Leevi Limnell

Android-laitehallintaratkaisun kommunikaation analyysi ja kehittämismahdollisuudet

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

16.5.2025

Tiivistelmä

Tekijä:	Leevi Limnell
Otsikko:	Android-laitehallintasovelluksen kommunikaation analyysi ja kehittämismahdollisuudet
Sivumäärä:	42 sivua + 1 liite
Aika:	16.5.2025
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Mobile solutions
Ohjaajat:	Tutkijaopettaja Hannu Markkanen Senior Manager Ville Kinnunen

---

Tämän työn tarkoituksena oli tutkia Miradoren pilvipohjaisen laitehallinnan Android-asiakassovelluksen ja palvelimen välistä kommunikaatiota ja sen komponentteja etsien mahdollisia kehityskohteita. Löydetyt mahdolliset kehityskohteet oli tarkoitus esitellä ja ehdottaa niihin ratkaisuja kommunikaation tehostamiseksi.

Työn aikana huomattiin, että kommunikaatiossa on useita yksittäisiä pieniä kehityskohteita. Näihin ehdotettiin parannuksia, joiden avulla kommunikaatio kokonaisuudessaan saataisiin mahdollisimman tehokkaaksi. Myös suurempien muutosten tekemistä kommunikaation komponentteihin pohdittiin. Muutosten vaatima työn määrä sekä niistä saatavat hyödyt kuitenkin jättävät muutosten kannattavuuden kyseenalaiseksi.

Tutkimuksen perusteella kommunikaatio asiakassovelluksen ja palvelimen välillä on kokonaisuuden kannalta toimivaa, vaikka selkeitä pieniä kehityskohteita löytyikin. Kommunikoinnin nykytoteutus olisi mahdollista optimoida kohtuullisella työmäärällä, mikä parantaisi kommunikaation kulkua palvelimen ja asiakassovelluksen välillä.

Avainsanat: Android, kommunikaatio, laitehallinta

Abstract

Author:	Leevi Limnell
Title:	Analysis and Development Opportunities in the Communication of an Android Device Management Solution
Number of Pages:	42 pages + 1 appendix
Date:	16 May 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Mobile solutions
Supervisors:	Hannu Markkanen, Researching Lecturer Ville Kinnunen, Senior Manager

---

The purpose of this project was to examine the communication between the Android client application and the server used in the Miradore cloud-based device management system. The project focused on identifying potential improvements and proposing solutions to enhance this communication.

Throughout the work, it was noticed that there are multiple small areas where improvements could be made. Solutions for these findings were proposed with the goal of making the communication as efficient as possible. Possible larger modifications to the components of the communication were also explored. However, the amount of work required for these changes in comparison to the benefits gained left the feasibility of them questionable.

Based on the performed study, the communication between the client application and the server is effective overall, although some minor areas for improvement were identified. The current implementation of the communication could be optimized with a reasonable amount of work, improving the flow of communication between the server and the client application.

Keywords: Android, communication, mobile device management

Sisällys

Lyhenteet

APK: *Android Application Package.* Android-käyttöjärjestelmän tiedostomuoto, jolla jaetaan ja asennetaan sovelluksia.

EMM: *Enterprise mobility management.* Organisaation mobiililaitteiden, sovellusten, sisällön ja käyttäjäoikeuksien kattavaa hallintaa.

FCM: *Firebase Cloud Messaging.* Pilvipalvelu iOS-, Android- ja verkkosovelluksien viestien ja ilmoitusten lähetykseen.

HTTP: *Hypertext Transfer Protocol.* Protokolla, jota käytetään tiedon siirtämiseen verkon välityksellä mahdollistaen viestinnän esimerkiksi sovelluksen ja palvelimen välillä.

MDM: *Mobile device management.* Mobiililaitteiden, kuten älypuhelimien, tablettitietokoneiden ja kannettavien tietokoneiden, hallintaa.

ORM: *Object-relational mapping*. Ohjelmiston oliomallin mukaisen esityksen kuvaaminen relaatiotietokantamallin mukaiseksi esitykseksi ja kääntäen.

UEM: *Unified endpoint management.* Ohjelmistojärjestelmien luokka, joka tarjoaa yhden hallintaliittymän mobiililaitteille, PC:lle ja muille laitteille.

XML: *Extensible Markup Language.* Merkintäkielien standardi, joka määrittää tietojen muodon loogisella rakenteella.

1 Johdanto

Tämä insinöörityö on tehty Miradore Oy:lle, joka on suomalainen pilvipohjaiseen mobiililaitehallintaan erikoistunut yhtiö. Työn tarkoituksena on tutkia Miradoren Android MDM -asiakassovelluksen ja palvelimen välistä kommunikaatiota, ja sitä käsittelevien komponenttien toimintaa etsien siitä mahdollisia kehityskohteita. Toiminnan selvityksen aikana havaittuihin mahdollisiin kehityskohteisiin ehdotetaan parannuksia ja vaihtoehtoisia toteutustapoja niiden toiminnan tehostamiseksi.

Miradoren laitehallintaratkaisun avulla hallitaan suurta ja kasvavaa laitekantaa, ja uusia laitteita rekisteröidään ohjelmistopalveluun päivittäin. Android-asiakassovellus, jonka kehitys alkoi vuonna 2013, on keskeinen osa yrityksen palvelun toimintaa. Suuri määrä Android-laitteita kommunikoi Miradoren palvelimien kanssa päivittäin, ja ohjelmistoratkaisun käyttämästä kommunikaatioprotokollasta voi mahdollisesti löytyä useitakin mahdollisia parannuskohteita. Suuren laitevolyymin takia pienetkin parannukset voisivat tuoda huomattavia hyötyjä esimerkiksi tiedonsiirron määrän muodossa.

Työn esille tuomaan tietoon pohjaten sovelluksen mahdolliset epäoptimaaliset ratkaisut voidaan myöhemmin päivittää tehostettuihin vaihtoehtoihin. Tarpeen mukaan tiedon pohjalta voidaan myös suorittaa tähän työhön verrattuna syvempää tutkimusta mahdollisten muutosten vaikutuksista ja kannattavuudesta niiltä osin, kuin tarpeelliseksi koetaan.

2 Miradoren Android-sovellus

2.1 Johdatus MDM-ratkaisuihin

Mobiililaitteiden hallinta, eli MDM (Mobile Device Management), tarkoittaa älypuhelimien, tablettien ja kannettavien tietokoneiden hallinnointia. MDM toteutetaan yleensä kolmannen osapuolen tuotteella, joka sisältää hallintaominaisuuksia tuotteen tukemille mobiililaitteille (1, s. 1–2). Mobiililaitteiden hallintasovelluksien tarkoituksena on tarjota keskitetty tapa organisaation laitteiden seurantaan, hallintaan ja turvaamiseen. MDM parantaa turvallisuutta muun muassa pakottaen laitteita noudattamaan asetettuja salasanojen ja salausasetusten konfiguraatioita. MDM-ratkaisun hallitsema laite voidaan lukita etäyhteydellä tai palauttaa tehdasasetuksiin laitteen joutuessa vääriin käsiin. MDM myös helpottaa IT-järjestelmänvalvojien työtä tarjoamalla työkaluja laitteiden provisiointiin, hallintaan ja päivittämiseen. (2, s. 2–3.)

MDM-asiakassovellukset ovat hallitulle laitteelle asentuvia sovelluksia, jotka kontrolloivat sen toimintaa halutulla tavalla hyödyntämällä käyttöjärjestelmän niille antamia erikoisoikeuksia (1, s. 4–5). Nämä asiakassovellukset voivat muun muassa kontrolloida, mitä muita sovelluksia hallitulle laitteelle voidaan asentaa, mitkä laitteen sovelluksista ovat käytettävissä, sekä varmistaa, että sovellusten versiopäivitykset asennetaan ajallaan. (1, s. 3.)

Sovellusten päivitysten lisäksi myös laitteen järjestelmäpäivitykset voidaan asettaa asennettavaksi päivitysten julkaisuhetkellä haittaohjelmien riskin minimoimiseksi. Hallinnointi mahdollistaa myös laitteiden seurannan ja tilan raportoinnin antaen käsityksen laitteiden käytöstä ja niiden konfiguraatioista järjestelmänvalvojille. MDM-ratkaisua hyödyntämällä muun muassa laitteiden käyttöönotto ja ylläpidolliset tehtävät voivat nopeutua huomattavasti lisäten työntekijöiden tuottavuutta sekä tehostaen organisaation toimintaa (2, s. 6).

2.2 Miradoren Android-ratkaisun yleiskatsaus

Miradore Oy on MDM-tarjoaja, jonka pilvipohjaisella ratkaisulla voidaan hallita Android-, iOS-, macOS- ja Windows-laitteita samasta käyttöliittymästä (3). Koska Miradoren tarjoamalla ohjelmistolla voidaan hallita usean käyttöjärjestelmän laitteita, se MDM-ratkaisua tarkemmin luokitellaan UEM (Unified Endpoint Management) -ratkaisuksi (1, s. 2). Alalla vakiintunut termi MDM on kuitenkin yleisesti käytössä myös UEM-tuotteista puhuttaessa, kuten Miradoren laitehallinnankin tapauksessa.

Miradoren laitehallinnan ohjelmistopalvelu rakentuu taustalla toimivan palvelinkokonaisuuden varaan, jonka kanssa siihen rekisteröidyt laitteet kommunikoivat. Rekisteröityjen laitteiden tilaa voidaan seurata ja niiden toimintaa hallita helposti Miradoren verkkokäyttöliittymän avulla.

Android-laitetta rekisteröitäessä laitteelle joko manuaalisesti asennetaan tai siihen automaattisesti asentuu palvelimen kanssa keskusteleva asiakassovellus (4). Tämän päätelaitteen sovelluksen tarkoituksena on pitää laite ja palvelin synkronoituna ja huolehtia määriteltyjen konfiguraatioiden ja tehtävien suorittamisesta laitteella. Konfiguraatiot ovat kokoelma monenlaisia asetuksia, joilla on pitkäaikainen vaikutus laitteen toimintaan. Niillä voidaan muun muassa rajoittaa laitteen toimintaa, esimerkiksi estämällä tiettyjen sovellusten- tai Bluetooth-ominaisuuden käyttö kokonaan. Tehtäviksi voidaan lukea kertaluontoiset toimet, kuten tiedoston lataaminen tai laitteen palauttaminen tehdasasetuksiin.

Miradoren Android-asiakassovellus kommunikoi palvelimen kanssa HTTP-viestiketjujen avulla. Java-sovellukseen, jonka kehitystyö on alkanut vuonna 2013, ei ole tehty suuria uudelleenkirjoituksia vuosien aikana. Nykystandardien mukaan sovelluksen arkkitehtuurissa onkin havaittavissa jokseenkin vanhentuneita tekniikoita. Kehitystekniikoiden vanhuus näkyy muun muassa modernien, alun perin vuonna 2017 julkaistujen, Android Architecture -konseptien ja komponenttien käytön puutteena (5). Lisäksi sovelluksessa on käytetty niukasti ulkoisia

kirjastoja toiminnallisuuksien toteuttamiseen, joiden käyttö mahdollisesti helpottaisi kehitystyötä ja voisi tuoda hyötyjä myös automaatiotestaamiseen.

Asiakassovelluksen ja palvelimen välillä kulkee lukuisia viestejä päivittäin ohjelmiston normaalissa käytössä. Kommunikaatioketju voi esimerkiksi alkaa Miradoren järjestelmänvalvojan asettaessa laitteelle uuden konfiguraation (6). Laitteelle asetetut konfiguraatiot ja tehtävät tallennetaan palvelimen tietokantaan odottamaan seuraavaa asiakassovelluksen yhteydenottoa. Tilanteen mukaan palvelin voi tämän jälkeen lähettää laitteelle synkronointipyynnön, jonka vastaanottaessaan asiakassovellus ottaa heti yhteyden palvelimeen. Jos pyyntöä ei lähetetä laitteelle, asetetut konfiguraatiot ja tehtävät jäävät odottamaan seuraavaa ajoitettua synkronointia, jonka aikana asiakassovellus hakee näitä palvelimelta.

