarkoitetaan sitä, että jollain laitteella olevaa tietoa kopioidaan toiselle laitteelle. Lopputuloksena on siis tilanne, jossa sama tietosisältö sijaitsee useammalla laitteella.

Tiedon monistaminen laitteelle on perusteltua, kun palvelimen saatavuus on heikko, mutta sovelluksen käytölle on suorituskykyvaatimuksia. Tällöin laitteelle voidaan monistaa suurempia määriä tietoa. Päätelaitteelle monistettu tieto on käytettävissä paikallisesti laitteella ilman yhteyttä palvelimelle. [1]

Tiedon monistaminen päätelaitteelle toimii erityisen hyvin tilanteissa, joissa tietoa on tarkoitus ainoastaan lukea laitteella. [1] Näissä tilanteissa tietosisältöä siirretään kahden laitteen välillä ainoastaan yhteen suuntaan. Mikäli olemassa olevaa tietosisältöä halutaan muokata, poistaa tai luoda päätelaitteella, voidaan tätä muuttunutta tai uutta tietoa joutua järjestelmän vaatimuksista riippuen monistamaan järjestelmän eri osien välillä.

Tilanne, jossa useampi järjestelmän osa on monistanut samaa tietosisältöä, jota ne voivat muokata, voidaan ratkaista synkronoimalla. Synkronoinnin tehtävänä on saada eri laitteiden sisältämät tietosisällöt keskenään samanlaisiksi, jolloin siis yhdellä laitteella tehdyt muutokset ovat siirtyneet muille laitteille. Synkronoinnin keskeiset

haasteet muodostuvat siitä, että kyseessä on monesta itsenäisestä osasta koostuva hajautettu järjestelmä.

2.2 Hajautetut järjestelmät

Hajautettu järjestelmä on useammasta osasta koottu järjestelmä, jonka käyttäjä kokee käyttävänsä yhtä, yhtenäistä kokonaisuutta. Jokainen hajautetun järjestelmän osa on oma itsenäinen kokonaisuutensa, kuten tietokone. Hajautetut järjestelmät voivat olla heterogeenisia, eli ne voivat koostua monista keskenään erilaisista laitteista. Järjestelmän laitekanta voi koostua esimerkiksi sekä suurista keskustietokoneista että pienistä yksiköistä sensoriverkossa. Keskeinen piirre hajautetussa järjestelmässä on, että sen osat voivat kommunikoida jollain tavalla keskenään. [1]

Hajautettuihin järjestelmiin liittyy haasteita verrattuna yhden järjestelmän kokonaisuuksiin. Näistä haasteista synkronointiin liittyy keskeisesti aika sekä poissulkeminen.

2.3 Aika

Ajan käsittely hajautetussa järjestelmässä poikkeaa merkittävästi ajan käsittelystä keskitetyssä järjestelmässä, sillä keskitetyssä järjestelmässä aika on yksikäsitteinen. Tyypillisesti aika on saatavilla käyttöjärjestelmän ytimeltä systeemikutsulla. Näin ollen mikäli prosessi B kysyy aikaa käyttöjärjestelmältä prosessin A jälkeen, saa B vähintään yhtä suuren arvon vastauksena kuin A. Hajautetun järjestelmän osien välillä ei ole olemassa yhteisymmärrystä ajasta. [1] Esimerkiksi kahden prosessin hajautetussa järjestelmässä mikäli prosessilta P1 pyydetään aikaa ensin, ja sen jälkeen prosessilta P2, ei ole taattua että P1:ltä saadun ajan arvo olisi vähintään yhtä pieni kuin P2:ltä saatu ajan arvo.

Lähes jokaisessa tietokoneessa on piiri, joka pitää kirjaa ajasta. Kyseessä ei ole varsinaisesti kello, vaan ajastin, joka tyypillisesti on toteutettu kvartsikiteellä. Kun kiteelle syötetään sopivasti jännitettä, se värähtelee tunnetulla taajuudella. Tätä värähtelyä tulkitsemalla on mahdollista ajastaa eri toimintoja tietokoneella. Tällainen toiminto on esimerkiksi ajastimen arvon kasvattaminen. Vaikka kiteet tyypillisesti toimivat melko vakaasti, ei voida taata, että kaksi kidettä värähtelevät tismalleen samalla taajuudella. [1]

Yhden tietokoneen järjestelmissä aikakäsitys on yksinkertaisesti hallittavissa, sillä tällaisissa järjestelmässä on vain yksi kide ja yksi ajastin. Kun kaikki prosessit käyttävät samaa kelloa, ovat ne sisäisesti yhtenäisiä. Monissa tapauksissa on riittävää,

että tapahtumat ovat suhteellisesti järjestyksessä. Tämä tarkoittaa sitä, että vaikka tietokoneen kello ei vastaa ympäröivän maailman aikakäsitystä, voidaan tapahtumat silti järjestää tietokoneen sisäisesti oikeaan tapahtumajärjestykseen. [1]

Usean tietokoneen järjestelmissä tilanne ei ole yhtä yksinkertainen kuin yhden tietokoneen järjestelmissä. Tällaisissa järjestelmässä on useita ajastimia ja kiteitä. Käytännössä tästä seuraa, että kun järjestelmässä on n tietokonetta, on siinä n kidettä, jotka värähtelevät eri taajuudella. Vaikka tietyllä ajanhetkellä tietokoneiden kellot ovat samassa ajassa, menevät kellot vähitellen eri aikaan, jonka seurauksena samalla ajanhetkellä sovellukset eri koneilla saavat eri kellonajan käyttöjärjestelmältä. [1]

Vaikka kellojen synkronointi on mahdollista, ei ole useinkaan välttämätöntä että kahdella järjestelmän osalla on sama absoluuttinen aika. Absoluuttisella ajalla tarkoitetaan sellaista aikaa, jonka arvo ei perustu mihinkään järjestelmän sisäiseen, vaan johonkin yleisesti tiedettyyn kiintopisteeseen. Esimerkiksi seinäkellomuotoinen aika on absoluuttista. Yleensä ei ole oleellista olla yksimielisyyttä siitä, mitä kello on, vaan siitä, missä järjestyksessä tapahtumat tapahtuvat. [1]

Eräs tapa loogisten kellojen synkronointiin on Lamportin looginen kello. Lamport määritteli tapahtumien a ja b välille suhteen $a \sim b$, joka luetaan "tapahtuma a tapahtuu ennen tapahtumaa b". Tämä tarkoittaa sitä, että kaikki järjestelmän osat ovat yksimielisiä siitä, että tapahtuma a tapahtuu ensin ja vasta sen jälkeen tapahtuma b tapahtuu. Suhdetta voidaan tarkastella kahdessa tilanteessa: [1]

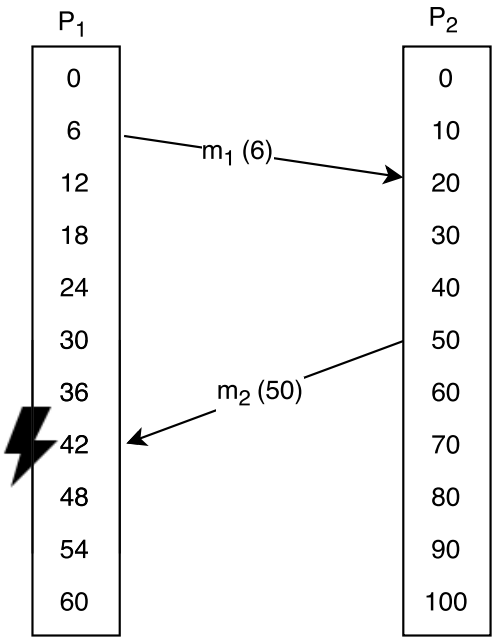
1. Jos tapahtumat a ja b tapahtuvat samassa prosessissa ja a tapahtuu ennen b:tä, $a \sim b$ on tosi.
2. Jos tapahtuma a on tapahtuma jossa prosessi lähettää viestin ja tapahtuma b on tapahtuma jossa toinen prosessi vastaanottaa viestin, $a \sim b$ on tosi.

Relaatio on transitiivinen, eli jos $a \sim b$ ja $b \sim c$, niin $a \sim c$. Mikäli kaksi tapahtumaa (a ja b) tapahtuvat eri prosesseissa ja nämä prosessit eivät kommunikoi keskenään, on sekä $a \sim b$ ja $b \sim a$ epätosia ja tapahtumien sanotaan olevan samanaikaisia. Käytännössä näiden kahden tapahtuman järjestyksestä ei ole tietoa. [1]

Jotta järjestelmän osat voivat olla yhtä mieltä tapahtumien järjestyksestä, täytyy tapahtuman tapahtuma-aika olla sama eri prosesseilla. Merkitään tapahtuman a tapahtuma-aikaa $C(a)$. Nyt jos $a \sim b$, niin $C(a) < C(b)$. Nyt siis jos a ja b ovat samassa prosessissa tapahtuneita tapahtumia ja a tapahtui ennen b:tä, pätee $C(a) < C(b)$. [1]

Koska järjestelmässä on kerrallaan useampi kello, joiden arvo ei kasva tismalleen samaa tahtia, täytyy kellonaikoihin tehdä korjauksia. Menetelmän kannalta on oleellista, että kellonaika C on kasvava. Mahdolliset korjaukset aikaan on tehtävä aina kasvattamalla vanhaa arvoa. [1]

Kuvassa 2.1 esitellään tapaus, jossa kelloa ei korjata ristiriitatilanteessa. Kuvassa P_1 ja P_2 esittävät järjestelmän prosesseja 1 ja 2. P_1:n kello etenee yhden tarkkailujakson aikana 6 yksikköä ja P_2:n 10 yksikköä. Näin ollen P_2:n kello etenee P_1:n kelloa nopeammin, eikä P_1:n kellon arvo ikinä saavuta P_2:n kellon arvoa. Viestiliikenne P_1:sta P_2:iin onnistuu täten aina, sillä kun otetaan huomioon, että viestin kulkemiseen kuluu positiivinen määrä aikaa, viestin saavutettua P_2 on viestin sisältämä lähetysaika aina pienempi kuin P_2 aika. Kuvassa on esimerkkinä viesti m_1, joka lähetettiin P_1:sta ajanhetkellä 6, ja joka vastaanotettiin P_2:ssa ajanhetkellä 20. Nyt lähetyshetkelle a ja vastaanottohetkelle b pätee $a \sim b$ sekä $C(a) < C(b)$, koska $6 < 20$. [1]

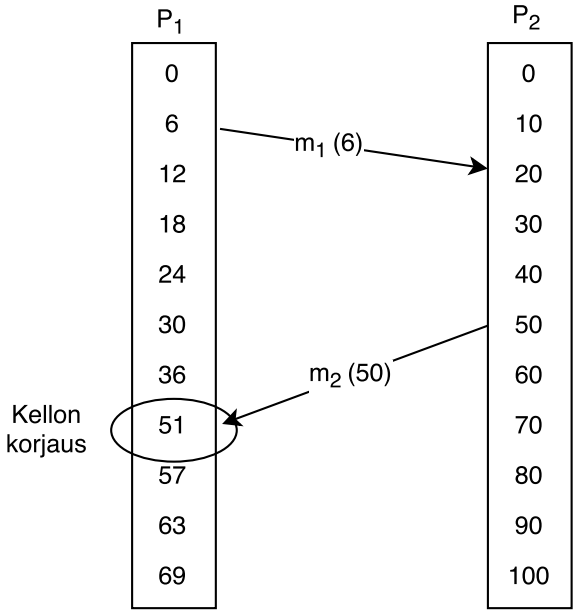


Kuva 2.1. *Järjestelmä, jossa ristiriitatilanteissa kellonaikaa ei korjata.*

Edellisestä poiketen kun P_2 lähettää viestin P_1, päädytään aina ristiriitatilanteeseen. Kuvan esimerkkiviestissä m_2 viesti on lähetetty ajankohtana 50 ja vastaanotettu ajankohtana 42. Nyt ei siis lähetyshetkelle a ja vastaanottohetkelle b ei päde $a \sim b$ sekä $C(a) < C(b)$, koska $50 > 42$. [1]

Kuvassa 2.2 on korjattu kuvassa 2.1 esiintynyt ongelma tekemällä tarvittaessa korjaus kellonaikaan. Viestin m_1 kohdalla järjestelmä toimii tässä samalla tavalla kuin

kuvan 2.1 järjestelmässä. Kellonaikaa korjaamalla aiemmasta tapauksesta poiketen on mahdollista käsitellä viestin m_2 vastaanottamisesta seurannut ristiriitatilanne. Kun P_1 vastaanottaa paketin m_2, on sen kellonaika 42 ja saapuvan viestin 50. Koska tämä aiheuttaa ristiriidan, korjataan prosessin 1 kelloa siten, että lähetyshetkelle a ja vastaanottohetkelle b pätee $a \sim b$ sekä $C(a) < C(b)$. Ehdot voidaan toteuttaa siirtää kelloa eteenpäin arvoon 51. [1]



Kuva 2.2. *Järjestelmä, jossa ristiriitatilanteissa kellonaikaa siirretään eteenpäin, jotta tapahtumien järjestys pysyy eheänä.*

2.4 Poissulkeminen

Koska hajautetuissa järjestelmissä on useita toimivia yksikköjä, samanaikaisuus ja prosessien yhteistoiminta liittyvät siihen vahvasti. Usein tästä seuraa, että useampi prosessi pyrkii samanaikaisesti käsittelemään samaa resurssia. Samanaikaisesta käytöstä voi seurata resurssin meneminen epäeheään tilaan. Tätä tilannetta kutsutaan poissulkemisongelmaksi. [1]

Poissulkemisongelmassa voidaan koodin tasolla erottaa käskyjakso, joka käsittelee ongelman aiheuttavaa tietorakennetta. Tätä käskyjaksoa kutsutaan kriittiseksi alueeksi. Poissulkemisongelman ratkaisemiseksi kriittinen alue on suoritettava yhtenä loogisena operaationa, eli atomisena. [2]

Suoraviivainen tapa toteuttaa poissulkeminen on suorittaa toimenpiteet kriittiseen resurssiin keskitetysti yhden suorittavan osan toimesta. Yhden järjestelmän osan on siis koordinoitava muiden osien aiheuttamia operaatioita resurssiin. [1]

Poissulkeminen hajautetussa järjestelmässä voidaan suorittaa keskitetysti siten, että järjestelmässä on yksi koordinoiva osa. Kun prosessi pyrkii käyttämään jaettua resurssia, se pyytää koordinaattorilta käyttölupaa resurssiin. Mikäli samaan aikaan useampi kuin yksi prosessi pyrkii käyttämään samaa jaettua resurssia, voi koordinaattori antaa yhdelle prosessille pääsyn resurssiin ja laittaa muut prosessit jonottamaan vuoroaan päästä käyttämään resurssia. Jonotus voi tapahtua esimerkiksi niin, että prosessit ilmoittaa halukkuutensa käyttää resurssia ja koordinaattori lähettää viestin prosessille, kun on sen vuoro. Kulloinkin vuorossa olevan prosessin tulisi ilmoittaa koordinaattorille, milloin se ei enää käytä resurssia. Tällä tavoin toteutettuna prosessit odottavat koordinaattorin ilmoitusta käyttöluvasta. Vaihtoehtoisesti koordinaattori voi lähettää prosessille välittömästi hylkäävän vastauksen, mikäli resurssi on jo käytössä. Tällöin prosessin vastuulle jää tiedustella resurssin saatavuutta uudelleen. [1]

Huonona puolena keskitetyssä algoritmissa on, että järjestelmässä on yksittäinen osa, koordinaattori, jonka vikaantuminen estää koko järjestelmän toiminnan. Mikäli resursseja jonotetaan, ei pääsyä resurssiin odottava prosessi välttämättä tule tietoiseksi, mikäli jonotuksen aikana koordinaattori on mennyt vikatilaan. Koordinaattorista voi tulla myös järjestelmän suorituskykyä rajoittava tekijä. Keskitetyn algoritmin suurin vahvuus on sen yksinkertaisuus. Yksinkertaisen luonteensa takia se on usein käytännöllinen menetelmä poissulkemisen toteuttamiseen huolimatta sen heikkouksista. [1]

3. OLEMASSAOLEVAT RATKAISUT

Projektin alkuvaiheessa tutustuttiin saatavilla oleviin valmiisiin ratkaisuihin. Saatavilla olevista vaihtoehdoista Realm Mobile Platform koettiin mahdolliseksi ratkaisuksi asetetuille vaatimuksille. Realm Mobile Platform on valmis kokonaisratkaisu, joka on saatavilla itsenäisenä tuotteena. Lisäksi tutustuttiin saatavilla oleviin julkaistuihin algoritmeihin, joista tämän työn aiheeseen sopivaksi havaittiin Ashish Kedian ja Anusha Prakashin vuonna 2015 julkaisema synkronointialgoritmi. Kyseisellä algoritmilla pyrittiin ratkaisemaan samankaltaista ongelmaa kuin tässä työssä.