Kun tehtävä tai konfiguraatio saavuttaa asiakassovelluksen, se toteuttaa määritellyn toimenpiteen, kuten käynnistää tiedoston lataamisen tai poistaa Bluetooth-ominaisuuden käytöstä. Tämän jälkeen asiakassovellus raportoi toimenpiteen tuloksen takaisin palvelimelle, esimerkiksi kertoen, että laitteen Bluetooth-ominaisuus otettiin pois käytöstä onnistuneesti. Joissakin tilanteissa lopullista tulosta toimenpiteelle ei voida heti tietää. Esimerkiksi sovelluksen latautumisessa ja asentumisessa voi kestää useitakin minuutteja. Tällöin laite ilmoittaa aluksi toimenpiteen käynnistymisestä ja raportoi lopullisen tuloksen myöhemmin.

Tämä asiakassovelluksen ja palvelimen välinen kommunikaatio on keskeisessä osassa Miradoren laitehallintaratkaisun toimintaa. Tämän työn tarkoituksena on varmistaa kommunikaation tehokkuus ja mahdollisesti ehdottaa konkreettisia parannuksia sen toimintaan. Tulevissa luvuissa käsitellään kommunikaatiota yleisesti, esitellään sitä tarkemmin esimerkin kautta, analysoidaan kommunikaation toimintaa ja lopuksi tuodaan esille mahdollisesti löydettyjä parannuskohteita.

3 Kommunikaation tarpeet

3.1 Kommunikaation tarkoitus

Kommunikaation päätarkoituksena on pitää laite synkronoituna palvelimen kanssa varmistaen laitteen noudattavan määritettyjä konfiguraatioita ja suorittavan sille määrätyt tehtävät. Palvelimen ja asiakassovelluksen välillä tapahtuva kommunikointi rakentuu pääosin sovellukseen rakennetun synkronointitapahtuman ympärille, jonka toiminnallisuudet voidaan jakaa teemallisesti seuraaviin osiin:

1. Laitteen tietojen lähettäminen. Asiakassovellus päivittää laitteen nykyisen tilan palvelimelle. Lähetettyihin tietoihin kuuluu täydellinen laiteinventaario sisältäen muun muassa laitetiedot, muut asennetut sovellukset versioineen ja käytetyt mobiili- ja Wifi-verkot.

2. Konfiguraatioiden päivittäminen. Laitteelle lisätyt konfiguraatiot haetaan palvelimelta ja ne asetetaan käyttöön. Konfiguraatioiden toimenpiteitä ovat esimerkiksi tunnuskoodin asettaminen, taustakuvan vaihtaminen tai laitteen toimintojen ja ominaisuuksien rajoittaminen.

3. Tehtävien suorittaminen. Asiakassovellus hakee laitteen suoritettavaksi asetetut tehtävät palvelimelta, ja suorittaa vaadittavat toimenpiteet. Näihin lukeutuu muun muassa uuden sovelluksen asentaminen, tehdasasetusten palauttaminen tai laitteen lukitseminen.

Laitteella suoritettavat synkronointitapahtumat voidaan erotella kahteen muotoon. Kevyemmässä synkronoinnissa haetaan uudet konfiguraatiot ja tehtävät palvelimelta sekä päivitetään laitteella suoritettujen toimenpiteiden tilat palvelimelle. Raskaammassa versiossa myös laitteen inventaario toimitetaan palvelimelle kaiken muun lisäksi. Inventaariota ei lähetetä jokaisen synkronoinnin yhteydessä, sillä sen sisältämät tiedot eivät yleensä muutu usein. Tästä syystä asiakassovelluksen ei tarvitse kerätä tarkkaa inventaariota laitteesta jokaisen synkronoinnin yhteydessä. Tavanomainen Android-inventaario on kooltaan

melkein 200 kilotavua, kun taas kommunikaation muiden viestien sisältö jää yleensä alle kilotavun. Kahden erilaisen synkronoinnin ansiosta palvelimen ylimääräiseltä rasittumiselta vältytään, kun sen ei tarvitse jokaisen yhteydenoton aikana käsitellä suurta inventaarioviestiä.

Synkronointitapahtumien aikana asiakassovellus ilmoittaa palvelimelle laitteen piilotettuja määrityksiä, joita käytetään esimerkiksi laitteen synkronointipyyntöjen lähettämiseen palvelimelta. Synkronoinnit käsittelevät myös autentikointitunnusten automaattisen päivittämisen varmistaen kommunikaation sujuvan jatkumisen.

Lähes kaikki kommunikointi hoituukin näiden synkronointitapahtumien aikana. Synkronointitapahtumien ulkopuoliseen kommunikointiin kuuluu kuitenkin osa toiminnoista kuten laitteen sijainnin päivitykset palvelimelle, mitkä tapahtuvat Miradoren asiakkaan asettamien konfiguraatioiden mukaan. Myös laitteen lokitietojen lähetys ongelmatilanteissa Miradoren asiakastuelle tapahtuu synkronointitapahtumien ulkopuolella.

3.2 Kommunikaation tiheys ja ajoitus

Synkronointitapahtumat pohjautuvat ajoitettuun polling-menetelmään. Tässä menetelmässä laite ottaa yhteyden palvelimeen tietyn aikataulun mukaan ja tiedustelee uusia toimenpiteitä palvelimelta HTTP-pyyntöjen kautta. Automaattisen aikataulutuksen ohella synkronointi voidaan käynnistää myös palvelimen määrittämällä ajanhetkellä. Tällöin palvelimelta lähetetty synkronointipyyntö laitteelle käynnistää synkronoinnin heti ohittaen asiakassovelluksen ajoitusmekanismin.

Ajoitetusti suoritettavan synkronoinnin lisäksi synkronointipyyntöjä tarvitaan, sillä jotkin laitteelle asetetut toimenpiteet ovat aikakriittisiä ja niiden vaikutuksen on näyttävä laitteella mahdollisimman nopeasti. Esimerkiksi laitteen katoamis- tai varkaustilanteessa laitteen lukitsemisen tai nollaamisen on erittäin tärkeää tapahtua mahdollisimman nopeasti. Kaikkien toimenpiteiden suorittaminen ei

kuitenkaan ole aina yhtä kiireellistä, kuten laitteen uuden taustakuvan asettaminen.

Tästä syystä tärkeiden toimenpiteiden yhteydessä, esimerkiksi tehdasasetuksia palauttaessa, heräteviesti lähetetään laitteelle aina automaattisesti. Toisissa tilanteissa Miradoren järjestelmänvalvojalle annetaan valinta heräteviestin lähettämisestä, kun toimenpide on asetettu laitteelle, jotta haluttaessa muutokset olisivat mahdollisimman pian havaittavissa laitteella.

Asiakassovelluksen oletusasetuksena on suorittaa synkronointitapahtuma noin kolmen tunnin välein, ja laitteen inventaario lähetetään kerran vuorokaudessa yksittäisen synkronoinnin yhteydessä. Arvot ovat kuitenkin konfiguroitavissa, ja tarpeen mukaan synkronointi sekä inventaarion lähetys voidaan asettaa tapahtumaan jopa 15 minuutin välein. Sovellus ei kuitenkaan suorita yhteydenottoa palvelimeen täysin määritetyllä ajan hetkellä, vaan asetettu arvo on maksimiaika, jolloin synkronointi viimeistään suoritetaan. Liukuvan aikaikkunan ansiosta laite voi odottaa omien resurssiensa kannalta optimaalisinta hetkeä synkronoinnin suorittamiseksi, eikä palvelimelle synny toistuvia ruuhkia tiettyihin kellonaikoihin synkronointien hajautuessa. Koodiesimerkistä 1 nähdään, miten synkronointitapahtumien ajoitus on käytännössä toteutettu PeriodicQueryJob-luokan scheduleJob-metodissa.

```
public static void scheduleJob() {
  ...

  // Set minimum interval to 15 mins
  long intervalInMinutes = Math.max(settings.getQueryInterval(), 15 );

  // Use half of the interval as a flex period. If interval is 60
  // minutes, flex period would be 30 minutes,
  // meaning the job would trigger between 30 and 60 minutes from
  // now at a time WorkManager considers best.
  PeriodicWorkRequest.Builder request = new PeriodicWorkRequest
    .Builder(
      PeriodicQueryJob.class,
      intervalInMinutes,
      TimeUnit.MINUTES,
      intervalInMinutes / 2,
      TimeUnit.MINUTES
    ).addTag( TAG );
  ...

  WorkManager.getInstance().enqueueUniquePeriodicWork(
    TAG,
    ExistingPeriodicWorkPolicy.REPLACE,
    request.build()
  );
}
```

Esimerkkikoodi 1. Ote PeriodicQueryJob-luokan metodista, jota kutsumalla synkronointitapahtumat ajoitetaan.

Metodia scheduleJob suoritettaessa laitteelle tallennettu synkronointiaikaväli haetaan asiakassovelluksen asetuksista. Tallennetun aikavälin ollessa suurempi kuin mahdollinen minimiaikaväli (15 minuuttia), sitä käytetään PeriodicWorkRequest-luokan pyynnön rakentamiseen, joka myöhemmin käynnistää synkronoinnin määritellyn ajoituksen mukaan.

Rakennettu PeriodicWorkRequest-pyyntö kuuluu Androidin WorkManager-kirjastoon, jota asiakassovellus hyödyntää synkronointien ajoittamiseen. WorkManager onkin Googlen suosittelema ratkaisu jatkuvien tehtävien hallintaan (7). Rakennetulle pyynnölle määritellään aikaikkuna, jolloin synkronointi halutaan suoritettavan. Tämä tapahtuu puolittamalla määritetty maksimiaika, josta saatua arvoa ja määritettyä maksimiaikaa käytetään intervallina.

Tämä rakennettu pyyntö määrityksineen annetaan eteenpäin WorkManager-objektin hoidettavaksi. Android-käyttöjärjestelmä pitää tämän jälkeen huolen

synkronoinnin käynnistämisestä varmistaen sen suorittamisen ja uudelleenajoittamisen, esimerkiksi laitteen uudelleenkäynnistyksen yhteydessä.

Synkronointien suorittaminen automaattisen aikataulun ulkopuolella onnistuu aiemmin esille tuotujen synkronointipyyntöjen avulla. Synkronointipyynnöt ovat käytännössä heräteviestejä, jotka laitteelle saapuessaan käynnistävät synkronointitapahtuman. Nämä viestit kulkeutuvat muusta tietoliikenteestä poiketen Googlen Firebase Cloud Messaging (FCM) -alustaa käyttäen. FCM on pilvipalvelu, joka mahdollistaa ilmoitusten ja viestien lähettämisen Android-, iOS- ja Web-sovelluksiin (8). Palvelun käyttäminen takaa kiireellisten viestien lähettämisen laitteelle luotettavasti ja nopeasti, valmiiksi optimoidulla ratkaisulla laitteen resurssien kannalta. Heräteviesti lähetään Googlen FCM-ohjelmointirajapintaan, joka järjestää viestin turvallisen kuljetuksen asiakassovellukseen.

FCM-palvelun viestien vastaanottaminen vaatii Googlen palvelujen asentamisen päätelaitteelle. Kaikki laitteet eivät kuitenkaan syystä tai toisesta tue Googlen palveluja, jolloin laitteille ei voida lähettää palvelimen heräteviestejä. Näissä tilanteissa laitteet toimivat ajoitettujen synkronointien varassa.

Mikäli tarve vaatii, synkronointitapahtumat voidaan käynnistää myös manuaalisesti laitteella suoraan asiakassovelluksesta.

4 Kommunikaation nykytoteutus

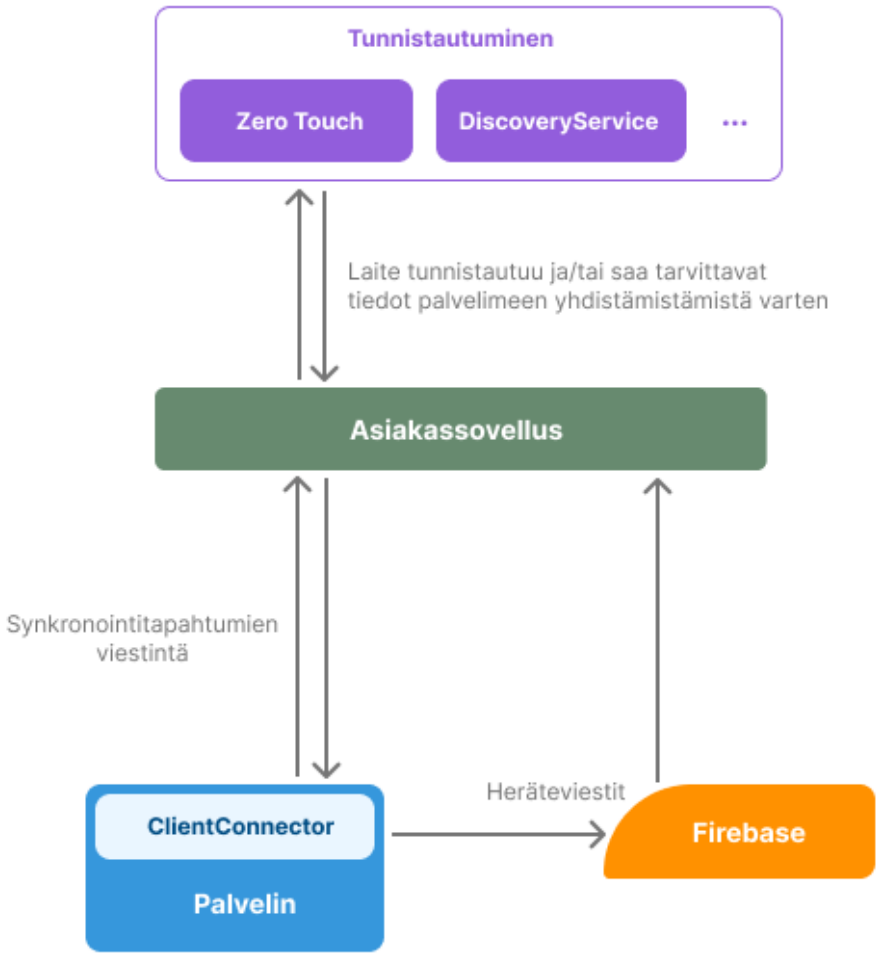
Kommunikointiprotokollat ovat joukko sääntöjä ja ohjeita tiedonsiirtoon verkon välityksellä kahden tai useamman systeemin osan välillä. Protokollat määrittelevät muun muassa viestien rakennetta, virheenkäsittelyä ja semantiikkaa. Nämä ohjeet ovat tärkeitä kaikenlaisen kommunikaation mahdollistamiseksi, yksinkertaisesta tiedonsiirrosta monimutkaisiin vuorovaikutuksiin systeemien osien välillä.

Miradoren asiakassovelluksen ja palvelimen välinen kommunikointi perustuu HTTP-protokollaan (9). HTTP määrittelee omat sääntönsä ja ohjeensa, joita

Miradoren asiakassovellus noudattaa. Näihin lukeutuvat HTTP-pyynnön osat, kuten URL (Uniform Resource Locator), joka tarkentaa halutun resurssin sijainnin palvelimella, sekä metatietoja pyynnöstä sisältävä otsake (Header). Miradore määrittelee omat sääntönsä HTTP:n määrittelyjen perusteella muodostaen oman protokollansa, jonka avulla palvelin ja asiakassovellus voivat kommunikoida. Tämän protokollan määritelmään sisältyvät esimerkiksi HTTP-viestien sisältö ja rakenne, autentikoinnin käsittely ja palvelimen päätepisteiden muodostuminen.

4.1 Kommunikaation kuvaus

Kommunikaatio Android-asiakassovelluksen ja palvelimen välillä saa alkunsa rekisteröitäessä laitetta Miradoren laitehallintaohjelmistoon. Tällöin asiakassovellus on rekisteröintitavan mukaan yhteydessä yhteen Miradoren ohjelmiston käyttämistä autentikointipalveluista, josta asiakassovellus saa tarvittavat tiedot palvelimen kanssa kommunikointiin. Kuvassa 1 kuvataan yleisellä tasolla kommunikaation kulkua sen eri osien välillä.



Kuva 1. Asiakassovelluksen ja palvelimen kommunikaatio yleisellä tasolla.

Käytetyn rekisteröitymistavan mukaan asiakassovellus voi pyytää käyttäjää syöttämään kirjautumistunnukset autentikoidakseen itsensä. Kaikkien rekisteröintitapojen yhteydessä manuaalista tunnuksien syöttämistä ei kuitenkaan tarvita, vaan asiakassovellus voi saada tarvittavat tiedot laitteen käyttöönoton yhteydessä automaattisesti. Tämä onnistuu esimerkiksi laitevalmistajan palvelusta, jonka kautta laite on määritetty Miradoren laitehallintaan.

Manuaalisen autentikoinnin yhteydessä käyttäjän syöttämiä tunnuksia käytetään yhdistettäessä Miradoren DiscoveryService-palveluun. Tämän palvelun tarkoitus on autentikoida asiakassovellus ja palauttaa sille tiedot, joita käyttäen

voidaan muodostaa yhteys Miradoren palvelimeen. Jos laite halutaan rekisteröidä laitehallintaan automaattisesti heti käyttöönoton yhteydessä, voidaan tähän hyödyntää esimerkiksi Googlen Zero-Touch-palvelua (10). Tiiviistetysti, Zero-Touch mahdollistaa yrityksen omistaman laitteen rekisteröinnin MDM-palveluun automaattisesti laitteen ensimmäisen käynnistymisen yhteydessä, ennen laitteen käyttöönottoa. Zero-Touch-ympäristön käyttöä varten laitteen omistaja asettaa Miradoren verkkokäyttöliittymästä saatavat tiedot Googlen Zero-Touch-portaaliin automatisoidun rekisteröinnin mahdollistamiseksi (11). Portaalista ne kulkeutuvat laitteelle ensimmäisen käynnistämisen yhteydessä automaattisesti.

Laitteen saatua tarvittavat tiedot kommunikointiin tavalla tai toisella yhteys Miradoren palvelimeen voidaan muodostaa, ja jatkossa asiakassovelluksen viestit lähetetään palvelimen ClientConnector-komponentille. ClientConnector sisältää yksittäisen päätepisteen, joka käsittelee asiakassovelluksen lähettämät tiedot ja palauttaa sille asetetut konfiguraatiot ja tehtävät. Kaikki kommunikaatio asiakassovelluksen ja palvelimen välillä tapahtuu tämän päätepisteen kautta, lukuun ottamatta FCM-palvelun heräteviestejä sekä laitteen lokien lähetystä ongelmanratkaisua varten, joka ohjautuu DiscoveryService-palveluun.

Palvelimen ja sovelluksen väliset viestit käyttävät XML (Extensible Markup Language) -standardia datan mallintamiseen. XML on joustava sekä ihmisten että koneiden luettavissa oleva tekstimuoto, jolla voidaan esittää rakenteellista dataa. Asiakassovelluksen viestit rakennetaan XML-muodossa, sarjoitetaan (Serialize) tavutaulukkoon siirtoa varten ja lähetetään palvelimelle. Palvelimella viestit puretaan ja prosessoidaan sekä käsitellään mahdolliset virheet asianmukaisesti.

HTTP-kommunikaation toteuttamiseen asiakassovellus käyttää java.net:in kirjaston HttpUrlConnection-komponenttia (12). Java-net-kirjaston käyttäminen mahdollistaa halutunlaisen ratkaisun toteuttamisen HTTP-pyyntöjen lähettämiseen. Komponentin käyttäminen ei myöskään vaadi ulkoisien kirjastojen lisäämistä HttpUrlConnection-komponentin sisältyessä Javaan itsessään. Nykystandardien mittapuulla kirjastolle löytyisi mahdollisesti kuitenkin parempia

vaihtoehtoja, jotka tarjoaisivat laajemman kirjon ominaisuuksia, mikäli niille tarvetta ilmenisi. Jos kommunikaatiossa esimerkiksi haluttaisiin siirtyä käyttämään HTTP:n versiota 2 nykyisen 1.1-version sijaan, ei kyseinen toteutus tukisi tätä. Tällöin voitaisiin siirtyä käyttämään muun muassa Javan uudempaa HttpClient-komponenttia, johon HTTP/2-tuki on lisätty (13).

4.2 Kommunikaatioesimerkki

Kuten luvussa 3.1 tuotiin esille, lähes kaikki palvelimen ja laitteen välinen tiedonsiirto tapahtuu synkronointitapahtumien kautta. Yksinkertaistetusti synkronointi voidaan jakaa kahteen kokonaisuuteen, jonka ensimmäisen osan tarkoituksena on päivittää laitteen tiedot ja raportoida suoritettujen toimenpiteiden tilat palvelimelle. Jälkimmäinen osa hakee uudet tehtävät ja konfiguraatiot palvelimelta ja suorittaa niiden vaatimat toimenpiteet laitteella.