Tässä luvussa tutustutaan tarkemmin saatavilla oleviin ratkaisuihin ja arvioidaan niiden soveltuvuutta sovelluksen tarpeisiin. On syytä huomioida, että tuotteita kehitetään jatkuvasti ja tilanne yksittäisen tuotteen soveltumisen suhteen voi muuttua ajan myötä.

3.1 Realm Mobile Platform

Realm Mobile Platform on Realmin kehittämä ratkaisu mobiililaitteen tietokannan synkronointiin. Ratkaisu koostuu päätelaitteella toimivasta Realm-tietokannasta ja palvelimelle asennettavasta Realm Object Server -palvelimesta. [3]

Realm Mobile Platformin lisäksi on mahdollista käyttää omaa taustajärjestelmää, joka voi kommunikoida sekä Realm Mobile Platformin palvelimen että asiakassovelluksen kanssa. [3] Näin ollen se mahdollistaisi kokonaisuuden räätälöinnin monipuolisesti erilaisiin käyttökohteisiin.

Ream Mobile Platform on tehty niin sanotuksi ”offline first”-alustaksi. Tämä tarkoittaa sitä, että asiakassovellus kopioi itselleen palvelimelta tietosisällön. Asiakassovelluksella tehdyt muutokset tehdään sen omiin kopioihin objekteista. Realm-tietokanta synkronoi muuttuneet tiedot palvelimelle asynkronisesti. Kaikki muutokset tehdään deterministisesti. Sovelluksen saatavuus on korkea riippumatta esimerkiksi verkkoyhteyden toimivuudesta, sillä kaikki luku- ja kirjoitusoperaatiot tapahtuvat ensisijaisesti laitteella paikallisesti. [4]

Realm Mobile Platform tarjoaa tiedon synkronoinnin palvelimen sekä asiakassovellusten välillä. Synkronointitoiminto on pitkälti automatisoitu, ja sen käyttöönotto Realm-tietokantaa käyttävässä sovelluksessa on varsin yksinkertaista. Mitä enemmän oletustoiminnallisuudesta poikkeavaa toiminnallisuutta tarvitaan, sitä enemmän alustaa tarvitsee konfiguroida.

Projektissa ei päädytty käyttämään Realm Mobile Platformia, sillä sitä ei oltu julkaistu hetkellä, jona teknologiavalintoja tehtiin. Tämän työn kirjoitushetkellä Realm Mobile Platform on julkaistu, mutta monet sen ominaisuudet ovat vielä beta-asteella. Se on lisäksi vielä varsin uusi tuote, joten sitä ei ole testattu tuotantokäytössä vielä tarpeeksi, jotta sen voisi ottaa käyttöön tähän sovellukseen. Koska asiakassovelluksessa käytetään Realm-tietokantaa, on sovelluksella hyvät edellytykset Realm Mobile Platformin käyttämiselle. Realm Mobile Platformin soveltuvuutta projektin tarpeisiin tullaan todennäköisesti arvioimaan tarkemmin, kunhan sen arvioidaan olevan kyllin testattu ja ominaisuuksiensa puolesta valmis.

3.2 Ashish Kedia ja Anusha Prakash - synkronointialgoritmi

Ashish Kedia ja Anusha Prakash julkaisivat vuonna 2015 artikkelin "Data Synchronization on Android Clients", jossa he esittelevät kaksisuuntaisen synkronointimenetelmän Android-asiakaslaitteiden ja palvelimen välillä. Menetelmä pyrkii olemaan suorittimen käyttöasteen, virrankulutuksen ja verkon käytön kannalta taloudellinen [5].

Menetelmässä jokaiseen tietueeseen lisätään taulukon 3.1 mukaiset kentät. Näitä kenttiä käytetään pitämään kirjaa siitä, mitä tietueita kulloinkin tarvitsee siirtää, jotta järjestelmä on täysin synkronoituneessa tilassa. Kenttien tilaa tulee ylläpitää esimerkiksi tietokantatriggereiden avulla. Käytännössä esimerkiksi "*created at*"-aikaleima tulee luoda tietueelle, kun se luodaan. [5]

Taulukko 3.1. Lista tietueen tarkkailuun käytettävistä kentistä. [5]

Kenttä	**Selitys**	**Säilytyspaikka**
created at	Luonnin aikaleima	Molemmat
updated at	Päivityksen aikaleima	Molemmat
sync status	Synkronoinnin tila	Molemmat
client id	Pääavain asiakassovelluksessa	Palvelin
server id	Pääavain palvelimella	Asiakassovellus

"*Sync status*"-kenttä on tietotyypiltään enumeraali, eli se voi sisältää jonkin ennalta määrätyistä arvoista. Sen mahdolliset arvot on listattu taulukkoon 3.2. [5]

Taulukko 3.2. Synkronoinnin tilan mahdolliset arvot.

Sync Status	**Selitys**
Synchronized	Ei muutoksia edellisen synkronoinnin jälkeen
Updated	Sisältöä päivitetty edellisen synkronoinnin jälkeen
Inserted	Lisätty edellisen synkronoinnin jälkeen
Deleted	Poistettu edellisen synkronoinnin jälkeen
Update Sent	Mahdollinen päivityksen epäonnistuminen viime synkronoinnissa
Insert Sent	Mahdollinen lisäämisen epäonnistuminen viime synkronoinnissa
Delete Sent	Mahdollinen poistamisen epäonnistuminen viime synkronoinnissa
Server Deleted	Palvelin on merkinnyt poistetuksi

Palvelimella tulee lisäksi olla tieto jokaisen käyttäjän viimeisimmän synkronointikerran ajankohdasta. Tätä tietoa ei säilytetä lainkaan käyttäjän laitteella. [5]

Synkronointi aloitetaan asiakassovelluksen puolelta säännöllisin väliajoin tai jollain muulla menetelmällä. Palvelin voi esimerkiksi aloittaa sen lähettämällä asiakassovellukselle push-viestin. Menetelmässä asiakassovellus lähettää oman tietokantansa uudet, muuttuneet ja poistetut tietueet ensin, jonka jälkeen se hakee palvelimelta muualla muuttuneet tietueet. [5]

Asiakassovellus hakee synkronoimattomat tietueet ja järjestää ne päivitysajankohdan mukaan nousevaan järjestykseen. Näin ollen listan alkupäässä on aikaisemmin muuttuneet tietueet. Algoritmissa on määritelty suurin kerrallaan lähetettävien tietueiden määrä sekä asiakassovellukselle että palvelimelle. Synkronoimattomat tietueet jaetaan korkeintaan maksimikokoisiin eriin. [5]

Kun tietueita lähetetään palvelimelle, merkitään niiden "Sync status-kentän arvoksi tapahtunutta muutosta kuvaava arvo. Esimerkiksi lisätylle tietueelle merkitään tila "*Insert Sent*"ja päivitetylle "*Update Sent*". [5]

Kun palvelin vastaanottaa muutokset asiakassovellukselta, tekee se muutokset omaan kantaansa. Mikäli tietueen tila on "*Insert Sent*", luo palvelin uuden tietueen ja luo sille uuden pääavaimen. Palvelin vastaa asiakassovelluksen lähettämään viestiin viestillä, jossa on luotujen tietueiden pääavaimet. Tietueen tilan ollessa "*Update Sent*", hakee palvelin tietokannastaan vastaavan tietueen pääavaimen avulla ja päivittää sen sisältöä. Vastauksena palvelin lähettää asiakassovellukselle päivitysaikaleiman, eli *update_at*-kentän arvon. Poisto-operaatiossa, eli tilan ollessa "*Delete Sent*" palvelin merkitsee tietueen poistettavaksi ja lähettää tietueen pääavaimen vastauksena. [5]

Mikäli palvelimen vastaus ei saavuta asiakassovellusta, asiakassovellus saattaa lähettää samat päivittyneet tietueet kahdesti. Tästä ei seuraa kuitenkaan kahden tietueen luomista tilan ollessa "*Insert Sent*", sillä Palvelin tallentaa taulukon 3.1 mukaisesti tietueen asiakassovelluksen pääavaimen ja näin ollen voidaan havaita, että tietue on jo luotu. [5]

Mahdolliset ristiriidat ratkaistaan palvelimella. Ristiriita tapahtuu, kun sekä palvelimella että asiakassovelluksessa olevalla tietueella on synkronoimaton tietue tai kun tietuetta ollaan muokattu useasta asiakassovelluksesta. Ristiriitojen ratkaisuun käytetään taulukon 3.1 mukaisia kenttiä. Tietueissa voi olla myös muita kenttiä, kuten sekvenssinumero tai looginen kello auttamassa ratkaisemaan ristiriitatilannetta. Tietueen lopullisen tilan tulee propagoitua sekä palvelimelle, että kaikille asiakassovelluksille. [5]

Saatuaan vastauksen palvelimelta, asiakassovellus varmistuu siitä, että muutokset on tallennettu palvelimelle onnistuneesti. Asiakassovellus päivittää tietueisiinsa vastaanottamansa palvelimen pääavaimet, joita vastaavan tietueen palvelin loi synkronoinnin aikana ("*Insert sent*"-tilalliset). Kaikkien päivitettyjen tietueiden *updated_at*-arvo päivitetään palvelimelta saatuun aikaleimaan. Saatuaan poistetun tietueen pääavaimen, asiakassovellus voi turvallisesti poistaa tietueen tietokannastaan. Kaikkien synkronoitujen tietueiden tilaksi asetetaan *Synchronized*. [5]

Kun asiakassovellus on lähettänyt kaikki muuttuneet tietueet palvelimelle, alkaa se pyytämään palvelimelta päivittyneitä tietueita. Palvelin lähettää määritetyn maksimimäärän tietueita vastauksena asiakassovellukselle samalla tavalla, kuin asiakassovellus lähetti palvelimelle. Asiakassovellus päivittää omaa tietokantaansa palvelimen vastauksen mukaan, ja lähettää palvelimelle vahvistuksen päivitetyistä tietueista. Palvelin päivittää kuittausviestissä olevien tietueiden tilat synkronoiduiksi. Vastausviestinä palvelin lähettää seuraavan erän muuttuneita tietueita. Tätä jatketaan, kunnes palvelin on lähettänyt kaikki muuttunet tietueet. Tällöin palvelin lähettää tyhjän vastauksen ja tallentaa tiedon synkronoinnin ajankohdasta. [5]

4. ALGORITMI

Synkronointi on monivaiheinen prosessi, jonka lopputuloksena asiakassovelluksen tila vastaa taustajärjestelmän tilaa käyttäjän omistaman tietosisällön osalta. Esitellyssä menetelmässä synkronointi tapahtuu aina asiakassovelluksen ja taustajärjestelmän välillä. Näin ollen muutokset välittyvät asiakassovellukselta toiselle ainoastaan palvelimen taustajärjestelmän kautta.

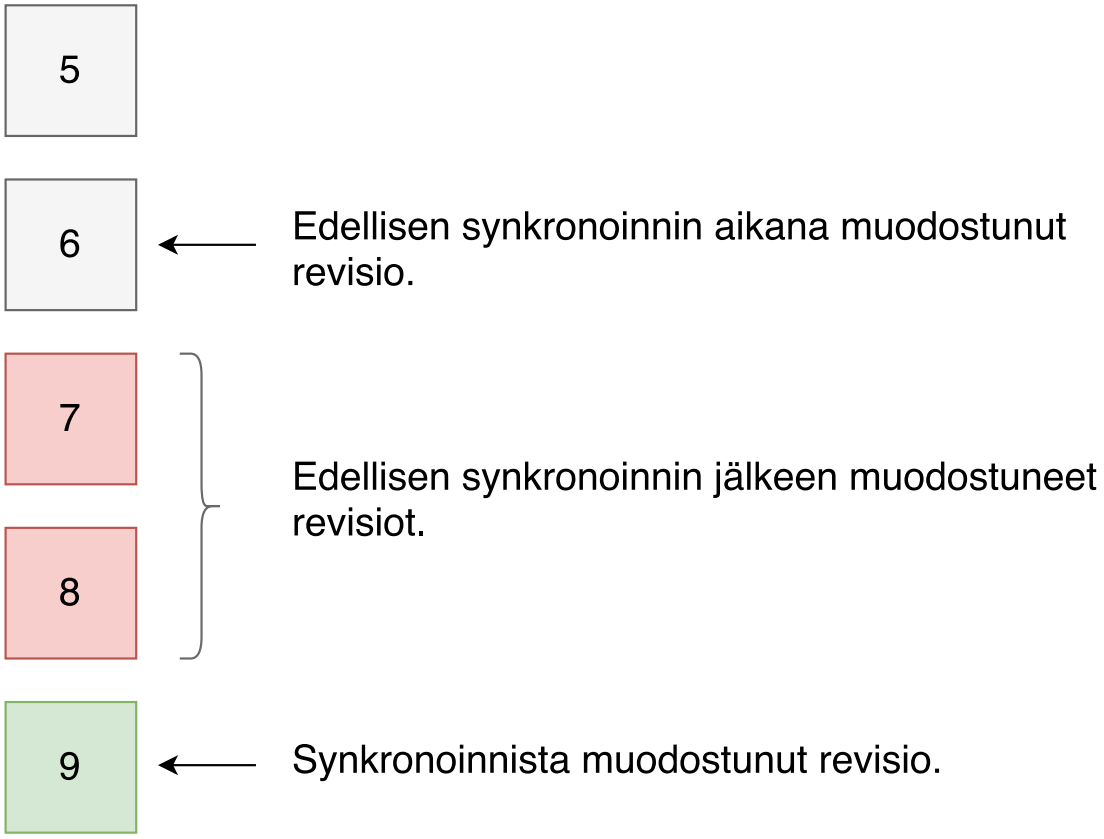
Synkronoinnin aloittaakseen asiakassovellus lähettää sen hallussa olevat muuttuneet ja uudet objektit sekä edellisen synkronointikerran revisionumeron taustajärjestelmälle ja saa vastauksena muiden järjestelmän osien muuttamat ja luomat objektit. Mikäli asiakassovellus ei ole suorittanut synkronointia aikaisemmin, se ei lähetä revisionumeroa taustajärjestelmälle. Sovellus merkitsee lähetettävät objektit *uploading*-tilaan.

Taustajärjestelmä kasvattaa synkronointia suorittavan käyttäjän käyttäjäkohtaista revisionumeroa yhdellä. Tämän jälkeen se käsittelee vastaanottamansa objektit sellaisessa järjestyksessä, jossa objekteja voidaan tallentaa tietokantaan niiden välisiä riippuvuussuhteita rikkomatta. Käytännössä siis objektit käsitellään siten, että aloitetaan sellaisista, joilla ei ole riippuvuuksia ja edetään sitten objekteihin, jotka riippuvat jo käsitellyistä objekteista ja niin edelleen. Taustajärjestelmä tarkastaa, onko sen tietokannassa jo samalla tunnisteella yksilöity objekti. Mikäli objekti löytyy, eikä sitä ole merkitty poistetuksi, päivitetään sen tiedot vastaanotetun objektin tiedoilla. Jos objektia ei löydy, tallennetaan tietokantaan uusi objekti vastaanotetun objektin tiedoilla. Tallennuksen yhteydessä muokattaville tai luotaville objekteille asetetaan revisionumeroksi käyttäjän revisionumero.

Taustajärjestelmän asiakassovellukselle vastauksena lähettämät objektit valitaan sen mukaan, onko kyseessä asiakassovelluksen ensimmäinen synkronointikerta vai olemassa olevan tietosisällön päälle tehtävä päivitys. Mikäli kyseessä on asiakassovelluksen ensimmäinen synkronointikerta, haetaan tietokannasta kaikki käyttäjän objektit, joita ei ole merkitty poistetuiksi. Jos kyseessä on päivitys, haetaan tietokannasta objektit, joiden revisio on suurempi kuin edellisen synkronointikerran seurauksena muodostunut revisio, mutta pienempi kuin meneillään olevan synkronoin-

nin seurauksen muodostunut revisio. Näistä haetaan myös poistettujen objektien tunnisteet, jotta poistetut objektit voidaan poistaa asiakassovelluksesta.

Kuvassa 4.1 esitellään visuaalisesti revisionumerot synkronointitilanteessa. Revisio 9 on muodostunut meneillään olevan synkronoinnin seurauksena. Synkronoinnissa asiakassovelluksen lähettämät muutokset siis on tallennettu tälle revisiolle. Revisio 6 muodostui saman asiakassovelluksen edellisellä synkronointikerralla ja asiakassovellus lähetti sen taustajärjestelmälle meneillään olevan synkronointikerran aloittavassa verkkopyynnössä. Näiden tietojen perusteella tiedetään, että asiakassovelluksella on kaikki muutokset, jotka ovat tapahtuneet revisiossa 6 tai sitä aikaisemmin sekä muutokset, jotka ovat tapahtuneet revisiossa 9. Näin ollen tiedetään, että asiakassovellukselta puuttuu muutokset, jotka on tehty revisioiden 6 ja 9 välillä. Objektit, joiden revisio on 7 tai 8 valitaan siis lähetettäväksi asiakassovellukselle synkronoinnin vastauksessa.



Kuva 4.1. *Revisioiden lähihistoria synkronointitilanteessa.*

Kun asiakassovellus on vastaanottanut taustajärjestelmältä saamansa listat, muokkaa se tietosisältöään niiden pohjalta: uudet objektit lisätään tietokantaan, muokatut objektit päivitetään tietokantaan sekä poistetuiksi merkityt objektit poistetaan tietokannasta. Asiakassovellus tallentaa synkronoinnin seurauksena muodostuneen

revision numeron, jotta se voi lähettää sen taustajärjestelmälle seuraavan synkronointipyynnön yhteydessä.

Onnistuneen synkronoinnin päätteeksi lähetetyt objektit merkitään *ok*-tilaan osoittamaan, että ne ovat synkronoidut taustajärjestelmän kanssa onnistuneesti. Mikäli synkronoinnin havaitaan epäonnistuneen, merkitään objektit takaisin *modified*-tilaan, jotta ne lähetettäisiin taustajärjestelmälle uudelleen seuraavan synkronointikerran yhteydessä.

5. TOTEUTUS

Menetelmän toteutuksen kuvaus voidaan jakaa kolmeen kokonaisuuteen: ympäristöön, teknologioihin ja tietomalliin. Ympäristöllä tarkoitetaan infrastruktuuria ja laitteistoa, jossa sovellusta käytetään. Teknologioihin lukeutuvat ohjelmointikielet, sovelluskehykset ja ohjelmistot, joita hyödynnetään menetelmän toteutuksessa. Tietomalli esittelee mallia, jossa menetelmän edellyttämää tietoa säilytetään käytännön tasolla toteutuksen eri osissa. Tässä luvussa esitellään keskeisimmät osat edellä mainituista aihealueista.

5.1 Ympäristö

Sovellus voidaan jakaa kahteen kokonaisuuteen: taustajärjestelmään ja asiakassovellukseen. Taustajärjestelmä on pilvipalvelu, joka tarjoaa tarvittavat rajapinnat sovelluksen toiminnalle. Asiakassovellus on älypuhelimella ajettava sovellus, joka kommunikoi taustajärjestelmän kanssa.

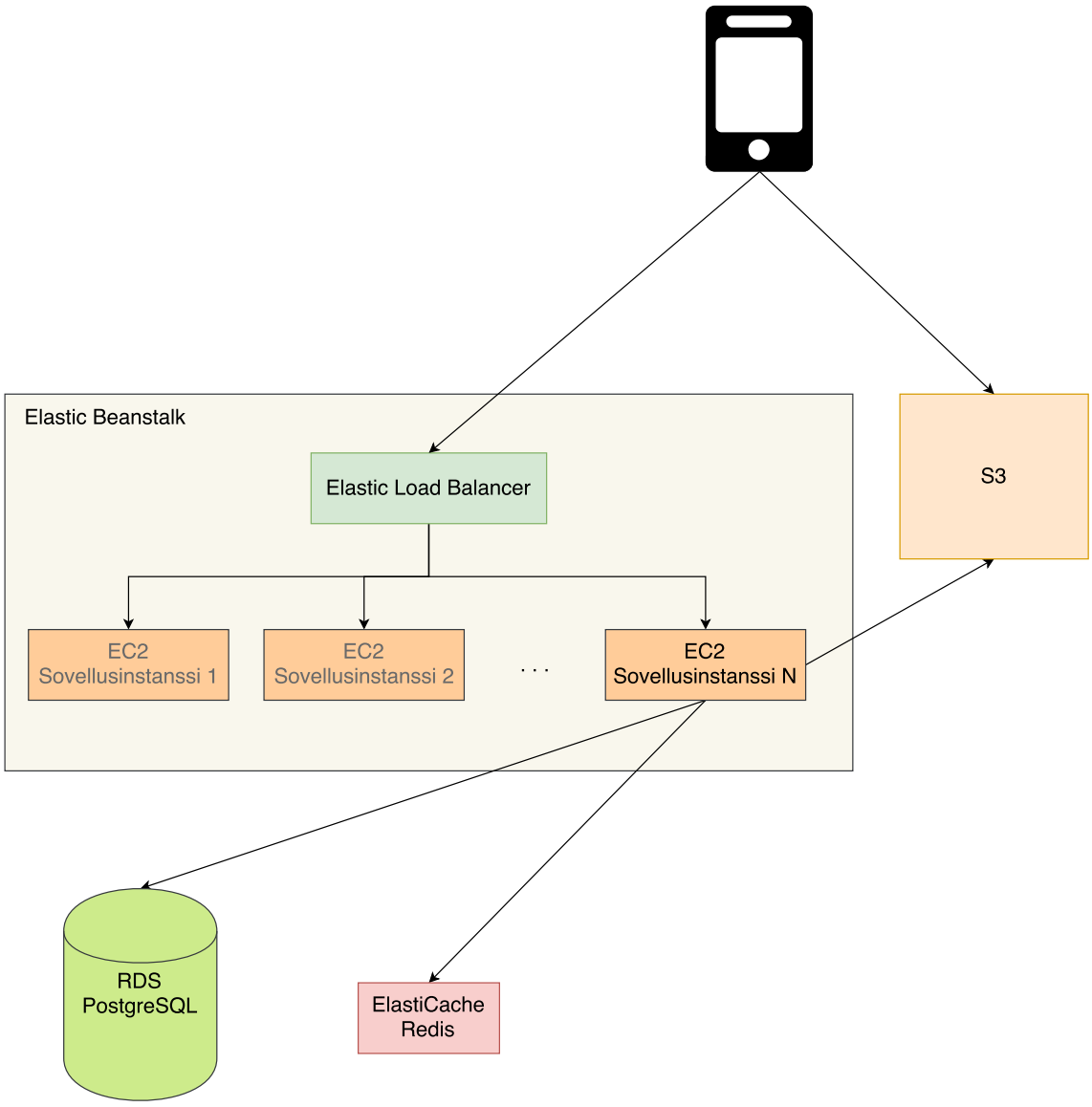
Taustajärjestelmä koostuu useista palveluista sekä itse sovelluskoodista. Käytettäviin palveluihin lukeutuu muun muassa virtuaalikone-, tietokanta- sekä välimuistipalvelut. Modulaarinen rakenne mahdollistaa muutostöiden tekemisen joustavuuden.

Asiakassovellus on iPhone-puhelimessa ajettava sovellus, joka käyttää taustajärjestelmän palveluita. Kun sovellukseen on luotu käyttäjätunnus ja kirjauduttu, voidaan sitä käyttää itsenäisesti ilman taustajärjestelmää. Kun taustajärjestelmä on käytettävissä, voi sovellus käyttää sitä sosiaalisten toimintojen tarjoamiseen sekä tiedon varmuuskopiointiin ja jakamiseen käyttäjän laitteiden välillä.

5.1.1 Taustajärjestelmä

Sovelluksen taustajärjestelmän alustana käytetään Amazon Web Servicesiä (AWS). AWS on 2006 perustettu palvelu, joka tarjoaa pilvipalveluita. Toimintamallin vahvuutena on liiketoiminnan mukaan skaalautuvat kustannukset sekä mahdollisuus nopeisiin IT-infrastruktuurimuutoksiin.

Taustajärjestelmän toimintaympäristönä käytetään Amazonin Elastic Beanstalk -palvelua. Elastic Beanstalk on palvelu, joka koostaa AWS:n eri palveluista koostuvan kokonaisuuden. Sen tarkoituksena on helpottaa Amazonin palveluiden käyttöönottoa ja käyttöä muun muassa huolehtimalla toteutusyksityiskohdista käyttäjän puolesta. Se tarjoaa automaattisesti muun muassa kapasiteettiprovisioinnin, kuormantasauksen, automaattisen skaalauksen ja sovelluksen kunnon tarkastuksen [6]. Ylläpitotoimien helpottamiseksi Elastic Beanstalk tarjoaa myös yhtenäisen ylläpitokäyttöliittymän sen kautta luoduille palvelukokonaisuuksille. Kuva 5.1 esittää taustajärjestelmän arkkitehtuurin AWS-ympäristössä.



Kuva 5.1. *Taustajärjestelmän arkkitehtuuri AWS-ympäristössä. Kuvan selkeyttämisen vuoksi vain sovellusinstanssi N:n suhteet muihin palveluihin on merkitty.*

Elastic Beanstalk mahdollistaa muun muassa sovelluksen nopean päivittämisen. Ympäristön asetuksista on mahdollista asettaa ympäristön päivitysmenetelmä usean palvelimen ympäristössä. Elastic Beanstalk voi esimerkiksi päivittää puolet ympäristön palvelimista kerrallaan ja näin ollen palveluun ei tule käyttökatkoa päivityksen aikana.

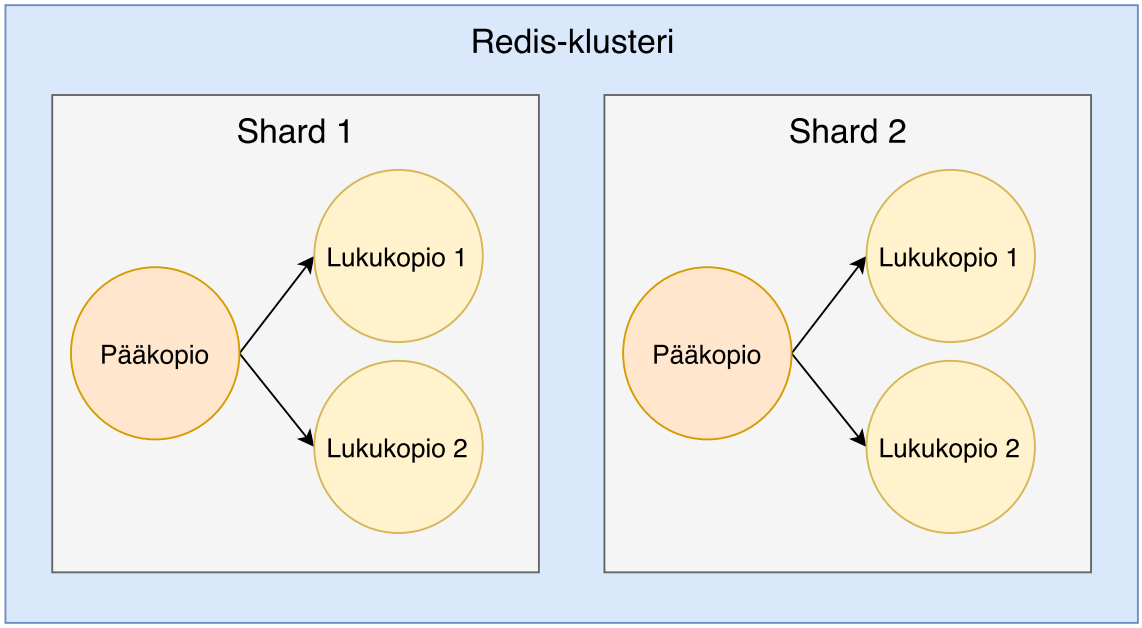
Amazon tarjoaa Elastic Beanstalkille komentorivityökalun, jolla on mahdollista suorittaa ympäristön ylläpitotoimia komentoriviltä [7]. Komentorivityökalulta on mahdollista esimerkiksi päivittää ympäristön palvelimille uusin sovellusversio paikallisesta Git-versionhallinnasta [8].