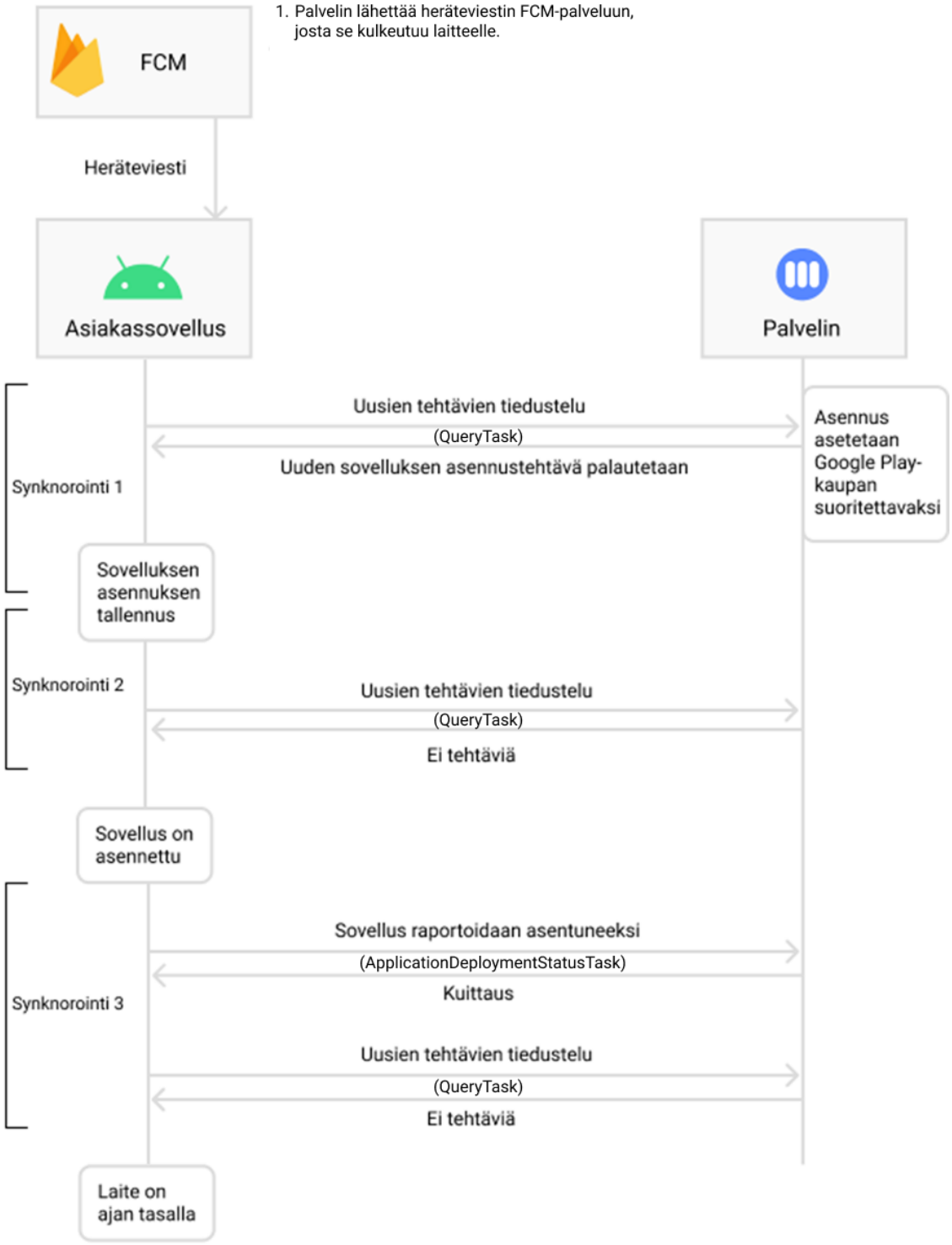
Yksittäinen laitteen synkronointitapahtuma ei useimmissa tilanteissa riitä päivittämään laitetta palvelimelle asetettuun tilaan, vaan laitteella suoritetaan useampi synkronointi yhden toimenpiteen kokonaan loppuun viemiseksi. Esimerkiksi ensimmäinen synkronointitapahtuma voi aloittaa jonkin laitteelle asetetun toimenpiteen. Tämän jälkeen asiakassovellus jää odottamaan toimenpiteen valmistumista. Toimenpiteen valmistuttua aloitetaan uusi synkronointitapahtuma, joka raportoi suoritetun toimenpiteen lopputuloksen.

Synkronoinnin yhteydessä laite lähettää useita HTTP-pyyntöjä tarpeen mukaan. Synkronoinnin aikana asiakassovelluksen ja palvelimen välillä voi liikkua yhdestä HTTP-viestistä kymmeniin viesteihin, laitteen tilan ja palvelimen tilan mukaan.

Synkronointitapahtumien kommunikaation kulkua voidaan selvittää tarkemmin esimerkin avulla. Otetaan esimerkiksi yleinen tilanne, jossa Miradoren laitehallintaan rekisteröityyn Android-laitteeseen asennetaan uusi sovellus Managed Google Play -kauppaa käyttäen. Tätä kautta sovelluksia asennettaessa, niiden varsinainen asentuminen laitteelle tapahtuu automaattisesti Play-kaupan kautta.

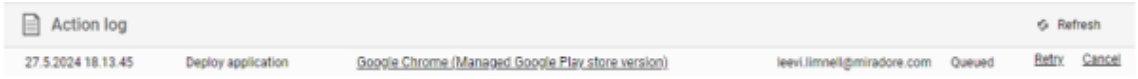
Tässä esimerkissä oletetaan, että kaikki tarvittavat toimenpiteet toimintoa varten, kuten Managed Google Play Account -tilin luominen laitteelle, on suoritettu (14).

Kuvassa 2 esitellään uuden sovelluksen asennusprosessi. Prosessin kulku on kuvattu vaiheittain, ja siinä näytetään kommunikaatio palvelimen ja asiakassovelluksen välillä. Kuvassa nähdään asiakassovellus vasemmalla ja palvelin oikealla. Näiden välille piirretty nuolipari esittää HTTP-pyyntöä, jonka sisältö näkyy nuolen yhteydessä. Pyynnön lähettänyt asiakassovelluksen koodiluokka on merkitty sulkuihin nuolien väliin. Kuvasta nähdään myös uuden sovelluksen asennuksen prosessin aikana käynnistettyjen kolmen synkronointitapahtuman sisältö sekä niiden alkamis- ja loppumiskohdat.



Kuva 2. Esimerkkitilanteen kommunikointi asiakassovelluksen ja palvelimen välillä. Vasemmalla merkitty synkronointitapahtumat, jotka asiakassovellus käynnistää.

Uuden sovelluksen asentamisen toimintaketju alkaa, kun uuden sovelluksen asennus on asetettu laitteelle. Sovelluksen asennuksen voi aloittaa esimerkiksi Miradoren järjestelmänvalvoja verkkokäyttöliittymästä käsin. Sovelluksen asennustehtävä on sen asettamisen jälkeen palvelimen tietokannassa odottamassa laitteen seuraavaa yhteydenottoa. Sovelluksen asentumisen tilaa voidaan seurata palvelimen toimintolokista, joka päivittyy hallitun laitteen lähettämien tietojen mukaan. Kuvassa 3 näkyy esimerkkitilanteen sovellus ”Queued”-tilassa Miradoren verkkokäyttöliittymän toimintolokissa.



Kuva 3. Google Chrome-sovellus laitteen toimintolokissa Miradoren verkkokäyttöliittymässä.

Kun asennustehtävä on Queued-tilassa, laitteelle on lähetetty FCM-palvelun kautta heräteviesti synkronoinnin käynnistämiseksi. Viestin lähetys palvelun kautta tapahtuu nopeasti, suurimassa osassa tapauksia sen vieminen laitteelle kestää vain muutamia sekunteja.

Asiakassovelluksen vastaanottaessa heräteviestin esimerkin ensimmäinen synkronointitapahtuma käynnistyy. Synkronointitapahtumat muodostuvat asiakassovelluksessa yhdeksästä erillisestä Task-koodiluokasta, jotka kaikki instansoidaan ja suoritetaan synkronoinnin toteuttamiseksi.

Koodiluokkien voidaan ajatella suorittavan synkronoinnin kolmessa osassa. Ensimmäisenä tarkistetaan asiakassovelluksen tietokanta aiemmin suoritettujen toimenpiteiden varalta. Suoritettavat koodiluokat tarkistavat ja lähettävät palvelimelle tiedot laitteella suoritetuista toimenpiteistä, joita ei ole vielä raportoitu. Seuraavaksi kerätään laitteen inventaario, jos viimeisimmästä on kulunut tarpeeksi aikaa, sekä lähetetään laitteen tilatiedot, kuten tieto laitteen vapaan muistin määrästä. Viimeisenä aloitetaan mahdollisten uusien konfiguraatioiden ja tehtävien tiedustelu palvelimelta.

Seuraavassa koodiesimerkissä 2 näkyvät kaikki synkronoinnin muodostavat koodiluokat, jotka instansoidaan create-metodikutsulla ja asetetaan asiakassovelluksen suoritettavaksi yksi kerrallaan:

```
ITaskQueue queue = Services.getTaskQueue();
queue.addTask( ConfigurationProfileDeploymentStatusTask.create() );
queue.addTask( ApplicationDeploymentStatusTask.create() );
queue.addTask( FileDeploymentStatusTask.create() );
queue.addTask( CertificateDeploymentStatusTask.create() );
queue.addTask( SecurityActionStatusTask.create() );
queue.addTask( NotificationMessageStatusTask.create() );
queue.addTask( DeviceStatusReportingTask.create() );
queue.addTask( InventoryTask.create( aForceInventorySending ) );
queue.addTask( QueryTask.create( aSchedulePeriodicQuery ) );
```

Esimerkkikoodi 2. Synkronointitapahtuman muodostavat koodiluokat.

Esimerkin ensimmäiset kuusi StatusTask-koodiluokkaa käyvät kukin läpi laitteen suoritettujen toimenpiteiden tiloja omalta vastuualueeltaan. Esimerkiksi sovellusten asennuksien tilaa laitteella käsittelevä ApplicationDeploymentStatusTask ilmoittaa palvelimelle sovelluksen asennuksesta sen tapahduttua. Kaikki synkronoinnin muodostavat koodiluokkien instanssit lisätään ITaskQueue-luokan jonoon, josta ne suoritetaan taustalla yksi kerrallaan kutsuen luokkien doExecute-metodia.

Kaikkien StatusTask-koodiluokkien doExecute-metodin toimivat samalla tavalla. Ne lukevat sovelluksen tietokannasta oman vastuualueensa suoritetut toimenpiteet, joiden viimeisintä tilaa ei ole vielä päivitetty palvelimelle. Löytäessään raportoitavan tilan asiakassovellus rakentaa viestin ja lähettää sen palvelimelle. Koodiesimerkistä 3 nähdään, miten sovelluksista vastaavan ApplicationStatusTask-koodiluokan doExecute-metodi käytännössä on toteutettu.

```
@Override
protected void doExecute() {

  ...

  List<AppDeploymentContainer> deployments
    = database.loadPendingData();

  ...

  for (AppDeploymentContainer deployment : deployments) {

    ...

    AppDeploymentStatusMessage message;

    ...

    ServerMessage response = Services.getConnectionManager()
      .reportAppDeploymentStatus(message);

    ...

  }

  ...

}
```

Esimerkkikoodi 3. Yksinkertaistettu ote ApplicationDeploymentStatusTask-luokan doExecute-metodista.

Mikäli raportoitava tila löytyy, asiakassovellus lähettää HTTP-pyynnön heti palvelimelle jääden odottamaan sen vastausta. Vastauksen saapuessa lähetetty tila päivitetään raportoiduksi asiakassovelluksen tietokantaan. Prosessi toistetaan jokaiselle löytyneelle raportoimattomalle tilalle yksi kerrallaan, kunnes kaikki raportoimattomat tilat on lähetetty palvelimelle. Jos raportoitavia tiloja ei löydetä, metodi ei jatka suorittamista ja seuraava synkronoinnin koodiluokka aloittaa toimintansa.

Uuden sovelluksen asennuksen ensimmäisen synkronoinnin aikana ei löytynyt päivitettäviä tiloja, joten asiakassovellus jatkaa seuraavaan synkronoinnin vaiheeseen ilman HTTP-pyyntöjen lähettämistä.

Seuraavaksi suoritettavat, esimerkkikoodi 2. kolmanneksi ja toiseksi viimeiset koodiluokat ovat vastuussa laitteen kattavasta tietojen keräämisestä

synkronoinnin aikana. DeviceStatusReportingTask-koodiluokan keräämät tiedot sisältävät laitetietoja, kuten tietoa akusta, laitteen sensoreista ja muistin käytöstä.

InventoryTask-koodiluokan keräämään ohjelmistotason informaatioon kuuluu muun muassa käyttöjärjestelmätiedot sekä lista asennetuista sovelluksista ja laitteen tiedostoista. Palvelimelta lähetetty heräteviesti ei aina käynnistä tätä inventaarion keräystä synkronoinnin yhteydessä. Jos asiakassovellus huomaa viimeisestä inventaarion toimittamisesta kuluneen tarpeeksi aikaa, inventaario kerätään ja toimitetaan palvelimelle. Inventaarion keräys voidaan myös palvelimen tai laitteen pyynnöstä suorittaa millä tahansa ajanhetkellä tarpeen mukaan. InventoryTask kuljettaa kerätyt tiedot palvelimelle omalla HTTP-kutsullaan. Palvelin kuittaa viestin vastaanottamisen ja aloittaa sen prosessoinnin asiakassovelluksen jatkaessa toimintaansa.