Elastic Beanstalk tekee palvelinsovellukselle halutun määrän palvelininstansseja Amazon Elastic Compute Cloud (Amazon EC2) -palveluun. EC2 on palvelu, joka tarjoaa mahdollisuuden luoda ja hallita virtuaalipalvelimia. Virtuaalipalvelimen suorituskykytason ja käyttöjärjestelmän voi valita laajasta valikoimasta. Palvelu veloitetaan minuuttikohtaisesti ja minuutin hinta riippuu muun muassa valitusta laitteistotasosta, käyttöjärjestelmästä ja laitteiston fyysisestä sijainnista. Koska palvelu on minuuttivelotteinen, voidaan kustannuksia optimoida skaalaamalla järjestelmän kokonaissuorituskykyä tarpeen mukaan.

EC2:n yhteyteen on mahdollista ottaa käyttöön Elastic Load Balancing. Elastic Load Balancing -palvelulla voidaan automatisoida palvelun suorituskyvyn skaalautumista. Palvelussa voidaan määrittää raja-arvot, joissa palvelininstanssien määrää nostetaan tai lasketaan. Palvelu ottaa siis tarvittaessa käyttöön uusia palvelininstansseja ja asentaa niihin halutut sovellukset. Vastaavasti instansseja voidaan sammuttaa sitä mukaa, kun niitä ei enää tarvita ja näin ollen kustannuksia voidaan madaltaa, kun järjestelmä on vähäisemmällä käytöllä. Elastic Load Balancing mahdollistaa myös esimerkiksi kaatuneiden palvelininstanssien havaitsemisen, ja osaa ohjata liikenteen muihin instansseihin kunnes kaatunut instanssi on saatu jälleen toimintakuntoon. [9]

Amazon ElastiCache on AWS:n palvelu, joka tarjoaa nopean välimuistipalvelun sovellusten käytettäväksi. ElastiCache tukee Redis- sekä Memcached-tietosäiliöitä. Taustasovelluksen välimuistina on käytetty Redistä, joka on avain-arvo-varasto, joka säilyttää arvoja keskusmuistissa. ElastiCachea voidaan käyttää usein tietokannasta haettavan tiedon välimuistitallennukseen. Välimuistin hakuajat ovat relaatiotietokantaa huomattavasti nopeampia. Lisäksi välimuistiin voidaan tallentaa tietoa, jonka muodostaminen relaatiotietokannasta vaatisi useamman kuin yhden kyselyn. Näin ollen kun välimuistiin tallennetaan usein luettavaa tietoa, voidaan palvelun suorituskykyä parantaa merkittävästi.

ElastiCachea on mahdollista skaalata suorituskykyisemmäksi lisäämällä järjestelmään lukukopiota sekä lisäämällä shardien lukumäärää klusteroimalla kuvan 5.2 esittämällä tavalla. Shardiksi kutsutaan yhtä loogista kokonaisuutta, joka sisältää yhden pääkopion sekä 0–5 lukukopiota välimuistista. Shardissa olevien kopioiden tarkoituksena on siis parantaa shardin lukusuorituskykyä. Kirjoitussuorituskykyä voidaan parantaa käyttämällä klusterointia, jolloin shardien lukumäärää voidaan kasvattaa 15 asti. Käyttäessä klusterointia on mahdollista ottaa käyttöön maksimissaan 15 shardia. [10] Klusteroituna joukko shardeja saa yhden rajapinnan ja näin ollen näyttää yhdeltä loogiselta Redis-välimuistilta. Tallennustilaa on saatavilla suurimmillaan 3,55 tebitavua [11].



Kuva 5.2. *Redis-klusteri, jossa on kaksi shardia ja kaksi lukukopiota jokaista shardia kohden.*

Taustajärjestelmän tietokanta on Amazonin RDS-palvelussa. RDS tarjoaa eri relaatiotietokantoja ja näihin liittyviä palveluita. RDS:n tarjoamia tietokannanhallintajärjestelmiä ovat muun muassa Amazon Aurora, PostgreSQL ja MySQL. [12]

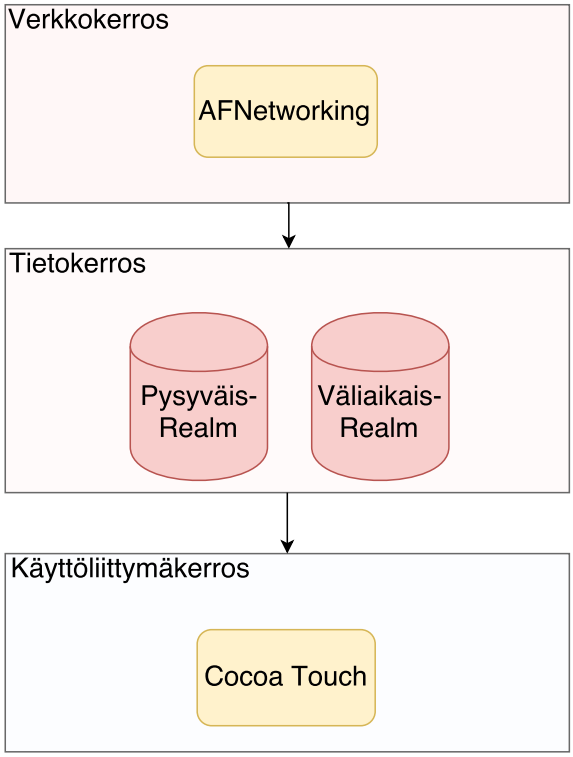
RDS:n työkalujen avulla on mahdollista esimerkiksi vaihtaa tietokantapalvelin tehokkaampaan, mikäli käytössä oleva palvelinratkaisu ei toimi riittävän nopeasti esimerkiksi käyttäjämäärän kasvaessa. Vastaavasti palvelin voidaan vaihtaa tarvittaessa hitaampaan ja halvempaan. Suorituskykyä voidaan tarvittaessa kasvattaa myös ottamalla päätietokantapalvelimen rinnalle lukukopioita. Lukukopioiden avulla lukuoperaatiot voidaan jakaa useamman tietokantapalvelimen kesken, jolloin lukunopeudet kasvavat varsinkin ruuhkatilanteissa. Lukukopiot voidaan myös sijoittaa

maantieteellisesti eri paikkoihin, jolloin verkkoviivettä on mahdollista pienentää lukuoperaatioissa. Lisäksi varmuuskopioiden ottamisen automatisointi ja varmuuskopioiden säilytys kuuluu palveluvalikoimaan, joka helpottaa ylläpitotyötä. [12]

Vaikka Elastic Beanstalk -ympäristöstä on mahdollista luoda tietokantapalvelin, Amazon ei suosittele sitä tuotantokäyttöön. Mikäli tietokantapalvelin on luotu ympäristön sisälle, on sen elinkaari sidoksissa ympäristöön. Lisäksi Elastic Beanstalk -ympäristön tietokantapalvelinta ei voi vaihtaa sen jälkeen, kun se on kerran asetettu. Koska RDS-tietokannan varmuuskopion palauttaminen tapahtuu käytännössä luomalla uusi tietokantapalvelin varmuuskopiovedoksen (snapshot) pohjalta, ei Elastic Beanstalk -ympäristöön luodun tietokannan varmuuskopion palauttaminen ole nopea operaatio. Toimenpide käytännössä vaatii kokonaan uuden ympäristön luomista, johon otettaisiin käyttöön varmuuskopiovedoksesta palautettu tietokanta. Tämän jälkeen liikenne ohjataan uuteen ympäristöön ja vanha ympäristö poistetaan. [13]

5.1.2 Asiakassovellus

Asiakassovelluksen arkkitehtuuri voidaan karkeasti jakaa kolmeen osaan: käyttöliittymäkerrokseen, tietokerrokseen sekä verkkokerrokseen. Kuvassa 5.3 on esitetty asiakassovelluksen arkkitehtuuri korkealla tasolla.



Kuva 5.3. *Korkean tason kuvaus asiakassovelluksen rakenteesta.*

Käyttöliittymäkerroksena käytetään Applen Cocoa Touch -sovelluskehystä. Cocoa Touch tarjoaa kattavasti eri käyttöliittymäkomponentteja sekä käyttäjän syötteen käsittelyä helpottavia rajapintoja. [14]

Tietokerroksessa keskeisenä teknologiana on Realm, joka on ilmainen, avoimen lähdekoodin tietokanta [15]. Tietokanta mahdollistaa tiedon pysyväistallennuksen, jolloin verkosta ladattu ja laitteella tuotettu tietosisältö säilyy, vaikka sovelluksen käynnistäisi uudelleen. Tietokerroksella on kaksi erillistä Realm-tietokantaa: yksi pysyvästi säilytettävälle tiedolle ja yksi väliaikaisesti säilytettävälle tiedolle. Osa käsiteltävästä tiedosta säilytetään myös muistinvaraisina objekteina. Aiheena olevan algoritmin kannalta tietovarastoista ainoastaan pysyväissäilytykseen tarkoitettu Realm-tietokanta on oleellinen.

Asiakassovellus käyttää taustajärjestelmän kanssa kommunikointiin HTTP-protokollaa. Verkkokerroksella käytetään AFNetworking-kirjastoa, joka yksinkertaistaa HTTP-pyyntöjen lähettämistä ja käsittelyä [16]. AFNetworking tarjoaa korkean tason rajapinnan Applen Foundation -sovelluskehyksen URL Loading System:lle, jota käytetään palvelinten kanssa kommunikointiin standardeilla internet-protokollilla [16; 17].

5.2 Teknologiat

Tässä luvussa esitellään teknologiat, joilla tässä diplomityössä esitelty tiedon synkronointimenetelmä on toteutettu. Projektista käytettävistä teknologioista esitellään oleellisimmat.

5.2.1 HTTP

Taustajärjestelmä ja asiakassovellus käyttävät HTTP-protokollaa kommunikointiin. HTTP on monikäyttöinen sovellustason protokolla, jonka keskeinen tunnuspiirre on sen tilattomuus. Tämä tarkoittaa sitä, että protokollatasolla yksittäisten pyyntöjen välillä ei ole jaettua tilaa, vaan pyynnöt käsitellään aina itsenäisinä kokonaisuuksinaan. [18]

Pyyntö on viesti, jonka asiakas lähettää palvelimelle. Viestillä on aina jokin metodi, jolla voidaan määrittää pyynnön luonnetta. Metodeita ovat esimerkiksi GET ja POST. Näistä GETiä käytetään tyypillisesti tiedon hakemiseen palvelimelta ja POSTia tiedon tallettamiseen palvelimelle. Pyyntö kohdistetaan aina johonkin osoitteeseen palvelimella. Esimerkiksi

```
GET http://www.vincit.fi/index.php
```

pyytää www.vincit.fi-palvelimelta index.php-nimistä tiedostoa. [18] Kun palvelin on vastaanottanut pyynnön, se vastaa asiakkaalle vastausviestillä. Keskeisimmät osat vastausta ovat status-koodi sekä sisältö. Status-koodi on kolminumeroinen luku, jolla voidaan ilmaista verkkopyynnön onnistumista. Koodit jaetaan viiteen luokkaan ensimmäisen numeron perusteella taulukon 5.1 mukaisesti.

Taulukko 5.1. HTTP-vastauskoodit. [18]

Koodi	**Tyyppi**	**Kuvaus**
1xx	Informatiivinen	Pyyntö on vastaanotettu ja sen prosessointia jatketaan.
2xx	Onnistuminen	Toiminto on vastaanotettu, ymmärretty ja hyväksytty.
3xx	Uudelleenohjaus	Lisätoimia tarvitaan, jotta pyyntö voidaan suorittaa.
4xx	Asiakasvirhe	Pyynnön syntaksi on viallinen tai sitä ei voida muusta syystä suorittaa.
5xx	Palvelinvirhe	Palvelin ei onnistunut suorittamaan näennäisesti kelvollista pyyntöä.

Vastauksen sisältöosuus on vapaaehtoinen osa vastausta. Sitä voidaan käyttää tiedon siirtämiseen palvelimelta asiakkaalle.

5.2.2 Objective-C

Objective-C on Applen ylläpitämä ohjelmointikieli, jota käytetään nykyään pääasiassa Mac OS- ja iOS-kehitykseen. Se on C-kielen ylijoukko, eli kaikkia C:n ominaisuuksia on mahdollista käyttää Objective-C-koodin yhteydessä. Lisäksi se tarjoaa mahdollisuudet olio-ohjelmointiin sekä mahdollistaa dynaamisen ajoympäristön. [19]

Cocoa ja Cocoa Touch ovat sovelluskehitysympäristöjä Mac OS- ja iOS-käyttöjärjestelmille. Tämän työn käsittelemä sovellus on tehty iOS-käyttöjärjestelmille, joten käytössä on Cocoa Touch. Cocoa Touch koostuu Foundation- sekä UIKit-kehyksistä. Foundation-kehys sisältää paljon perustyökaluja sovelluskehitykseen iOS-alustalle. Se määrittelee esimerkiksi juuriluokan NSObject, josta periytyy valtaosa ympäristössä käytettävistä luokista. Foundation määrittelee myös muun muassa primitiivityypit (esimerkiksi merkkijonot ja luvut) sekä tietorakenteet (esimerkiksi taulukot ja sanakirjat). Se sisältää myös rajapinnat tyypillisesti käytettyihin järjestelmän kohteisiin ja palveluihin. [14]

Objective-C-sovelluksessa on mahdollista käyttää Swift-kirjastoja. [20] Tämä laajentaa käytettävissä olevien kolmansien osapuolten tekemien kirjastojen valikoimaa.

Vastaavasti se tarvittaessa mahdollistaa myös asiakassovelluksen kirjoittamisen osittain Swift:llä Objective-C:n lisäksi. Ominaisuus voi olla tarpeellinen esimerkiksi tilanteessa, jossa sovellus kirjoitettaisiin uudelleen Swift:llä osa kerrallaan.

Projektissa päädyttiin käyttämään Objective-C-kieltä asiakasohjelman kehitykseen. Syynä valinnalle oli, että sovelluskehityksen aloituksen aikoihin Objective-C:lle oli parempi kolmannen osapuolten tekemien kirjastojen valikoima kuin Swift:lle. Lisäksi Objective-C itsessään oli kyseisenä ajankohtana huomattavasti Swiftiä valmiimpi käyttöön. Swift on projektin sovelluskehityksen alkamisen jälkeen kokenut kaksi suurta ja vanhan koodin toiminnallisuuden rikkovaa muutosta. On mahdollista, että sovellus tulevaisuudessa kirjoitetaan uudelleen Swift:llä, sillä Apple sekä macOS-/iOS-yhteisö tukevat sitä nykyään paremmin kuin Objective-C:tä.

5.2.3 Python

Python on yleiskäyttöinen avoimen lähdekoodin ohjelmointikieli, joka on alun perin kehitetty seuraajaksi ABC-ohjelmointikielelle. [21; 22] Se on tehty erityisesti tukemaan ohjelmistojen laadukkuutta, sovelluskehittäjän tuottavuutta, sovellusten siirrettävyyttä sekä sovelluskomponenttien integrointia. [21]

Pythonista on olemassa useita eri toteutuksia, joista tunnetuin on CPython. CPythonia käytettäessä Python-koodi käännetään ajon aikana tavukoodiksi, jota ajetaan Python-tulkilla. Tyypillisesti koodin ajaminen Pythonilla on hitaampaa, kuin binäärikoodiksi kääntyvällä kielellä kirjoitettuna. [23]

Python tukee useampaa ohjelmointiparadigmaa. Sen luokkamalli tukee muun muassa perintää, moniperintää sekä operaattorien kuormitusta. Pythonia ei ole pakko ohjelmoida olio-ohjelmointikielenä, vaan sitä voi käyttää myös lausekieliseen ohjelmointiin. Lisäksi se sisältää monia funktionaalisten ohjelmointikielten ominaisuuksia, kuten generaattorit, koosteet, sulkeumat, kartoituksen, dekoraattorit, anonyymit lambda-funktiot sekä funktiot ensimmäisen luokan kansalaisina. Eri paradigmoja voi käyttää sellaisenaan tai niitä voi käyttää toistensa täydentämiseen. [23]

5.2.4 Django

Django on avoimen lähdekoodin ohjelmistokehys, jota käytetään verkkopalveluiden toteuttamiseen. Se on toteutettu Python-ohjelmointikielellä. [24]

Djangon kehityksessä on keskitytty erityisesti siihen, että yleisimmät verkkosivukehitykseen liittyvät tehtävät olisivat nopeita ja helppoja toteuttaa. Verkkosovelluksen rakenne koostuu yhdestä projektista ja useammasta sovelluksesta. Projekti

kattaa tyypillisesti kaikki yksittäisen projektin tai kirjaston kooditiedostot. Näiden lisäksi on mahdollista tuoda koodia muista projekteista, joka onkin tyypillinen tapa Django-kirjastoille. Django sisältää kattavan objekti-relaatio-kuvauksen, joka helpottaa relaatiotietokannan käyttämistä sovelluskehityksessä. Kuvauksen käyttämiseksi tulee määrittää niin sanottu malliolio, jonka pohjalta Django luo oliota vastaavan tietokantaskeeman luomiseen tarkoitetun SQL-koodin.

Listauksessa 5.1 on määritelty *Measurement*-tietokantamalli. Malli määritellään Python-luokkana, joka on peritty Djangon *Model*-luokasta.

Listaus 5.1. Django-tietokantamallin määrittely.

```
1 from django.db import models
2
3 class Measurement(models.Model):
4     timestamp = models.DateTimeField()
5     value = models.FloatField()
```

Tietokantamigraatiot ovat Python-koodia, jossa käytetään Djangon tarjoamia migraatiotyökaluja. Nämä työkalut tarjoavat yksinkertaisen rajapinnan SQL-tietokannalle tehtäviin yleisimpiin operaatioihin, kuten relaation lisäämiseen, muokkaamiseen tai poistamiseen. Tyypillisesti migraatiot luodaan tietokantamallin pohjalta. Tietokantamalli määritellään listauksen 5.1 mukaisella tavalla. Kun malli on ensimmäisen kerran määritelty, voidaan sen pohjalta luoda tietokantamigraatio, joka luo mallia vastaavan tietokantarelaation. Jos olemassa olevaa mallia on muokattu, voidaan mallin ja aikaisemmin luotujen migraatioiden pohjalta luoda uusi migraatio, joka muokkaa relaation vastaamaan mallia. Migraatioon voidaan lisäksi kirjoittaa Python-koodia, jos halutut muutokset vaativat sovelluslogiikkaa, jota Django ei voi luoda automaattisesti. Itse kirjoitettava Python-koodi voi käyttää kattavasti Djangon tarjoamia migraatio- ja mallirajapintoja.

Kun komentoriviltä ajetaan Djangon *makemigrations*-komento, Django luo määriteltyjen mallien pohjalta tarvittavat tietokantamigraatiot. Migraatiotiedostoja on mahdollista muokata käsin, mikäli tehdyt muutokset vaativat erityistä tietojen käsittelyä. Esimerkiksi tietotyyppien muuttaminen voi vaatia ylimääräistä työtä, jotta tieto saadaan siirrettyä uuteen muotoon halutulla tavalla.

Djangon mukana tulevalla komentorivityökalulla voidaan luoda migraatiot automaattisesti:

```
$ python manage.py makemigrations
```

Listauksessa 5.1 esitellyssä mallissa ei ole määritetty lainkaan pääavainta, jonka SQL-tietokannat kuitenkin vaativat. Django ratkaisee ongelman luomalla automaattisesti kokonaislukutyyppisen *id*-kentän.

Kun migraatioiden luomisen jälkeen ajetaan

```
$ python manage.py migrate
```

suorittaa Django tarvittavat toimenpiteet tietokannalle. Mikäli tietokannalle on aikaisemmin jo ajettu migraatioita, aloitetaan migraatioiden suorittaminen ensimmäisestä suorittamattomasta. Migraatioissa on myös mahdollista siirtyä taaksepäin määrittämällä haluttu sovellus sekä migraatioversio:

```
$ python manage.py migrate sovelluksennimi 0001_initial
```

Kun tarvittavat migraatiot on ajettu, voidaan tietokantaa käyttää objekti-relaatiokuvauksen tarjoaman rajapinnan läpi. Rajapintaa käyttöä esitellään listauksessa 5.2, jossa tietokannasta haetaan nykyisen päivän kaikkien mittapisteiden keskiarvo.

Listaus 5.2. Tietokantahaku Djangon QuerySet-oliolla.

```
1  from django.utils.timezone import datetime, now
2  from django.db.models import Avg
3
4  from .models import Measurement
5
6  Measurement.objects.create(timestamp=now(),
7                             value=1.0)
8
9  today = datetime.today()
10 measurements = Measurements.objects.all()
11 measurements = measurements.filter(timestamp__date=today)
12
13
14 today_avg = measurements.aggregate(Avg('value'))
15 # {'value__avg': 1.0}
```

Listauksessa 5.2 lisätään ensin tietokantaan yksi mittapiste nykyhetkelle ja sen jälkeen lasketaan tietokannassa tämän päivän mittapisteiden keskiarvo. Rivillä 6 luodaan tietokantaan uusi mittapiste. *Create*-metodin kutsuminen suorittaa tietokantakutsun välittömästi.

Rivillä 10 luodaan *QuerySet*, jonka tulosjoukkoon kuuluvat kaikki *Measurement*-tyyppiset monikot tietokannassa. *QuerySet*:n luominen ei vielä suorita tietokantakyselyä, joka mahdollistaa sen muodostamisen useammassa osassa. Luodulle *QuerySet*:lle kutsutaan *filter*-metodia, joka suodattaa tulosjoukkoa siten että siihen kuuluvat mittapistetaulusta monikot, joiden *timestamp*-kentän arvo sijoittuu nykyiselle päivälle. Toimenpide ei muuta alkuperäistä *QuerySet*:ia, vaan palauttaa paluuarvona uuden. Tässä esimerkissä kuitenkin ylikirjoitamme muuttujan *measurements* arvon uudella tulosjoukolla.

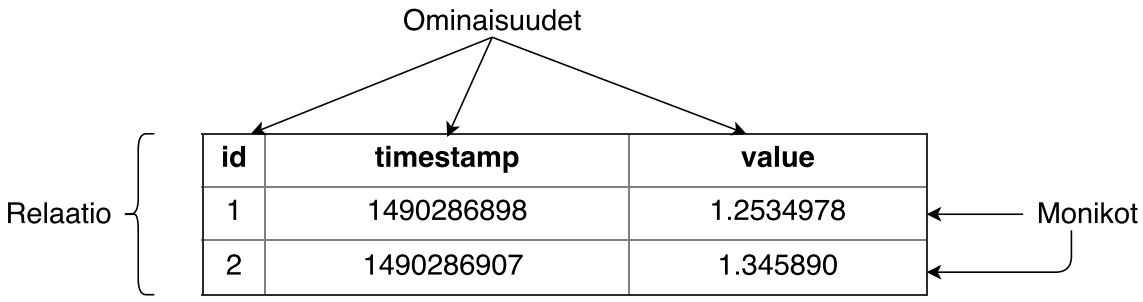
Rivillä 14 kutsutaan *QuerySet*:lle *aggregate*-metodia, jolle parametrina annetaan *Avg*-luokka, jonka parametrina on sarake, josta keskiarvo halutaan laskea. *Aggregate*-metodin kutsuminen aiheuttaa tietokantakyselyn suorittamisen tietokantapalvelimella, ja paluuarvona saadaan hakemisto, josta voidaan lukea pyydetty keskiarvo.

5.2.5 PostgreSQL

PostgreSQL on avoimeen lähdekoodiin perustuva objekti-relaatiotietokantajärjestelmä. Sitä on kehitetty aktiivisesti yli 15 vuotta. Tietokanta tukee vierasavaimia, liitoksia, herättimiä ja proseduureja. Se tukee ANSI-SQL:2008-standardia ja siinä voi käyttää suurinta osaa SQL:2008:n tietotyypeistä. PostgreSQL tukee lisäksi suurienkin binääritiedostojen sekä JSON-objektien tallennusta. [25]

Relaatiotietokanta rakentuu relaatiomallin päälle. Relaatiomallissa keskeiset käsitteet ovat määrittelyjoukot (domain), relaatiot (relation), ominaisuudet (attribute) sekä monikot (tuple). [26]

Relaatioita käytetään tiedon säilyttämiseen tietokannassa. Ne voidaan esittää kaksiulotteisena taulukkona kuvan 5.4 tavoin, jossa jokainen rivi vastaa yhtä monikkoa ja sarake tiettyä ominaisuutta. Näin ollen ominaisuuden voidaan sanoa olevan relaatiossa oleva nimetty sarake. Ominaisuuksien järjestyksellä relaatiossa ei ole merkitystä sen sisällön tai ominaisuuksien kannalta. Ominaisuuksien lukumäärä määrittää tietokannan asteisuuden. [26]



Kuva 5.4. *Esimerkki relaatiosta, jonka monikot kuvaavat mitta-arvoja eri ajanhetkillä.*

Määrittelyjoukot ovat käytännössä tietokannan tietotyyppejä. Määräämätön määrä relaation tietueita voidaan määritellä samalle määrittelyjoukolle. On huomattavaa, että tyypillisesti määrittelyjoukossa on arvoja, jotka eivät esiinny tietokannan ominaisuuksissa. Määrittelyjoukkojen keskeinen tehtävä pitää tietokannassa oleva tieto halutunlaisena. Niillä voidaan esimerkiksi erottaa merkkijono- ja kokonaislukutyyppiset kentät toisistaan. [26]

Monikot ovat relaation rivejä. Monikoiden järjestyksellä relaatiossa ei ole merkitystä. Relaation kardinaalisuus on siinä olevien monikoiden lukumäärä. Tyypillisesti relaation muut ominaisuudet kuin monikot eivät muutu ajan käytön myötä. Monikoita voidaan näin ollen kutsua relaation tilaksi. [26]

PostgreSQL täyttää ACID-ominaisuudet täydellisesti. [25] ACID-ominaisuudet määritellään seuraavasti: [26]

- Atomicity, atomisuus: transaktion vaikutukset tulevat yhtäaikaisesti voimaan, tai mikään niistä ei tule voimaan.
- Consistency, eheys: tietokannan täytyy siirtyä eheästä tilasta eheään tilaan. Ehdeydellä tarkoitetaan sitä, että tietokannan tila on sen rajoitteiden mukainen.
- Isolation, erillisyys: transaktiot tapahtuvat toisistaan riippumatta. Käytännössä siis kesken oleva transaktio ei vaikuta muihin meneillään oleviin transaktioihin.
- Durability, pysyvyys: sitoutettu transaktio on pysyvästi kirjattu tietokantaan, eikä sitä seuraava virhetilanne saa hävittää sitoutetun transaktion kirjoittamaa tietoa.

5.2.6 Realm-tietokanta

Asiakassovelluksen pysyväistietokannaksi valittiin Realm Mobile Database. Pysyväistietokanta mahdollistaa tiedon säilyttämisen sovelluksen uudelleenkäynnistysten välillä. Lisäksi hetkellisesti tarpeetonta tietoa voidaan säilyttää massamuistilla ja näin ollen säästää keskusmuistia sekä verkkoliikennettä.

Sovelluksessa käsitellään lukumäärällisesti suuria määriä objekteja, joita luetaan suuria määriä tietokannasta kerrallaan. Näin ollen tietokannan lukusuorituskyky koettiin tärkeäksi kriteeriksi tietokantaa valittaessa. Realmin omien mittausten mukaan lukusuorituskyky on paras ja kirjoitusnopeus koettiin riittäväksi [27]. Suorituskykynäkökulman kannalta myös säieturvallisuus koettiin tärkeäksi, sillä sujuvan käyttökokemuksen takaamiseksi suuria tietomääriä joudutaan käsittelemään taustasäikeessä. Lisäksi tietokannan rakenteen ja kyselyt voi tehdä suoraan samalla ohjelmointikielellä, kuin millä itse sovellus on kirjoitettu. Esimerkiksi SQLite-tietokannassa rakenne ja kysely jouduttaisiin määrittelemään SQL-kielellä, tai ottamalla mukaan jokin ominaisuudet tarjoava kirjasto.

Realm-tietokantaa voidaan käsitellä kolmen eri tyyppisen luokan kautta: olio (Object), taulukko (Array) ja Realm [28]. Oliot vastaavat pääpiirteissään relaatiotietokannan monikoita. Taulukoita voidaan käyttää muodostamaan moneen-suhteita tietokannassa. Realm-luokkaa vastaava käsite relaatiotietokantojen yhteydessä on tietokanta.

Realm-tietokannassa oliot kuvataan määrittelemällä luokka, joka vastaa relaatiotietokannan relaation määrittelyä. Taulukoita käytetään olioiden välisten moneen-suhteiden määrittelyyn. Vastaava toiminnallisuus voidaan toteuttaa vierasavaimilla relaatiotietokannassa. Niillä voidaan siis määrittää yhdelle oliolle suhde ennalta määräämättömään määrään muita olioita. Realm tarkoittaa luokkana tietovarastoa, joka voi sisältää mielivaltaisen määrän olioita. Se vastaa siis tietokantaa relaatiotietokannoissa.

Näiden lisäksi saatavilla on *Migrations*-apuluokka, jonka avulla tietokannan rakennetta voidaan muuttaa hävittämättä olemassa olevaa tietoa. [27]

Realm-oliolla voi olla yksi-yhteen-, yksi-moneen- sekä monta-moneen-suhteita muihin tietokantaoliohin. [27]

Listaus 5.3. Realm-luokkien määrittely.

```
1  @interface Image : RLMObject
2
3  @property NSData *file;
4
5  @end
6
7
8  @interface Sensor : RLMObject
9
10 @property NSString *name;
11 @property Image *image;
12
13 @end
14
15
16 @interface User : RLMObject
17
18 @property NSString *name;
19 @property RLMArray<Sensor *><Sensor> *sensors;
20
21 @end
```

Listauksessa 5.3 on määritelty kolme Realm-luokkaa: *Image*, *Sensor* ja *User*. Esimerkissä on yksi-moneen-suhde *User*- ja *Sensor*-luokkien välillä ja näin ollen yhdellä käyttäjällä voi olla monta sensoria. *User*-luokan puolelta katsottuna tämä on esimerkki moneen-suhteesta. *Sensor*-luokalla on *image*-ominaisuus, jonka vuoksi *Sensor*- ja *Image*-luokkien välillä on yksi-yhteen- tai yksi-moneen-suhde. *Sensor*-luokan puolelta katsottuna tämä on esimerkki yhteen-suhteesta.