Tämän esimerkkitilanteen aikana suoritetuista synkronoinneista yksikään ei lähetä laiteinventaariota, kuten kuvasta 2 käy ilmi.

Viimeisenä suoritettavan QueryTask-koodiluokan vastuulla on hakea uudet laitteelle asetetut toimenpiteet. QueryTask tiedustelee uusia toimenpiteitä palvelimelta, ja saa nyt vastauksena sovelluksen asentamistehtävän. QueryTask käynnistää vastauksen saadessaan laitteella uusia sovelluksen asennuksia käsittelevän prosessin. Tämän jälkeen QueryTask on suorittanut osuutensa tähän synkronointitapahtumaan. Sovelluksien asennuksia käsittelevä prosessi merkitsee asennuksen alkaneeksi asiakassovelluksen tietokantaan.

Tapahtumaketjun ensimmäinen synkronointitapahtuma on nyt suoritettu kokonaan. Sovelluksen asennusta käsittelevä prosessi käynnistää viimeisenä toimintonaan synkronoinnin uudelleen, sillä QueryTask ei palauta kaikkia laitteelle määritettyjä toimenpiteitä kerrallaan. Varmuus siitä, että laite ja palvelin on synkronoitu täysin, saadaan kun uusien toimenpiteiden kyselyyn palautetaan palvelimelta tyhjä vastaus.

Synkronointitapahtuma 2 alkaa. Aiemmin esitellyn ITaskQueue-jonon koodiluokat suoritetaan nyt uudelleen. Tälläkään kerralla yksikään StatusTask-koodiluokka ei löydä suoritettuja toimenpiteitä, joiden tila tulisi ilmoittaa. Inventaarion kerääminen ohitetaan, ja synkronoinnin viimeisenä osuutena suoritetaan QueryTask, joka tällä kertaa saa tyhjän vastauksen. Asiakassovellus ja palvelin on nyt synkronoitu, eikä synkronointitapahtumaa toistaiseksi tarvitse jatkaa.

Kun jossain vaiheessa uusi sovellus asentuu Managed Google Play -kaupan kautta, Android-käyttöjärjestelmä lähettää tästä tiedon asiakassovellukseen, joka kuuntelee laitteelle asennettujen sovelluksen muutoksia. Laitteen tietokanta päivitetään onnistuneella sovelluksen asennuksen tilalla ja viimeinen synkronointitapahtuma aloitetaan.

Synkronoinnin aikana suoritettava ApplicationDeploymentStatusTask huomaa tällä kertaa onnistuneen tilan ja päivittää sen palvelimelle. Uusien toimenpiteiden kyselyyn saadaan taas tyhjä vastaus ja synkronointi on suoritettu. Uuden sovelluksen asennusprosessi on nyt viety loppuun onnistuneesti, ja Miradoren järjestelmänvalvoja näkee sen onnistuneena verkkokäyttöliittymässä. Tämän esimerkkitilanteen aikana asiakassovellus lähetti neljä HTTP-pyyntöä kolmen käynnistetyn synkronointitapahtuman sisällä.

5 Kommunikaation toteutuksen analyysi

Nykyisessä kommunikaation toteutuksessa ei ole ongelmia, jotka ilmenisivät MDM-palvelun päivittäisessä toiminnassa suurissa määrin. Paikoittain yksittäisiä parannuksia viestiketjun toimintaan olisi kuitenkin mahdollista tehdä.

Synkronoinnin suorituksen komponenttiriippuvuus

On vaikea käsittää tarkasti, mitä suuren viestintäketjun aikana asiakassovelluksessa oikeastaan tapahtuu, koska useat asiakassovelluksen koodin prosessit käynnistävät synkronointitapahtuman uudelleen eri tilanteiden mukaan. Tämä

johtaa vaikeasti seurattaviin, monimutkaisiin riippuvuuksiin sovelluksen osien välillä kommunikaation suorittamiseksi.

Voisi olettaa, että laitteen synkronointitapahtuman kommunikaatiokulku vietäisiin loppuun asti, vaikka sen yksittäisissä osissa esiintyisi ongelmia. Näin ei kuitenkaan aina ole. Esimerkiksi suurten tiedostojen latauksiin liittyvät ongelmat voivat aiheuttaa synkronoinnin pysähtymisen pitkäksikin aikaa.

Kun asiakassovellus saa tehtäväkseen ladata tiedoston synkronoinnin aikana, prosessin jatkuminen on riippuvainen tiedoston latauksen valmistumisesta. QueryTask-koodiluokan vastaanottaessa tiedoston lataustehtävän, tiedoston lataus aloitetaan, ja asiakassovelluksen synkronointitapahtuma jää odottamaan latauksen valmistumista. Synkronointi jatkuu vasta sen jälkeen, kun Android-järjestelmän DownloadManager on ladannut tiedoston onnistuneesti tai epäonnistunut siinä riittävän monta kertaa. Laitteen ladatessa suurta tiedostoa huonon verkkoyhteyden välityksellä synkronointi voi pysähtyä jopa kymmeniksi minuutiksi. Prosessin jumittuminen voidaan ohittaa käynnistämällä uusi synkronointi, mutta sen ei pitäisi jäädä jumiin alun perinkään tässä tilanteessa. Samankaltaisia tilanteita synkronoinnin pysähtymisestä löytyisi todennäköisesti asiakassovelluksesta enemmänkin, jos kaikki mahdolliset tilanteet tutkittaisiin läpi tarkasti.

Monimutkainen, osiensa toiminnoista riippuvainen synkronointitapahtuma ei itsessään ole valtava sovelluksen kokonaistoiminnallisuutta lamauttava ongelma. Monimutkaisuus kuitenkin ehdottomasti hankaloittaa ylläpito- ja virheenkorjaustyötä, yksittäisten tapahtumaketjujen selvittämisen viedessä paljon aikaa.

Ylimääräinen työ

Kun synkronointitapahtuma käynnistetään asiakassovelluksessa, se käy läpi kaikki mahdolliset laitteella suoritetut toimenpiteet, joiden tilat vaativat raportoimista StatusTasks-koodiluokkien avulla. Kaikki nämä kuusi tilanraportoinnin koodiluokkaa, jotka tekevät tilojen tarkistuksia, avaavat uuden yhteyden asiakassovelluksen tietokantaan ja hakevat näitä toimenpiteitä. Esimerkiksi FileDeploymentStatusTask tarkistaa, löytyykö tietokannasta merkintä onnistuneesti

ladatusta tiedostosta, jonka latauksen suorittamisesta ei ole vielä ilmoitettu palvelimelle.

Normaalisti synkronointien yhteydessä kaikkien suoritettujen toimenpiteiden tilojen tarkistaminen on haluttua. Asiakassovellus kuitenkin käynnistää kokonaisen synkronointitapahtuman silloin, kun se on suorittanut jonkin meneillään olevan toiminnon. Esimerkiksi kun uusi sovellus saadaan asennettua ja toimenpiteen tila pitäisi päivittää palvelimelle, asiakassovellus käynnistää saman synkronointitapahtuman kokonaan uudestaan.

Tässä tapauksessa riittäisi, että käynnistetään vain sovelluksien tiloja käsittelevä koodiluokka, ApplicationDeploymentStatusTask, ilman synkronoinnin muita StatusTask-koodiluokkia. Esimerkiksi ladattujen tiedostojen tiloja ei tässä tilanteessa tarvitse tarkistaa, vain sovelluksien tilojen päivitys on tarpeellista.

Tätä toiminnallisuutta voidaan perustella sillä, että laitteelta voi mahdollisesti löytyä muita suoritettuja toimenpiteitä, esimerkiksi tiedostojen lataustehtäviä, joiden tilat vaatisivat palvelimelle lähetystä. Normaalitapauksissa raportoimattomia tiloja ei kuitenkaan löydy. Raportoimaton tila voi jäädä laitteelle vain poikkeustilanteissa, joissa laitteen yhteys palvelimeen ei ole syystä tai toisesta toiminut.

Asiakassovellus käynnistää synkronoinnin itsestään uudelleen useassa kymmenessä eri tilanteessa, joissa yksittäisen tilapäivityksen koodiluokan suorittaminen olisi riittävää. Optimoinnin näkökulmasta tämä olisi mahdollista muuttaa muun muassa tarpeettomien tietokantayhteyksien ja -kyselyjen poistamiseksi, koska näissä tapauksissa tiedetään muiden kyselyjen olevan tarpeettomia.

HTTP-pyyntöjen jono

Asiakassovellus lähettää HTTP-pyynnöt synkronointitapahtuman aikana peräkkäin toinen toisensa jälkeen. Asiakassovellus ei esimerkiksi voi hakea uusia tehtäviä tai konfiguraatioita kuitatessaan toisen, jo suoritetun, toimenpiteen tilaa ~~tai~~ eikä se voi päivittää samanaikaisesti kahden suoritetun toimenpiteen tilaa.

Raportoimattomien toimenpiteiden tilat lähetetään synkronoinnin aikana yksi kerrallaan, kuten 3. esimerkkikoodista voidaan nähdä.

Sallimalla asiakassovelluksen HTTP-pyyntöjen lähetyksen rinnakkain laitteen ei tarvitsisi odottaa jokaisesta pyynnöstä vastausta ennen synkronoinnin jatkamista. Synkronointitapahtuma nopeutuisi tällöin huomattavasti laitteella, kun useita prosesseja suoritettaisiin samanaikaisesti.

Tulee kuitenkin pitää mielessä, että palvelimen kuormittaminen useilla samanaikaisilla pyynnöillä mahdollisesti vaikuttaisi negatiivisesti suorituskykyyn riippuen palvelimen kokoonpanosta ja yhdistävien laitteiden määrästä.

HTTP-pyyntöjen määrä

Asiakassovellus lähettää useita pyyntöjä yksittäisen synkronointitapahtuman aikana riippuen laitteen tilasta ja palvelimella olevista uusista toimenpiteistä. Monien pyyntöjen lähettämisen ja käsittelyn takia syntyvää mahdollisesti ylimääräistä kuormitusta olisi mahdollista vähentää, jos kommunikointi tapahtuisi pienemmällä määrällä HTTP-pyyntöjä. HTTP-protokollan kautta kuormitusta lisää muun muassa protokollan tarvitsemat otsakkeet, jotka liitetään jokaiseen pyyntöön, mikä kasvattaa lähetettävän datan määrää.

Otsakkeet sisältävät laitteen ja asiakassovelluksen tunnisteita, versiotietoja sekä lähetetyn viestin tyypin. Palvelin käsittelee vastaanotetun viestin sen tyypin mukaisesti. Esimerkkikoodin 4 otsakkeissa esiintyy uusien tehtävien ja konfiguraatioiden hakemiseen käytettävän viestin otsakkeet, joita synkronointitapahtuman QueryTask lähettää. Palvelin tunnistaa viestin MessageType-otsakkeen Query-arvosta ja palauttaa laitteelle uusia toimenpiteitä, mikäli niitä löytyy.

```
Android-AuthSignature = eqloFb979j7xytIlaD9t1mXB5Ma5BXr/eZlBDDZy/2Q=
Android-AuthID = 232516f5-f0d0-43ca-bd51-4a9d4ba2bfee
Android-ClientGUID = 989d5250-5adc-4e03-8a48-a474554b3acd
Android-GCMRegistrationID = 7kLZp3y9QzvJmNFO4rTxW1:BPB82cMv-
bnvbpFsN1TiuRLHHjJx4s9RwNgO0rHd99ymrUKbuhdK4czrQJxU3ezaizBzJkEj1eY4Yi-
m5SArJ-qbZPZJIFPI04s4USkUJj7NP7Fty8fF7QqXQW2QV4V7J7FOTR3bRck
Android-CompanyName = ZXhhbXBsZQ==
Android-LocalIP = fe80::d7a8:6645:d1d8:d4f3
Android-RequestID = e7b8a6d4-3f4b-4c2e-9a6d-1f2b3c4d5e6f
Android-MessageType = Query
Android-ClientVersion = 1.0.0
Android-ClientBuild = 100
Android-IMEI = 490154203237518,356938035643809
Android-ID = 9774d56d682e549c
Android-SerialNumber = R58M25Y0X6J
Android-DeviceOwnerType = DeviceOwner
Content-Type: application/xml
User-Agent: MDOnline/1.0.0 (Android 14; SM-G991B)
```

Esimerkkikoodi 4. Tyypillisen asiakassovelluksen HTTP-pyynnön otsakkeet.