Realm on suunniteltu helppokäyttöiseksi. Realmin transaktiot täyttävät ACID-ominaisuudet täydellisesti. [27]

5.3 Tietomalli

Järjestelmässä käytetään objekteja tietosisällön säilyttämiseen. Jokainen objekti esittää yksittäistä ja itsenäistä asiaa. Asiakassovelluksen ja taustajärjestelmän tietomallit poikkeavat toisistaan hieman.

5.3.1 Objektit asiakassovelluksessa

Synkronoitaville objekteille tarvitaan joitain synkronointimenetelmän kannalta välttämättömiä attribuutteja. Tämän vuoksi kaikki synkronoitavat objektit periytyvät *SyncObject*-kantaluokasta, joka on esitelty listauksessa 5.4. Koska objektit tallennetaan Realm-tietokantaan, asetetaan niiden kantaluokaksi *RLMObject*.

Listaus 5.4. Synkronoitavien mallien kantaluokka.

```
1 @interface SyncObject : RLMObject
2
3 @property NSString *UUID;
4 @property int64_t revision;
5 @property PSISyncState syncState;
6 @property BOOL deleted;
7
8 @end
```

UUID-attribuutti on objektin yksilöivä UUID4-muotoinen tunniste. Samaa tunnistetta käytetään sekä palvelimella, että asiakassovelluksessa yksilöimään objekti. Objektin luonut taho luo myös tunnisteen sille. *Revision*-attribuutti määrittää objektin revision palvelimella. Mikäli objektia ei ole vielä synkronoitu palvelimelle kertaakaan, on attribuutin arvo -1.

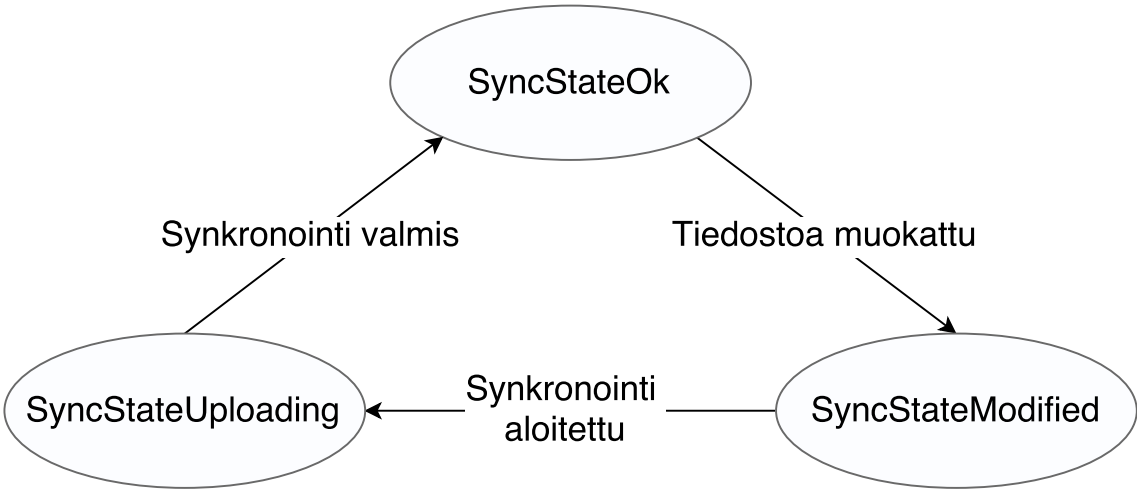
syncState-attribuuttia käytetään esittämään objektin tilaa synkronoinnin suhteen asiakassovelluksessa. *PSISyncState* on määritelty listauksen 5.5 mukaisesti enumeraationa.

Listaus 5.5. Synkronoitavien objektien mahdolliset tilat.

```
1 typedef NS_ENUM(int, SyncState) {
2     SyncStateOk = 0,
3     SyncStateModified = 1,
4     SyncStateUploading = 2,
5 };
```

Enumeraation arvo *SyncStateOk* tarkoittaa, että objekti on synkronoinnin kannalta niin sanotussa perustilassa. Tällöin siihen ei ole tehty muutoksia asiakassovelluksessa, ja näin ollen sitä ei tarvitse lähettää palvelimelle. Objekti on *SyncStateModified*-tilassa silloin, kun sitä on muokattu asiakassovelluksessa, mutta ei ole vielä lähetetty palvelimelle. *SyncStateUploading*-tilassa olevan objektia oli muokattu ennen juuri meneillään olevaa synkronointia ja sitä ollaan synkronoimassa palvelimelle

parhaillaan. Mahdolliset tilasiirtymät *syncState*:n arvojen välillä on esitetty kuvassa 5.5.



Kuva 5.5. *Objektin syncState-attribuutin arvon mahdolliset tilasiirtymät.*

5.3.2 Objektit taustajärjestelmässä

Objektit on mallinnettu taustajärjestelmässä Django-sovelluskehyksen mallien avulla. Jokainen synkronoitava objekti periytyy kantaluokasta, joka sisältää objektien yhteiset ominaisuudet. Kantaluokka on esitelty listauksessa 5.6.

Listaus 5.6. Synkronoitavien objektien kantaluokka.

```
1  class BaseSyncModel(models.Model):
2      uuid = models.UUIDField(primary_key=True,
3                              default=uuid4,
4                              editable=False)
5      created = models.DateTimeField(default=now)
6      modified = models.DateTimeField(default=now)
7      deleted = models.BooleanField(default=False)
8      revision = models.BigIntegerField(default=0)
9
10     ...
11
12     class Meta:
13         abstract = True
```

Rivillä 1 määritellään luokan nimi ja sen kantaluokka. *Model*-luokka on Djangon tarjoama kantaluokka, jota käytetään kaikkien tietokantamallien kantaluokkana. Mallin

attribuutit määritellään luokan luokkamuuttujina. Mallille määritellyt attribuutit ovat siis *uuid*, *created*, *modified*, *deleted* ja *revision*. *Uuid*-attribuutti on tyypiltään *UUIDField* ja se on asetettu mallin pääavaimeksi. Oletusarvioisesti arvo luodaan *uuid4*-funktiolla, joka generoi uuden satunnaisen uuid4-arvon. *Created*-attribuutti kertoo objektin luontiajan ja *modified* milloin objektia on viimeksi päivitetty. *Deleted*-attribuutilla ilmaistaan, mikäli objekti on poistettu. *Revision*-attribuutti kertoo objektin kulloisenkin revisionumeron.

Luokan sisäisessä *Meta*-luokassa määritellään luokka abstraktiksi. Tämä tarkoittaa sitä, että Django ei luo mallille omaa taulua, vaan sisällyttää mallin kentät siitä periviin malleihin. Tietokantatasolla tämä tarkoittaa sitä, että abstraktia mallia vastaavaa taulua ei luoda tietokantaan. Tästä johtuen abstraktista mallista ei voi luoda objekteja.

Objekteilla on riippuvuuksia toisiinsa, jotka ilmenevät taustajärjestelmän puolella vierasavaimina. Käytössä oleva relaatiotietokanta täyttää ACID-ominaisuudet, joka takaa sen eheyden. Jotta tietokannan asettamat eheysvaatimukset voidaan täyttää, on vierasavainviittausten oltava asetettu oikein.

Listauksessa 5.7 on esitelty tilanne, jossa malli B:ssä on vierasavainviittaus malliin A. Rivillä 10 luodaan mallin B pohjalta objekti b, jonka *parent*-kentän arvoksi asetetaan objektin a tunniste. Listauksessa rivillä 12 pyritään tallentamaan objekti b ennen, kuin objektia a, josta b riippuu, on tallennettu. Tästä aiheutuu virhetilanne tietokantatasolla mistä johtuen järjestelmän toiminnalta edellytetään sitä, että objektit tallennetaan tietokantaan tietyssä järjestyksessä: ensin sellaiset, jotka eivät viittaa muihin ja tämän jälkeen jo lisättyihin objekteihin viittaavat objektit ja niin edelleen. Muutokset objekteihin tehdään aina transaktiona, jolloin voidaan varmistua siitä, että kaikki tehdyt muutokset tulevat voimaan kerralla. Vastaavasti mikään muutoksista ei tule voimaan, jos yhdenkin muutoksen tekeminen epäonnistuu.

Listaus 5.7. Vierasavainviittauksen väärinkäyttötilanne.

```
1  class A(BaseSyncModel):
2      ...
3
4
5  class B(BaseSyncModel):
6      parent = models.ForeignKey(A)
7      ...
8
9  a = A()
10 b = B(parent_id=a.uuid)
11
12 b.save()  # Virhe
13 a.save()
```

Tyypillisesti kun objekti poistetaan asiakassovelluksesta, objektia ei voida poistaa taustajärjestelmän tietokannasta. Tämä johtuu siitä, että tieto objektin poistumisesta on voitava välittää muille asiakasohjelmille synkronoinnin yhteydessä. Varsinaisen poistamisen sijasta objektilla on totuusarvotyyppinen *deleted*-ominaisuus, joka ilmaisee onko objekti poistettu tietokannasta. Kun objekti on merkitty poistetuksi, voidaan tieto poistosta välittää asiakassovelluksille. Kyseisen objektin tila voidaan vastaavasti jättää kokonaan lähettämättä asiakassovelluksille, jotka eivät ole missään vaiheessa säilyttäneet kyseistä objektia.

Osa järjestelmän objekteista ovat sellaisia, että ne voidaan poistaa tietokannasta pysyvästi. Tällaisia objekteja ovat tyypillisesti sellaiset, joita ei synkronoida laitteiden välillä. Esimerkiksi mittapisteet ja kommentit ovat sellaisia objekteja, joita asiakasohjelma hakee taustajärjestelmältä vain tarvittaessa. Ne liittyvät johonkin synkronoitavaan objektiin vahvasti eikä niitä näytetä asiakassovelluksessa muuten kuin tämän objektin yhteydessä. Tästä johtuen niitä ei pääse tarkastelemaan asiakassovelluksessa mikäli objekti, johon ne liittyvät on poistettu. Esimerkiksi käyttäjien kirjoittamat kommentit liittyvät aina mittaustapahtumaan eikä niitä siirretä synkronoinnin avulla laitteiden välillä. Mikäli mittaustapahtuma merkitään synkronoinnin aikana poistetuksi, voidaan siihen liittyvät kommentit poistaa tietokannasta.

5.3.3 Versiointi

Järjestelmässä on käytössä kello jokaista käyttäjää kohden, jota käytetään muutosten järjestyksen hallintaan. Käytännössä siis kaikki käyttäjän käyttämät päätelait-

teet jakavat saman kellon.

Järjestelmässä kaikki synkronoitavat objektit sisältävät versionumeron. Versionumero päivittyy objektille sen luonti- sekä muokkaustilanteissa. Kun objekti luodaan tai sitä muokataan, asetetaan sille objektin omistavan käyttäjän tuolla hetkellä oleva versionumerolaskurin arvo. Versionumeron asetus tehdään aina palvelimella, eli kyseessä on keskitetty algoritmi.

Käyttäjän versionumero on kasvava lukuarvo. Aina, kun arvoa ollaan aikeissa käyttää objektin versionumeron päivittämiseen, korotetaan sitä yhdellä. Näin ollen viimeiseksi päivitetyillä objekteilla on aina suurin versionumero. Lisäksi tarkastelemalla objektien versionumeroita, voidaan päätellä missä järjestyksessä niihin kohdistuneet muutokset on suoritettu taustajärjestelmässä.

Versionumeron kasvavasta luonteesta johtuen voidaan olettaa, että mikäli palvelimella oleva käyttäjäkohtainen versionumero on sama kuin asiakassovelluksella oleva suurin versionumero, ei käyttäjän objekteihin ole tehty muutoksia asiakassovelluksen ulkopuolella. Vastaavasti mikäli palvelimella oleva käyttäjäkohtainen versionumero on suurempi kuin suurin asiakassovelluksen näkemä versionumero, voidaan olettaa, että taustajärjestelmässä on muutoksia, joita asiakassovelluksella ei ole.

5.4 Poissulkeminen SQL-tietokannassa

Poissulkeminen on suoritettu järjestelmässä keskitetysti taustajärjestelmässä. Koska varsinaisia sovelluspalvelimia on useita, säilytetään poissulkemisen kannalta oleellisia tilatietoja tietokantapalvelimella.

Kun käyttäjän tietoja ollaan päivittämässä, lukitaan tietokannasta käyttäjän version sisältävä rivi SQL:n *SELECT FOR UPDATE* -lauseella. *SELECT FOR UPDATE* -lause lukitsee kyselyssä määritetyt rivit [29]. Lukittuja rivejä ei voi lukita, muokata tai poistaa muista transaktioista ennen kuin lukitseva transaktio päättyy [29]. Näin saadaan luotua kriittinen alue, joka estää käyttäjän eri laitteilta yhtäaikaisesti tapahtuvat synkronoinnit. Listauksessa 5.8 on esitelty SQL-lause, jolla lukitaan käyttäjän 1 rivi *sync_revision*-relaatiossa.

Listaus 5.8. Tietokantarivin lukitseminen SQL:lla.

```
1 BEGIN;
2 SELECT * FROM sync_revision WHERE user_id=1 FOR UPDATE;
3
4 ...
5
6 COMMIT;
```

Rivillä 1 aloitetaan uusi transaktio. Rivillä 2 pyritään lukitsemaan sync_revision-relaation rivi, jossa *user_id*-ominaisuuden arvo on 1. Mikäli rivi on jo lukittu, jäädään odottamaan rivin vapautumista. Listauksen rivien 3–5 kohdalla voidaan nyt suorittaa halutut SQL-kyselyt. Tämä toimii nyt kriittisenä alueena. Rivillä 6 oleva *COMMIT*-lause päättää aloitetun transaktion ja vapauttaa lukon.

5.5 Algoritmin toteutus

Tässä työssä käytetty synkronointimenetelmä perustuu loogisen kellon ajatusmalliin siitä, että aikaa käsitellään kokonaislukuna, joka kuvaa tapahtumien järjestystä. Käytetty menetelmä on keskitetty: kellojen arvoa kasvatetaan ainoastaan taustajärjestelmässä.

Asiakassovellus pyrkii siihen, että sen tietosisältö olisi mahdollisimman ajan tasalla suhteessa palvelimella olevaan tietosisältöön. Sovellus synkronoi tietosisältönsä aika ajoin taustajärjestelmän kanssa.

Listauksessa 5.9 on esitetty sovelluskoodi, jolla asiakassovellus hakee Realm-tietokannasta lähetettävät objektit. Rivillä 4 esiteltävä predikaatti hakee tietokannasta kaikki tietyntyyppiset objektit, joiden *syncState*-attribuutin arvo on erisuuri kuin *SyncStateOk*. Predikaatti siis valitsee myös objektit, joiden tilaksi on asetettu *SyncStateUploading*. Nämä objektit valitaan, koska mikäli sovellus on kaatunut tai muuten sulkeutunut edellisen synkronointikerran aikana, lähetetään aikaisemmin mahdollisesti epäonnistuneesti lähetetyt objektit uudelleen taustajärjestelmälle.