Jokaisen HTTP-pyynnön yhteydessä lähetetään paljon tietoja yhteyttä ottavasta laitteesta otsakkeiden kautta. Luvussa 4.2 esitellyn uuden sovelluksen asennustehtävän aikana otsakkeiden sisältämät tiedot lähetettiin palvelimelle neljä kertaa alle 30 sekunnin aikana. Kymmenen konfiguraation tilan päivittäminen palvelimelle tapahtuu jokainen omalla pyynnöllään toinen toisensa perään ja jokainen pyyntö tarvitsee saman otsakkeen.

Tyypillisen Android-asiakassovelluksen lähettämän HTTP-pyynnön otsakkeiden kokonaiskoko on noin 780 tavua. Esimerkiksi uuden sovelluksen asennuksen esimerkin yhteydessä tiedusteltiin uusia toimenpiteitä kolmesti sekä raportoitiin sovelluksen asennuksen tila kerran sen valmistuttua (kuva 2). Uusien toimenpiteiden hakemiseen käytetyn viestin sisältö näkyy seuraavassa koodiesimerkissä 5.

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>
<Message>
  <PasswordToken>
     <Active><![CDATA[1]]></Active>
  </PasswordToken>
</Message>
```

Esimerkkikoodi 5. Android-sovelluksen lähettämän Query-tyyppisen viestin sisältö.

Viesti sisältää tiedon siitä, onko laitteen salasanan nollaamiseen käytettävä koodi aktiivinen ja valmis käytettäväksi. Query-viestin sisällön koko on 136 tavua, joka jää reilusti alle otsakkeen muodostaman datan määrän. Sisällön huomioon ottaen otsakkeiden muodostama datamäärä viestin kokonaiskoosta on suhteellisen suuri. Tiedusteluviestien lisäksi uuden sovelluksen asentamisen esimerkkitehtävässä lähetetään sovelluksen asentumisen tilapäivitys, jonka sisältö käy ilmi koodiesimerkistä 6.

```
<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>

<Message>
  <ApplicationDeployment>
    <DeploymentID><![CDATA[359]]></DeploymentID>
    <DeploymentType><![CDATA[1]]></DeploymentType>
    <EndTime><![CDATA[1723665323]]></EndTime>
    <Error>
      <Code><![CDATA[0]]></Code>
      <Description />
    </Error>
    <Status><![CDATA[3]]></Status>
  </ApplicationDeployment>
</Message>
```

Esimerkkikoodi 6. Sovelluksen asentumisen jälkeen lähetetty viesti.

Sovelluksen asennuksen tilapäivitysviesti on kooltaan 341 tavua. Asennusesimerkin aikana lähetettyjen HTTP-pyyntöjen varsinaisen viestisisällön kokonaiskoko on siis suunnilleen yhtä suuri kuin yhden HTTP-pyynnön otsakkeiden koko.

HTTP-protokollaa käyttäessä jokaisen pyynnön lähettämistä edeltää TCP-yhteyden (15) luonti, joka varmistaa luotettavan tiedonsiirron asiakassovelluksen ja palvelimen välillä. TCP-yhteys luodaan kolmitiekättelyllä ennen HTTP-pyynnön lähettämistä ja suljetaan nelitiekättelyllä tiedonsiirron päätyttyä. Myös näiden yhteyksien luonnin aiheuttamaa työtä prosessissa olisi mahdollista vähentää pyyntöjen määrää laskiessa.

Palvelin autentikoi jokaisen pyynnön erikseen HTTP:n tilattoman luonteen takia (9). Autentikoinnin jälkeen laitteen lähettämät otsakkeet tarkistetaan ja niiden sisältämä tieto tallennetaan tietokantaan muutosten varalta. Ennen varsinaisen pyynnön käsittelemistä myös tarkistetaan, ettei laitetta esimerkiksi ole poistettu laitehallinnasta tai ettei sitä ole jäädytetty ongelmatilanteiden takia.

Jokaisen HTTP-pyynnön yhteydessä asiakassovellus avaa yhteyden palvelimeen ja lähettää otsakkeiden mukana samat tiedot yhä uudelleen. Palvelin taas autentikoi pyynnön ja tekee erinäisiä tarvittavia tarkastuksia. Joissakin tilanteissa asiakassovellus lähettää useita pyyntöjä palvelimelle, joiden sisältö voitaisiin hyvin lähettää yksittäisen viestin sisällä. Muokkaamalla HTTP-pyyntöjen rakennetta ja sisältöä päästäisiin tästä ylimääräisestä käsittelystä eroon pyyntöjä vähentämällä.

StatusTasks-erehdys

Asiakassovelluksen StatusTask-koodiluokkia tarkasteltaessa niiden huomataan toimivan pääpiirteittäin samalla tavalla toistensa kanssa. Niiden toiminta noudattaa seuraavanlaista kaavaa: Luokan suoritettava metodi lukee asiakassovelluksen tietokantaa, etsii raportoimattomia toimenpiteiden tiloja ja lähettää löydetyt tilat yksitellen palvelimelle. Tämä toiminnallisuus on aiemmin esitelty ApplicationDeploymentStatusTask-koodiluokan otteessa koodiesimerkissä 3.

Toiminnallisuuden nykytoteutus aiheuttaa tilanteita, joissa asiakassovellus raportoi useita suoritettujen toimenpiteiden tiloja käyttäen useita HTTP-pyyntöjä. Saman toimenpiteen koodiluokan kaikkien toimenpiteiden raportoimattomat tilat voitaisiin olettaa kulkeutuvan palvelimelle yksittäisen pyynnön avulla.

Palvelimen koodi onkin toteutettu niin, että se kykenisi käsittelemään useita tiloja yksittäisen pyynnön sisällä jo nyt useissa tapauksissa. Kun tutkitaan toimenpiteiden tilojen päivityksiä käsittelevien luokkien metodeja palvelimella, voidaan huomata neljän luokan kuudesta käsittelevän useamman tilapäivityksen yksittäisen pyynnön sisällä, jos niitä pyyntöön sisältyy. Seuraava koodiesimerkki

7 on ote palvelimen metodista, joka käsittelee sovellusten asennustehtävien tilojen päivityksiä, josta toiminnallisuus voidaan havaita.

```
public async Task<ServerMessage> ExecuteAsync(AndroidClientMessage
message)
{
  AppDeploymentStatusMessage request = (AppDeploymentStatusMes-
sage)message;

  List<AppDeploymentStatusMessage.AppDeploymentStatus> deployments =
request.Deployments;

  _logger.Debug($"Updating status of {deployments.Count} application
deployments (client GUID {request.ClientGUID})");

    foreach (AppDeploymentStatusMessage.AppDeploymentStatus deployment
in deployments)
    {
        ...
    }