Listaus 5.9. Muokattujen objektien hakeminen tietokannasta.

```
1  - (NSDictionary *)modifiedObjects
2  {
3    NSMutableDictionary *data;
4    NSPredicate *modifiedPredicate = [NSPredicate
5      predicateWithFormat:@"syncState != %d", SyncStateOk]
6    data["modelTypeA"] = [ModelClassA
7                          objectsWithPredicate:modifiedPredicate];
8    ...
9    data["modelTypeN"] = [ModelClassN
10                         objectsWithPredicate:modifiedPredicate];
11   return [NSDictionary dictionaryWithDictionary:data];
12 }
```

Listauksessa 5.10 on sovelluskoodi, jolla asiakassovellus lähettää synkronoinnin aloittavan pyynnön palvelimelle. Rivillä 4 muodostetaan verkko-osoite taustajärjestelmään. Osoitteeseen liitetään parametrina edellisen synkronointikerran yhteydessä muodostunut revisionumero. Rivillä 7 kutsutaan listauksssa 5.9 esiteltyä funktiota, joka palauttaa lähetettävät objektit. Nämä objektit merkitään *SyncStateUploading*-tilaan rivillä 10 olevalla funktiokutsulla. Rivillä 13 lähetetään lähetettäväksi merkityt objektit taustajärjestelmälle asynkronisesti.

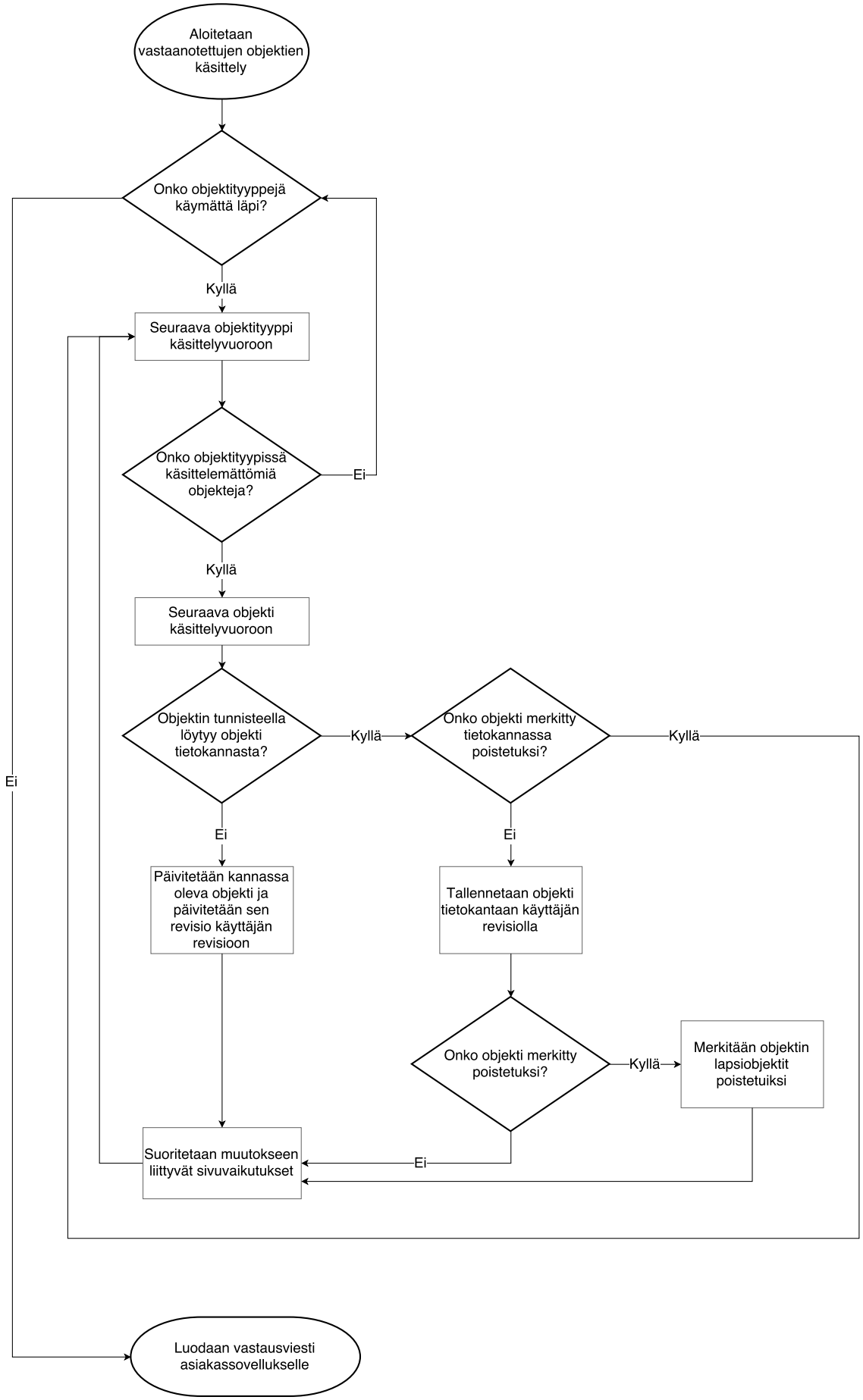
Listaus 5.10. Tiedon synkronoinnin runko asiakassovelluksella.

```
- (void)syncData:(void(^)(BOOL))finished
{
  // Construct URL with last sync revision as a parameter
  NSString *urlForSync = [_syncUrl withLastSyncRevision:
                                    _lastSyncRevision];

  NSDictionary *syncDataDict = [self modifiedObjects];

  // Mark objects as uploading
  [syncData markAsUploading];

  __weak typeof(self) weakSelf = self;
  [_networkManager sendDataToUrl:urlForSync
    parameters:syncDataDict
    success:^(id response, NSInteger statusCode) {
    [weakSelf handleSuccessfulSyncResponse:response];
    [syncData markAsUploadComplete];
    finished(YES);
  } failure:^(NSError *error) {
    // Mark objects for failed upload
    [syncData markAsUploadFailed];
    finished(NO);
  }];
}
```

Saatuaan synkronointipyynnön, taustajärjestelmä lukitsee käyttäjän revisiorivin tietokannasta. Rivin lukitsemisella voidaan muodostaa taustajärjestelmän koodissa kriittinen alue. Kun rivi on lukossa ja sitä yritetään lukita uudelleen, odotutetaan toista lukitsijaa, kunnes ensimmäinen lukitsija on vapauttanut lukkonsa. Koska lukitaan vain käyttäjäkohtainen tietokantarivi, saadaan aikaan käyttäjäkohtainen kriittinen alue. Käytännössä tästä seuraa, että vain yksi asiakassovellus kerrallaan käyttäjätunnusta kohden voi suorittaa synkronointia, eli toimia kyseisen käyttäjän kriittisellä alueella.

Seuraavaksi taustajärjestelmä korottaa käyttäjän revisionumeroa yhdellä. Kriittinen alue pitää huolen siitä, että revisionumero ei voi nousta ennen kuin vuorossa oleva asiakasohjelma on suorittanut synkronoinnin loppuun.

Taustajärjestelmä käy asiakasohjelmalta vastaanotetut objektit läpi objektityyppi kerrallaan. Nämä objektit ovat sellaisia, joita käyttäjä on muokannut tai luonut asiakassovelluksessa edellisen synkronointikerran jälkeen. Kuvassa 5.6 on esitetty vuokaaviona vaiheet, joiden mukaisesti palvelin käsittelee vastaanottamansa objektit synkronoinnin aikana. Objektien läpikäyminen on esitetty kooditasolla listauksessa 5.11.



Kuva 5.6. *Synkronoinnin vaihe, jossa palvelin käsittelee asiakassovellukselta saamansa objektit.*

Listaus 5.11. Asiakassovellukselta saatujen objektien käsittely taustajärjestelmässä.

```
def save_newer_objects_to_database(request_data,
                                   new_sync_revision):

  # A set containing objects which are updated as result of
  # modifying other object during the sync
  updated_objects = set()

  for (field, model, serializer_class) in SYNC_FIELDS:
    received_objects = request_data.get(field, [])

    for r_obj in received_objects:
      existing = model.filter(pk=r_obj.get('uuid'))
                      .first()

      # An object with the same uuid already exists
      if existing is not None:
        serializer = serializer_class(existing, data=r_obj)

        # Deleted object will always override others
        if not existing.deleted:
          obj = serializer.save(revision=new_sync_revision)

          if obj.deleted:
            try:
              updated_objects.remove(obj)
            except KeyError:
              del_children = obj.mark_children_deleted()
              updated_objects.update(del_children)

      # Create as a new object
      else:
        serializer = serializer_class(data=r_obj)
        obj = serializer.save(revision=new_sync_revision)

        # If object's parent object is deleted, mark the
        # just created object as deleted as well as it's
        # child objects.
```

```
38          if obj.parent.deleted:
39            obj.deleted = True
40            obj.save()
41            del_children = obj.mark_children_deleted()
42            updated_objects.update([obj] + del_children)
43    return updated_objects
```

Jokaisella objektilla on yksilöllinen tunniste. Taustajärjestelmä voi tarkastaa tietokannasta, löytyykö sieltä jo objekti, jolla on sama tunniste. Listauksessa 5.11 tämä on esitetty rivi 12, jossa tietokannasta haetaan objektia, jonka pääavain vastaa vastaanotetun objektin pääavainta. Mikäli objekti löytyy (listauksen rivi 16), kyseessä on päivitys kyseiseen objektiin. Jos taas tunnisteella ei löydy objektia (listauksen rivi 32), luodaan uusi objekti vastaanotetun objektin pohjalta. Luodun objektin revisio merkitään käyttäjän revisionumeroksi.

Mikäli tietokannassa oleva objekti on merkitty poistetuksi, ei sille tehdä mitään (listauksen rivi 20). Muussa tapauksessa päivitetään tietokannassa oleva objekti asiakassovellukselta saadulla objektilla, ja asetetaan sen revisionumeroksi käyttäjän nykyinen revisionumero. Mikäli asiakassovellus oli merkinnyt objektin poistetuksi, merkitään myös kyseisen objektin lapsiobjektit poistetuiksi listauksen riveillä 24–28 osoitetulla tavalla. Poistetuiksi merkittyjen lapsiobjektien revisionumerot päivitetään myös käyttäjän revisionumeroon, jotta muut käyttäjän laitteet saavat tiedon objektien poistosta. Tässä yhteydessä päivitettyjen objektien tunnisteet otetaan talteen, jotta niiden poistosta voidaan ilmoittaa asiakassovellukselle.

Joidenkin objektityyppien muutoksilla ja luomisilla on sivuvaikutuksia. Sivuvaikutukset toteutetaan, kun muutos tai luominen on tehty. Sivuvaikutuksia voivat olla esimerkiksi toisen, ei synkronoitavan objektin luominen tai muiden objektien merkitseminen poistetuiksi. Mikäli sivuvaikutuksen seurauksena käsitellään synkronoitaviksi luokiteltavia objektityyppejä, otetaan talteen muuttuneet tai poistetut objektit, jotta ne voidaan lähettää asiakassovellukselle myöhemmin synkronoinnin aikana.

Taustajärjestelmä koostaa vastauksen asiakassovellukselle listauksen 5.12 mukaisesti. Taustajärjestelmän asiakassovellukselle vastauksena lähettämät objektit valitaan sen mukaan, onko kyseessä asiakassovelluksen ensimmäinen synkronointikerta vai olemassa olevan tietosisällön päälle tehtävä päivitys. Mikäli kyseessä on ensimmäinen kerta, muuttujan *last_sync_revision* arvo on *None*. Mikäli kyseessä on asiakassovelluksen ensimmäinen synkronointikerta, haetaan tietokannasta kaikki käyttäjän objektit, joita ei ole merkitty poistetuiksi lisäämällä tietokantahakuun suodatin ri-

vin 10 mukaisesti. Jos kyseessä on päivitys, haetaan tietokannasta objektit, joiden revisio on suurempi kuin edellisen synkronointikerran seurauksena muodostunut revisio, mutta pienempi kuin meneillään olevan synkronoinnin seurauksen muodostunut revisio. Tällöin siis *qs* (tyypiltään *QuerySet*) muodostuu rivien 5 ja 8 tuloksena. Tällöin ei siis oteta kantaa hakutuloksena saatavien objektien *deleted*-attribuutin tilaan, jolloin saadaan myös poistetut objektit.

Listaus 5.12. Asiakassovellukselle lähetettävien objektien hakeminen taustajärjestelmässä.

```
1  def response_objects(last_sync_revision, new_sync_revision):
2    response = {}
3
4    for (field, model, serializer) in SYNC_FIELDS:
5      qs = model.objects
6                .filter(revision__lt=new_sync_revision)
7      if last_sync_revision is not None:
8        qs = qs.filter(revision__gt=last_sync_revision)
9      else:
10       qs = qs.filter(deleted=False)
11     response[field] = serializer(qs, many=True).data
12
13   return response
```

Asiakassovellukselle palautetaan meneillään olevan synkronointikerran seurauksena muodostunut revisionumero, lista synkronoinnin seurauksena poistetuiksi merkittyjen objektien tunnisteita sekä lista päivittyneistä objekteista. Lopuksi vapautetaan käyttäjän revision rivi tietokannasta, jotta käyttäjän muut asiakassovellukset pääsevät suorittamaan synkronointia.

Kun asiakassovellus on vastaanottanut taustajärjestelmältä saamansa listat, muokkaa se tietosisältöään niiden pohjalta: uudet objektit lisätään tietokantaan, muokatut objektit päivitetään tietokantaan sekä poistetuiksi merkityt objektit poistetaan tietokannasta. Käytännössä tämä tapahtuu listauksen 5.13 mukaisesti. Listauksessa esiteltyä funktiota kutsutaan listauksessa 5.10 verkkopyyntöfunktiokutsun parametrina annetussa lohkossa rivillä 16. Vastaus käydään läpi tietotyypeittäin.

Listaus 5.13. Palvelimelta vastaanotettujen objektien käsittely asiakassovelluksella.

```
1  - (void)handleSuccessfulSyncResponse:
2    (NSDictionary *)response
3  {
4    RLMRealm *realm = [RLMRealm defaultRealm];
5    [realm beginWriteTransaction];
6
7    // User data
8    [response.modelTypeA
9      makeObjectsPerformSelector:@selector(toRealmObject)];
10   ...
11   [response.modelTypeN
12     makeObjectsPerformSelector:@selector(toRealmObject)];
13
14   _lastSyncRevision = response.revision;
15
16   [realm commitWriteTransaction];
17 }
```

Kun vastauksena saadut tietueet muuttaa Realm-objekteiksi listauksen rivien 8–12 mukaisesti, ne ovat valmiita tallennettavaksi tietokantaan. Käytetty *makeObjectsPerformSelector*-funktio kutsuu samaa funktiota *toRealmObject* antaen vuoronperää sille parametrina jokaisen taulukon alkion. Kun muutos on tehty *beginWriteTransaction*- ja *commitWriteTransaction*-funktiokutsujen välissä, tallentuu kaikki muutokset tietokantaan yhdessä transaktiossa. Asiakassovellus tallentaa synkronoinnin seurauksena muodostuneen revision numeron, jotta se voi lähettää sen taustajärjestelmälle seuraavan synkronointipyynnön yhteydessä.