    return new AcknowledgeMessage();
}
```

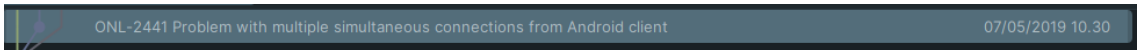
Esimerkkikoodi 7. Ote palvelimen AppDeploymentStatusHandler-luokan koodista.

Esimerkkikoodin metodin toiselta koodiriviltä voidaan nähdä palvelimen poimivan viestistä listan AppDeploymentStatus-tilapäivityksiä, jotka tämän jälkeen käsitellään foreach-silmukassa. Todellisuudessa asiakassovellus lähettää kuitenkin vain yhden tilan pyyntöä kohden, joten tämä käsittely jää toistaiseksi hyödyntämättä.

Tämä viestien vääränlainen käsittely voidaan vahvistaa tarkastelemalla tuotannossa toimivan palvelimen lokiviestejä. Kaikki lokiviestit, jotka koodiesimerkistä 7 nähtävä _logger.Debug-metodikutsu tallentaa, raportoivat yksittäisen statuksen päivityksen kerrallaan. Muut palvelimen luokat, jotka käsittelevät tilapäivityksiä, käyttäytyvät samoin.

Koska palvelin on toteutettu käsittelemään useita tiloja melkein kaikkien tilapäivityksien osalta, toiminnallisuuden tulisi olla helposti korjattavissa vain asiakassovelluksen vaatiessa työtä. Virheellinen asiakassovelluksen toiminta on ollut pitkään käytössä huomaamatta, sillä Git-ohjelmistoversiointityökalusta voidaan

nähdä nykyisen koodin olleen osana sovellusta vuodesta 2019 lähtien, kuten kuvasta 4 käy ilmi (16).



Kuva 4. Git-versiohallinnassa nähtävä commit-komento, jolloin asiakassovelluksen yksittäisten tilaviestien lähetys on toteutettu.

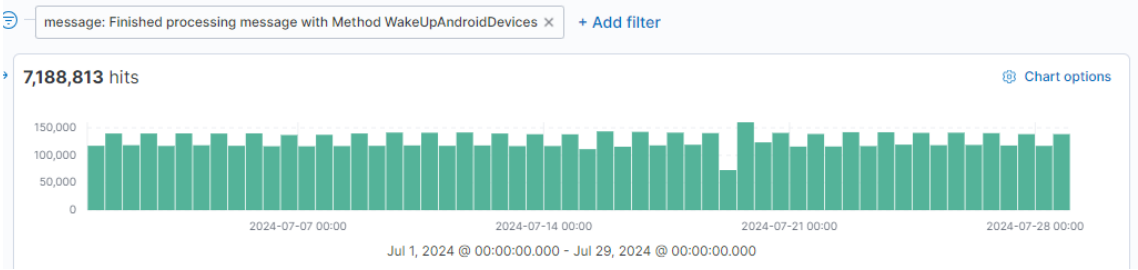
NotificationService

Heräteviestien ollessa keskeisessä osassa laitteen ja palvelimen välisessä kommunikaatiossa näistä vastuussa olevan NotificationService-komponentin toimintaa on syytä tarkastella syvemmin. Komponentti toimii erillään palvelimesta, jonka kanssa laitteet kommunikoivat, ja komponentti käsittelee heräteviestien lähettämisen ulkoisiin palveluihin. NotificationService käsittelee kaikkien Miradoren laitehallintaan rekisteröityjen laitteiden herättämisen alustasta riippumatta. NotificationService on siis osallisena Android-alustan viestien käsittelyn lisäksi myös muissa ohjelmistopalvelun osien toteutuksissa.

Palvelin tallentaa NotificationService:n käsiteltäväksi haluttavat viestit Azuren Queue Storage -jonoon (17). Viestien käsittely jatkuu Azure Functions -ympäristössä sijaitsevassa NotificationService-komponentissa, joka lukee viestit jonosta asynkronisesti. Tällä toteutustavalla palvelin voi jatkaa toimintojaan viestin jonoon kirjoittamisen jälkeen. Sen ei tarvitse jäädä odottamaan vastauksia HTTP-pyyntöihin kolmannen osapuolen palveluista, NoficationServicen käsitellessä kommunikoinnin.

NotificationService hyödyntää Azure Functions -palvelun palvelitonta (Serverless) infrastruktuuria. Azure luo automaattisesti lisää NotificationServicen instansseja prosessin kuormituksen kasvaessa. Instansseja myös vähennetään automaattisesti prosessoinnin tarpeen laskiessa, joka mahdollistaa viestien tehokkaan käsittelyn viestien määrän muuttuessa päivän aikana.

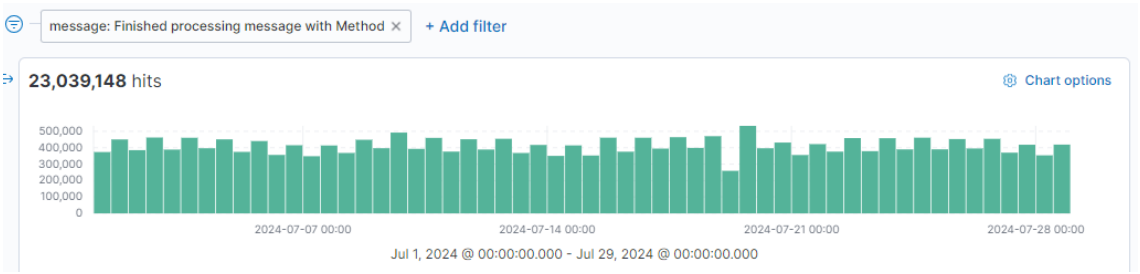
Android-käyttöjärjestelmän osalta NotificationService käsittelee laitteiden herätyksiä sekä Managed Google Play-sovelluksia ja niiden konfiguraatioita. Jonoon lisättyjä Android-herätteviestejä käsitellään tuotannossa yli 7 miljoonaa kuukaudessa, keskimäärin noin 10 000 jokaisen tunnin aikana. Kuvasta 5 nähdään Miradoren lokipalvelusta prosessoitujen herätteviestien määrä vuoden 2024 heinäkuun neljän ensimmäisen viikon ajalta.



Kuva 5. NotificationService:n käsittelemät Android-herätteviestit ajalta 1.7.2024–9.7.2024.

Tarkasteltaessa funktion lokitietoja keskimääräinen aika yksittäisen herätteviestin lähettämiseen jää reilusti alle 100 millisekunnin. Yksittäinen jonossa oleva heräteviesti sisältää enimmillään 60 laitetta kerrallaan. Jos käsitelty viesti sisältää useita herätettäviä laitteita kerrallaan, suoritus kestää kauemmin riippuen laitteiden määrästä. Tämä johtuu siitä, että FCM-palvelun kanssa kommunikoiva käsittelijä, AndroidNotificationRequestHandler, lähettää viestit yksi kerrallaan eteenpäin FCM-palvelulle. Käsittely voitaisiin muuttaa lähettämään viestejä samanaikaisesti, mikä nopeuttaisi suorittamista useiden laitteiden herättämisen yhteydessä.

NotificationService käsittelee lukuisia viestejä vaihteleviin tarkoituksiin. Android-herätteviestit muodostavat kuitenkin merkittävän osan kaikista käsitellyistä viesteistä, noin 30 % kaikesta palvelun liikenteestä. Kuvasta 6 nähdään NotificationService:n käsitelleen saman heinäkuun aikana kaiken kaikkiaan noin 23 miljoonaa viestiä.



Kuva 6. NotificationService:n käsittelemät viestit ajalta 1.7.2024–29.7.2024

Oman osuutensa NotificationService-komponentin viestinnästä muodostavat myös sen käsittelemät Managed Google Play -viestit. Liitteen 1 kuvista nähdään kaikkien näiden viestien liikenne heinäkuun neljältä viikolta. Yhteensä Managed Google Play -viestejä käsiteltiin tällä aikavälillä noin kaksi miljoonaa. Näiden viestien käsittely on toteutettu Googlen tarjoaman EMM API-kirjaston avulla (18). Managed Google Play -viestit lähetetään suurissa erissä useiden yksittäisten viestien sijaan ja näin ollen tehokkaammin verrattuna heräteviesteihin.

Yksittäisten prosessoitujen Android-viestien rakenne on kuitenkin optimaalinen, sillä ne sisältävät vain tarvittavavan tiedon viestin toiminnon suorittamiseksi. Muiden viestityyppien käsittelyä voitaisiin tulevaisuudessa tarkastella syvemmin, sillä niiden käsittelyn mahdolliset ongelmat kasvattavat jonon kaikkien viestien käsittelyn aikaa. Hitaasti toimiessaan NotificationService heikentäisi koko Miradoren laitekäsittelyn nopeutta ja vaikuttaen negatiivisesti koko ohjelmiston käyttöön.

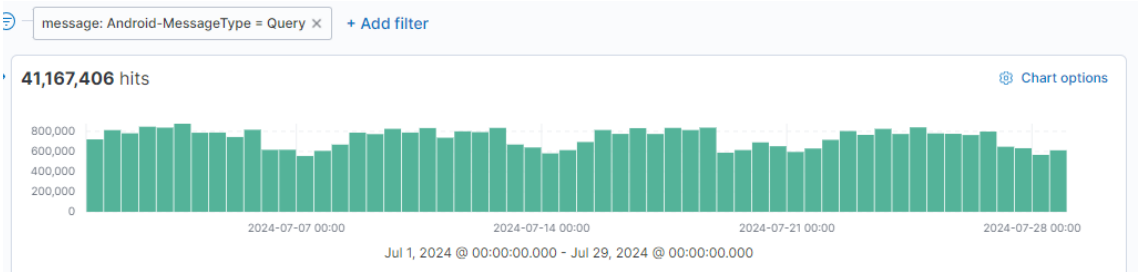
Kyselyviestit

Uusien tehtävien hakemiseen käytetty Query-tyyppinen kyselyviesti lähetetään jokaisen synkronointitapahtuman yhteydessä vähintään kerran riippuen laitteelle asetetuista tehtävistä ja konfiguraatioista. Oletusarvoinen asiakassovelluksen maksimiaika synkronointitapahtumien välillä on 3 tuntia, joten yleisimmissä tapauksissa jokaisella laitteella lähetetään useita synkronointeja päivän aikana.

Synkronoinnin kyselyviesti on varsin pieni, sillä laitteen ei tarvitse lähettää muuta tietoa kuin Query-viestin tunniste. Joissakin tapauksissa asiakassovellus lisää tähän kyselyviestiin ylimääräisen, kertaluontoisen HTTP-otsakkeen pyytääkseen lisäkonfigurointiarvoja palvelimelta. Esimerkiksi Samsung Knox -ominaisuuksia tukeva laite tulee rekisteröidä koodilla Samsungin tarjoaman laitteen ohjelmointirajapinnan kautta (19). Koodi haetaan kyselyviestin yhteydessä vain kerran, jota käyttäen laitteen Knox-ominaisuudet aktivoidaan. Aktivoinnin jälkeen koodia ei tarvitse enää lähettää.

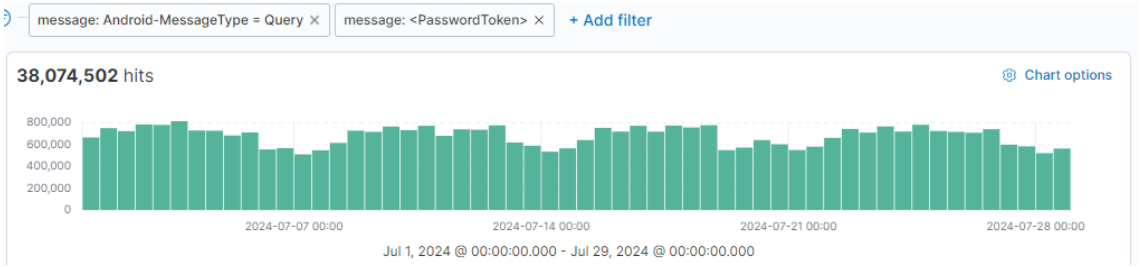
Lähtökohtaisesti asiakassovelluksen kyselyviestit ovat optimaalisia käsittäen vain tarvittavan tiedon, yhtä poikkeusta lukuun ottamatta. Asiakassovellus luo ensimmäisen synkronointitapahtuman aikana laitekohtaisen satunnaisen salasanatunnuksen. Tämä tunnus lähetetään palvelimelle ja aktivoidaan laitteessa, kun palvelin on vahvistanut sen vastaanottamisen. Tunnusta tarvitaan aktivoimisen jälkeen laitteen pääsykoodia nollatessa etäyhteydellä, lisäturvallisuuden saavuttamiseksi. Tunnusta ei vaihdeta aktivoimisen jälkeen laitteella, paitsi erittäin harvinaisissa virhetilanteissa. Tunnuksen aktiivisuuden tila kuitenkin ilmoitetaan jatkuvasti palvelimelle jokaisen kyselyviestin yhteydessä vahvistetun vastaanottamisen jälkeenkin. Synkronointimäärän ollessa suuri tästä kertyy paljon turhaa dataliikennettä.

Vuoden 2024 heinäkuun ensimmäisten neljän viikon aikana Miradoreen rekisteröityneet Android-laitteet lähettivät yli 41 miljoonaa kyselyviestiä synkronointien yhteydessä, kuten kuvasta 7 käy ilmi.



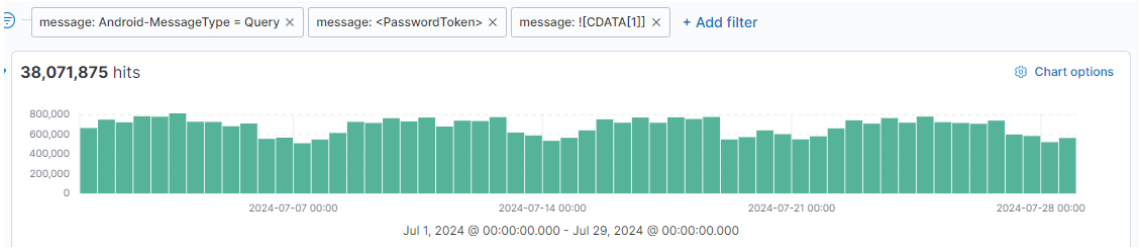
Kuva 7. Vastaanotettujen Query-tyyppisten viestien määrä 1.7.2024–29.7.2024.

Kaikki laitteet eivät kuitenkaan lähetä salasanatunnuksen tilaa, sillä ominaisuus erillisen tunnuksen tarvitsemiseen salasanan nollaamiseksi lisättiin Android-alustaan vasta versiossa 8.0 (20). Toisin sanoen vanhemmat laitteet eivät tunnusta lähetä. Kuvasta 8 voidaan näin ollen havaita, että kyselyviestien määrä, jotka sisältävät tunnuksen tilan, on hieman pienempi kuin kyselyviestien kokonaismäärä. Tarkasteluaikavälillä noin 38 miljoonaa viestiä lähetetystä 41 miljoonasta sisälsi salasanatunnuksen tilan.



Kuva 8. Vastaanotettujen Query-tyyppisten viestien määrä, jotka sisältävät salasanatunnuksen statuksen 1.7.2024–29.7.2024.

Aiemmin esitellyssä koodiesimerkissä 5 nähdään statuksen päivittämiseen käytetty viesti, joka Query-viesteissä lähetetään. Viestin sisällölle kokoa kertyy 136 tavua. Lähes jokainen nähtävissä oleva tunnuksen tila on lähetetty turhaan, sillä rekisteröinnin jälkeen status on lähtökohtaisesti aina aktiivinen. Tämä käy ilmi myös sisällyttämällä edelliseen hakuun vain aktiivisen tunnuksen raportoineet kyselyt, jolloin viestien määrä putoaa alle kolmella tuhannella (kuva 9).



Kuva 9. Vastaanotettujen Query-tyyppisten viestien määrä, jotka sisältävät aktiivisen salasanatunnuksen statuksen 1.7.2024–29.7.2024.

Kahden tuhannen viestin ero johtuu uusien laitteiden lähettämästä tunnuksen inaktiivisesta statuksesta ensimmäisten kyselyiden yhteydessä. Nämä ovat pyyntöjä, jolloin tunnuksen lähetystä oikeasti tarvittiin. Ylimääräisen aktiivisen tilan raportoinnin poistamisella voitaisiin dataliikennettä vähentää toteutuneiden viestimäärien perusteella yli viiden gigatavun verran kuukaudessa.

Toteutuksen kokonaiskuva

Suurimmaksi osaksi kommunikointi laitteen ja palvelimen välillä on optimaalista ja sen rakenne toimiva käyttötarkoitusta ajatellen. Käyttämällä Polling-metodia ja heräteviestejä saadaan aikaan yhdistelmä, jolla laitteet pidetään ajan tasalla tehokkaasti.

Turhaa sisältöä viesteissä on pyritty minimoimaan, ja lähtökohtaisesti palvelin sekä sovellus molemmat pitävät viestit minimaalisina. Komponentin tarvitessa tietoa toiselta HTTP-pyynnön otsakkeisiin lisätään kenttä osoittamaan tiedon tarvetta synkronoinnin yhteydessä. Kommunikoinnissa on kuitenkin jonkin verran ylimääräistä tiedonsiirtoa, jota olisi mahdollista vähentää kuten edellä esiteltiin.

Mahdolliset viestinnän virhetilanteet käsitellään sulavasti. Esimerkiksi asiakassovelluksen kohdatessa virheen synkronoinnin yhteydessä, synkronointia yritetään uudelleen perääntyvästi kuusi kertaa kasvavalla viiveellä. Viive virheen sattuessa on viisi sekuntia ja viimeisellä yrityskerralla kymmenen minuuttia.

6 Toteutuksen parantamisen vaihtoehtoja

Luvussa 5 esiteltyihin kommunikaation kehityskohteisiin tarjotaan seuraavaksi parannusehdotuksia. Kehityskohteet voidaan luokitella kahteen kategoriaan: pienimuotoisiin korjauksiin sekä suurempiin kokonaisuuksiin, riippuen muutosten vaativuudesta ja laajuudesta. Ensimmäisen kategorian korjausehdotusten toteuttaminen kestäisi arviolta muutamasta henkilötyöpäivästä henkilötyöviikkoon tai -kahteen. Jokaiseen löydettyyn mahdolliseen parannuskohteeseen ei

kuitenkaan ole selkeää tai helppoa ratkaisua, eikä arvioita näiden muutosten kestosta tai täydestä laajuudesta voi antaa ilman syvällistä selvitystä aiheesta. Esitellyt haastavammat parannuskohteet lukeutuvat jälkimmäiseen suurempien muutoksien kategoriaan, joita käsitellään toisessa alaluvussa erikseen.

6.1 Pienimuotoiset korjaukset

Asiakassovelluksen osasynkronoinnit

Synkronointitapahtumat aloitetaan nykyään laitteella myös tapauksissa, joissa yksittäisen suoritetun toimenpiteen lopputulos tulisi päivittää palvelimelle. Kuten luvussa 5 esiteltiin, tämän takia asiakassovellus tekee jonkin verran turhaa työtä. Kokonaisten synkronointitapahtumien käynnistämisen sijaan tilanteittain tehokkaampi vaihtoehto olisi suorittaa nykyisiä yksittäisiä synkronoinnin osia tarpeen mukaan.

Muutoksen voisi tehdä useammalla eri tavalla. Yksinkertaisin toteutustapa olisi lisätä uusi metodi Utils-luokkaan, joka nykyään käynnistää synkronointitapahtumat. Metodi ottaisi parametrina halutun tilanraportointityypin lisäten sitä vastaavan luokan suoritettavaksi ITaskQueue-jonoon (koodiesimerkki 2). Uuden metodin käynnistämä tilanraportointi-koodiluokka päivittäisi raportoimattomat tilat palvelimelle, ja muut synkronoinnin muodostavat koodiluokkien instanssit voitaisiin jättää suorittamatta, välttyen turhalta työltä. Tarkastellaan esimerkiksi luvussa 4.2 esiteltyä uuden sovelluksen asennustehtävää, jonka aikana suoritetaan kolme synkronointia. Viimeinen synkronointi olisi mahdollista korvata suorittamalla pelkästään ApplicationStatusDeploymentTask sovelluksen asentumisen jälkeen.

Tätä muutosta olisi mahdollista käyttää pienentämään asiakassovelluksen työmäärää yli 30 eri tilanteessa, jossa koko synkronointitapahtuma käynnistetään, yksittäisen osan suorituksen ollessa riittävä. Tällä vältyttäisiin myös ylimääräiseltä Query-viestien lähettämiseltä, mitä palvelimen ei tarvitsisi käsitellä. Joissakin tilanteissa Query-viestin lähettäminen tilapäivityksen jälkeen on kuitenkin

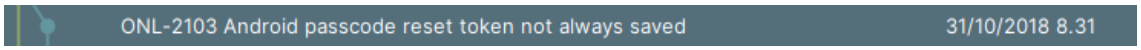
oleellinen osa synkronoinnin loppuun viemistä, joten muutokset tulisi varmistaa ja testata läpikotaisin.

Osasynkronointien tuomat hyödyt todennäköisesti olisivat kuitenkin melko minimaalisia, ja niiden vaikutusta olisi haastavaa mitata. Puhtaasti optimoinnin näkökulmasta muutos olisi kuitenkin positiivinen, ja toteutuksen ollessa nopea, voisi kehitystä ainakin harkita. Kuvatunlaisen käsittelyn toteuttaminen sovelluksien tilojen käsittelyyn saataisiin toteutettua muutaman henkilötyöpäivän aikana.

Query-viestin korjaus

Query-viestien korjaus, ylimääräisen datan lähetyksen eliminointi, voidaan luokitella nopeasti korjattavaksi. Asiakassovellus lähettää näiden viestien yhteydessä salasanatunnuksen tilan palvelimelle tarpeettomasti yhä uudelleen ja uudelleen riippumatta sen tallentamisen tilasta palvelimella.

Toiminnallisuus salasanatunnuksen tilan jatkuvaan lähettämiseen on lisätty vuonna 2018. Tämä on tehty korjauksena ongelmaan, jossa palvelin ei ollut tallentanut asiakassovelluksen lähettämää tunnuksen tilaa, kuten kuvasta 10 voidaan nähdä Git-versionhallinnan commit-komennon viestistä.



Kuva 10. Git-versionhallintatyökalussa nähtävä commit-komento, jossa toistuva salasanatunnuksen tilan lähetys lisättiin.

Asiakassovellus ei saa vahvistusta palvelimelta siitä, onko lähetetty tila tallennettu onnistuneesti. Muokkaamalla palvelimen koodia sovellukselle voitaisiin palauttaa tieto tallentamisen onnistumisesta esimerkiksi HTTP-otsakkeen kautta. Tämän vahvistuksen saadessaan asiakassovellus voisi lopettaa salasanatunnuksen tilan lähettämisen, kunnes siihen tulisi jotain muutoksia.

Tämä muutos vaatisi pienen muutoksen palvelimella ja jonkin verran salasanatunnuksen lisäkäsittelyä asiakassovelluksessa. Muutoksen toteuttaminen

ei kuitenkaan kestäisi arviolta muutamaa henkilötyöpäivää pidempään, sillä suurilta muutoksilta vältyttäisiin. Kuten luvussa 5 esiteltiin, tämä korjaus vähentäisi tiedonsiirtoa Query-viestien osalta useamman gigatavun verran kuukaudessa ja hyöty olisi suoraan helposti mitattavissa.

FCM-ohjelmointirajapinnan kommunikaatio

NotificationService-komponenttiin kohdistuva mahdollinen pienimuotoinen optimointi olisi heräteviestien nopeampi lähettäminen FCM-palveluun. Nykyinen toteutus lähettää heräteviestit laite kerrallaan odottaen vastausta palvelusta ennen seuraavan lähettämistä. Tämä toteutus voidaan nähdä seuraavasta koodiesimerkistä 8, jossa metodi odottaa jokaisen herätyspyynnön suoriutumista loppuun ennen seuraavan käsittelyä.