Kun vastaanotetut tiedot on tallennettu, asiakassovellus kirjaa lähettämiensä objektien *syncStatus*-attribuutiksi *SyncStatusComplete* merkitsemään, että nämä objektit ovat edellisen synkronointikerran valmistumishetkellä olleet ajan tasalla taustajärjestelmän tietokannan kanssa. Mikäli asiakassovellus havaitsee verkkopyynnön epäonnistuneen, se merkitsee objektien tilaksi *SyncStatusModified*, jotta objektit lähetettäisiin uudelleen seuraavan synkronointikerran aikana.

6. ARVIOINTI

Sovellus on yhä aktiivisessa kehityksessä, eikä sitä ole julkaistu vielä yleisesti saataville. Sovelluksen kehittäjät ja muutama testikäyttäjä ovat käyttäneet sovellusta.

Tässä työssä esitelty synkronointimenetelmä on osoittautunut toimivaksi ratkaisuksi ongelmaan, johon se kehitettiin. Synkronointialgoritmi mahdollistaa onnistuneesti sekä verkottomassa tilassa sovelluksen käyttämisen että saman käyttäjän usean laitteen yhtäaikaisen käytön. Menetelmässä ei ole havaittu ongelmia tiedon konsistenttiudesta. Käytännössä ei ole tullut tilannetta, jossa tietosisältö ei synkronoituisi laitteiden välillä tai tilannetta, jossa tuotettua sisältöä katoaisi odottamattomasti.

Suorituskyky on ollut riittävä testauskäytön käyttäjämäärällä. Käyttäjämäärät ovat kuitenkin olleet hyvin rajalliset, joten suorituskyky järjestelmän todellisessa käyttötapauksessa ei ole vielä selvillä. Järjestelmää ei ole toistaiseksi kuormitustestattu keinotekoisesti.

On huomattava, että menetelmän nopeuden tai skaalautuvuuden suhteen ei ole vielä tehty erillistä optimointityötä. Tästä johtuen on hyvin todennäköistä, että suorituskykyä ja skaalautuvuutta voidaan parantaa merkittävästi nykytilanteesta.

Menetelmän skaalautumista rajoittavimman tekijän oletetaan olevan taustajärjestelmän SQL-tietokanta. Tietokantaan joudutaan tekemään lukuisia hakuja jokaisen synkronointikerran aikana. Lisäksi tietokantaan kirjoituksia voidaan joutua tekemään paljon, mikäli asiakassovelluksella on paljon muokattuja tai lisättyjä tietueita.

6.1 Vertailu Kedian ja Prakashin menetelmään

Tässä työssä kuvatulla menetelmällä on joitain eroavaisuuksia verrattuna Ashish Kedian ja Anusha Prakashin julkaisemaan menetelmään. Molempien menetelmien käyttötarkoitus on kuitenkin pääpiirteissään sama, joten niitä voidaan käyttää ratkaisemaan samoja vaatimuksia.

Keskeinen ero menetelmien välillä on synkronoinnin laitekohtaisen tilan hallinta.

Tässä työssä kuvattu menetelmä käyttää revisioarvoja, kun taas vertailualgoritmissa pidetään objekti- ja asiakaslaitekohtaista kirjanpitoa synkronoinnin tilasta. Tämä mahdollistaa vertailualgoritmissa sen, että tietoa voidaan synkronoida helposti osissa. Tässä työssä kuvatussa menetelmässä on synkronoinnissa pakko synkronoida aina kaikki tietosisältö kerralla, joka tarkoittaa nykytoteutuksessa rajaamattoman suurta verkkopyyntöä. Etuna työssä esitellyssä tavassa on, että se vaatii vähemmän kirjanpitoa ja vie vähemmän tilaa taustajärjestelmässä: työssä esitellyssä ratkaisussa jokaiselle objektille lisätään taustajärjestelmässä vain revisionumerotieto, kun vertailualgoritmissa puolestaan taustajärjestelmä säilyttää neljästä attribuutista koostuvaa synkronoinnin tilannetietoa jokaisen objektin osalta jokaista asiakassovellusta kohden. Asiakassovelluksen puolella säilytettävän kirjanpitotiedon määrissä ei ole oleellista eroa.

Suorituskykymittauksia ei ole tehty kummallekaan menetelmälle, joten sen suhteen eroja ei voida arvioida.

6.2 Kehitysmahdollisuudet

Tietokannan käyttöä voitaisiin optimoida siten, että asiakassovellus ei lähettäisi säännöllisiä synkronointipyyntöjä taustajärjestelmälle, vaan pyyntöjä tehtäisiin vain, kun niille nähdään oikea tarve. Asiakassovellus voisi esimerkiksi odottaa, kunnes muutoksia ei ole määriteltyyn aikaan tullut lisää, jonka jälkeen muutokset lähetetään palvelimelle. Vastaavasti taustajärjestelmässä tapahtuneista muutoksista voitaisiin tiedottaa esimerkiksi push-viestien avulla.

Menetelmä vaatii nyt, että asiakassovelluksen tietosisältö on päivitettävä ajan tasalle yhdellä verkkopyyntö-vastaus-parilla. Tämä voi koitua ongelmaksi, jos käyttäjäkohtaisen tiedon määrä kasvaa paljon, jolloin verkkopyynnön tai vastauksen koko kasvaa suureksi. Suuri verkkopyyntö voi koitua ongelmaksi hyvinkin nopeasti, mikäli sovellusta käytetään tilanteessa, jossa verkkoyhteys on heikko. Menetelmää pitäisi kehittää jatkossa siten, että synkronoinnin voisi jakaa useampaan verkkopyyntöön. Vaihtoehtoisesti algoritmissa voitaisiin käyttää HTTP:n sijasta muuta tiedonsiirtoprotokollaa.

Menetelmän toteutuksessa ei hyödynnetä vielä ollenkaan välimuistia. Taustajärjestelmässä on käytössä Redis-muistitietokanta, jota voitaisiin hyödyntää tiedon väliaikaisessa tallentamissa. Mikäli tietokannasta haettavaa tietoa voitaisiin tallentaa välimuistiin, voitaisiin tietokantaan kohdistuvia hakuja vähentää. Redis on nopea ja hyvin skaalautuva tietokanta, joten tämä voisi parantaa järjestelmän skaalautuvuutta käyttäjämäärän kasvaessa.

Tällä hetkellä toteutusjärjestelmässä on vain yksi relaatiotietokanta taustajärjestelmän tietovarastona. Suorituskykyä voitaisiin parantaa lisäämällä tietokannasta lukukopioita. Tällöin tietokantaa on mahdollista lukea useammasta tietokantainstanssista, mutta kirjoitukset suoritetaan vain yhteen. Näin ollen tämä mahdollistaa suuremman määrän lukupyyntöjä kerrallaan. Tällaisessa ratkaisussa on kuitenkin ongelmana se, että tieto monistuu pääinstanssilta lukukopioihin ennalta määräämättömällä viiveellä. Tämän viiveen vaikutus synkronointimenetelmään tulee selvittää.

7. YHTEENVETO

Tässä työssä on esitelty ratkaisu ohjelmistoprojektin vaatimuksiin. Sovelluksen vaatimuksiin kuului mahdollisuus toimia ilman verkkoyhteyttä mahdollisimman samalla tavalla kuin verkkoyhteyden kanssa. Lisäksi sovellusta tulee voida käyttää useammalta laitteelta samanaikaisesti, ja yhdellä laitteella tehdyt muutokset välittyvät muille laitteille.

Edellä mainitut vaatimukset voidaan ratkaista monistamalla käyttäjän tietosisältö laitteille. Kun tieto on monistettu, voidaan sitä käyttää laitteella ilman verkkoyhteyttä. Koska tietosisältöä on tarve muokata päätelaitteella, täytyy muutokset lähettää taustajärjestelmälle verkkoyhteyden ollessa saatavilla. Tiedon siirron tarve on siis kaksisuuntainen: taustajärjestelmältä päätelaitteelle ja toisin päin. Ratkaisuksi esitettyihin vaatimuksiin suunniteltiin synkronointialgoritmi.

Synkronoinnin tarkoituksena on, että kun kaksi laitetta synkronoivat tietonsa, päätyy niiden tietosisältö keskenään samaan tilaan. Tämä siis mahdollistaa sen, että vaikka päätelaite olisi pidempiäkin aikoja verkottomassa tilassa, voidaan sen muutokset siirtää taustajärjestelmälle. Vastaavasti synkronoinnin seurauksena päätelaite saa muilla päätelaitteilla tehdyt muutokset.

Synkronointi toteutettiin kehittämällä oma synkronointialgoritmi. Oman algoritmin kehittämiseen päädyttiin, sillä saatavilla ei ollut projektin asettamiin vaatimuksiin sopivia ratkaisuja.

Algoritmi perustuu siihen, että synkronoitavilla objekteilla on revisionumero. Revisionumeroiden avulla on mahdollista tietää objekteihin tehtyjen muutosten järjestys. Kaikki muutokset objektien revisionumeroihin tehdään taustajärjestelmällä keskitetysti. Synkronointi tehdään aina taustajärjestelmän ja asiakassovelluksen välillä. Sen seurauksena on, että asiakassovelluksen tietosisällön tila vastaa taustajärjestelmän tilaa käyttäjän tietojen osalta.

Työssä esitelty synkronointimenetelmä on osoittautunut toimivaksi ratkaisuksi ongelmaan. Se on toiminut virheettömästi testikäytössä ja suorituskyky on ollut riittävä. Järjestelmälle ollaan tulevaisuudessa tekemässä suorituskykytestausta, jonka

jälkeen voidaan paremmin arvioida skaalautuvuutta tuotantokäytössä.

Keskeinen heikkous menetelmässä on, että se edellyttää kaikkien muutosten lähettämistä kerralla synkronoinnin aikana. Ongelmaksi voi muodostua se, että tietosisältöä on paljon ja saatavilla oleva verkkoyhteys ei ole riittävän hyvä. Tätä ongelmaa voitaisiin lähteä ratkaisemaan muuttamalla menetelmää siten, että se sallisi viestin pilkkomisen tai vaihtamalla siirtoprotokollaa. Lisäksi menetelmän suorituskykyä on todennäköisesti mahdollista parantaa oleellisesti esimerkiksi käyttämällä välimuistitietokantaa taustajärjestelmässä.

Kaiken kaikkiaan suunniteltua ja toteutettua synkronointimenetelmää voidaan pitää onnistuneena ratkaisuna projektin vaatimuksiin. Mikäli sen tiedossa olevat rajoitteet saadaan ratkaistua, voidaan sitä luultavasti hyödyntää sovelluksessa pitkään.

LÄHTEET

[1] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[2] Hannu-Matti Järvinen Ilkka Haikala. *Käyttöjärjestelmät (2. painos)*. Talentum, 2004.

[3] Realm. *Realm Mobile Platform: Data sync, event handling & offline-first functionality*. 2016. [Viitattu 1.11.2016] Saatavilla: https://realm.io/products/realm-mobile-platform/.

[4] Realm. *Realm Professional & Enterprise*. 2017. [Viitattu 19.6.2017] Saatavilla: https://realm.io/docs/realm-object-server/pe-ee/.

[5] A. Kedia and A. Prakash. Data synchronization on android clients. In *2015 IEEE International Conference on Communication Software and Networks (ICCSN)*, pages 212–216, June 2015.

[6] Amazon Web Services, Inc. or its affiliates. *What Is AWS Elastic Beanstalk?* 2016. [Viitattu 7.12.2016] Saatavilla: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/Welcome.html.

[7] Amazon Web Services, Inc. or its affiliates. *The Elastic Beanstalk Command Line Interface (EB CLI)*. 2016. [Viitattu 7.12.2016] Saatavilla: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb3-deploy.html.

[8] Amazon Web Services, Inc. or its affiliates. *eb deploy - AWS Elasic Beanstalk*. 2016. [Viitattu 7.12.2016] Saatavilla: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3.html.

[9] Amazon. *Amazon EC2 Product Details*. 2017. [Viitattu 26.1.2017] Saatavilla: https://aws.amazon.com/ec2/details/.

[10] Amazon. *Shards (Redis)*. 2017. [Viitattu 2.2.2017] Saatavilla: http://docs.aws.amazon.com/AmazonElastiCache/latest/UserGuide/Shards.html.

[11] Amazon. *Amazon ElastiCache*. 2017. [Viitattu 26.1.2017] Saatavilla: https://aws.amazon.com/elasticache/.

[12] Amazon. *What Is Amazon Relational Database Service (Amazon RDS)?* 2017. [Viitattu 26.1.2017] Saatavilla: http://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html.

[13] Amazon. *Using Elastic Beanstalk with Amazon RDS.* 2017. [Viitattu 26.1.2017] Saatavilla: http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/AWSHowTo.RDS.html.

[14] Apple Inc. *Cocoa (Touch).* 2015. [Viitattu 19.1.2017] Saatavilla: https://developer.apple.com/library/content/documentation/Swift/Conceptual/BuildingCocoaApps/MixandMatch.html.

[15] Realm. *Realm Database.* 2017. [Viitattu 13.11.2017], Saatavilla: https://realm.io/products/realm-database/.

[16] AFNetworking. *AFNetworking.* 2017. [Viitattu 13.11.2017], Saatavilla: https://github.com/AFNetworking/AFNetworking.

[17] Apple. *About the URL Loading System.* 2016. [Viitattu 13.11.2017], Saatavilla: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/URLLoadingSystem/URLLoadingSystem.html.

[18] Network Working Group. *Hypertext Transfer Protocol – HTTP/1.1.* 1999. [Viitattu 14.3.2017] Saatavilla: https://tools.ietf.org/html/rfc2616.

[19] Apple Inc. *Why Objective-C?* 2010. [Viitattu 17.1.2017] Saatavilla: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/OOP_ObjC/Articles/ooWhy.html.

[20] Apple Inc. *Swift and Objective-C in the Same Project.* 2016. [Viitattu 17.1.2017] Saatavilla: https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/Cocoa.html.

[21] Mark Lutz. *Programming Python, 4. painos.* O'Reilly Media, Inc., 2011.

[22] Python Software Foundation. *History and License.* 2017. [Viitattu 22.3.2017] Saatavilla: https://docs.python.org/3/license.html.

[23] Mark Lutz. *Learning Python, 5th Edition.* O'Reilly Media, 2013.

[24] Adrian Holovaty and Jacob Kaplan-Moss. *Introducing Django.* 2009. [Viitattu 6.2.2017] Saatavilla: http://djangobook.com/introducing-django/.

[25] The PostgreSQL Global Development Group. *PostgreSQL: About.* 2016. [Viitattu 7.12.2016] Saatavilla: https://www.postgresql.org/about/.

[26] Thomas M. Connolly and Carolyn E. Begg. *DataBase Systems: A Practical Approach to Design, Implementation and Management (4th Edition) (International Computer Science).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[27] Realm. *Introducing Realm*. 2014. [Viitattu 2.2.2017] Saatavilla: https://realm.io/news/introducing-realm/.

[28] Realm. *Realm Docs*. 2017. [Viitattu 13.11.2017], Saatavilla: https://realm.io/docs/objc/latest/.

[29] The PostgreSQL Global Development Group. *PostgreSQL: Documentation: Explicit Locking*. 2017. [Viitattu 20.6.2017] Saatavilla: https://www.postgresql.org/docs/9.4/static/explicit-locking.html/.