```
public async Task WakeUpAndroidDevicesAsync(List<string> registra-
tionIds, Guid instanceGuid, WakeUpAction action = WakeUpAction.None)
{
     ...
     foreach (string registrationId in registrationIds)
     {
     ...
          if (!await WakeupAsync(registrationId, action))
          {
               ...
          }
     }

     ...
}
```

Esimerkkikoodi 8. Ote FCM-palvelun kanssa keskustelevasta koodista NotificationService-komponentissa.

Vastausta odottaessa FCM-palvelusta ennen seuraavan pyynnön lähettämistä ehtii ylimääräistä odotusaikaa kertyä pahimmillaan useita sekunteja maksimäärää (60) laitteita herättäessä. Jos NotificationService-komponentin annettaisiin käsitellä useita pyyntöjä samanaikaisesti, metodin suorittamisesta tulisi moninkertaisesti nopeampaa. Esimerkiksi kymmenen pyynnön samanaikainen lähettäminen olisi tehokkaampi ratkaisu peräkkäisten HTTP-pyyntöjen sijaan, erityisesti tilanteissa, joissa FCM-palvelu sattuu olemaan hidas vastauksissaan. Jokaisen pyynnön kestäessä esimerkiksi 100 millisekuntia, nykytoteutuksella 10

pyyntöä vie aikaa sekunnin. Jos kaikki kymmenen pyyntöä suoritettaisiin samanaikaisesti, pyyntöjen ajan keston pysyessä samana, aikaa kaikkien pyyntöjen lähetykseen kuluisi yhteensä 100 millisekuntia.

Aiemmin käytettävissä olleen, hiljattain suljetun FCM legacy APIn kautta yksittäisellä HTTP-pyynnöllä oli mahdollista herättää samanaikaisesti tuhat laitetta (21). Vanhan ohjelmointirajapinnan kautta laitteiden kohdistaminen onnistui käyttämällä laitteella luotua Firebase-tunnusta, joita yksittäiseen pyyntöön oli mahdollista listata 1 000 kappaletta. Firebase-tunnukset ovat sovelluskohtaisia tunnuksia, joita käyttämällä viestit kulkeutuvat oikeaan laitteeseen. Firebase-tunnuksiin pohjautuva massaviestintä poistettiin uudessa ohjelmistorajapinnan versiossa, eikä yksittäisiä laitteita voida samalla viestillä herättää käyttämällä tätä laitetunnusta. Uusi ohjelmointirajapinta tarjoaa muita tapoja massaviestien lähettämiseen, mutta nämä eivät tue Miradoren tarpeita (22). Heräteviestit on kohdistettava tiettyihin laitteisiin joka kerralla, eikä uuden rajapinnan tarjoamat, toistaiseksi saatavilla olevat, massaviestinnän keinot ole tähän hyödynnettävissä.

Ottaen huomioon nykyisen määrän viestejä, jotka NotificationService lähettää FCM-palvelulle, viestien samanaikainen lähettäminen ei lähestyisi Googlen asettamia rajoituksia palvelun käytölle. Tarkasteltaessa esimerkkiä Googlen dokumentaatiossa rajapinnan käytöstä, NotificationServicen tuotannossa tunnin aikana käsittelemät 10 000 viestiä voitaisiin lähettää halutessa muutaman sekunnin aikana ilman ongelmia (23).

Edellä kuvailtu samanaikaisten pyyntöjen lähetyksen käsittely voitaisiin toteuttaa muutamassa henkilötyöpäivässä. Tämä vähentäisi palvelimen ja FCM-palvelun kommunikaatioon kuluvaa aikaa moninkertaisesti tapauksissa, joissa herätetään useita Android-laitteita kerralla.

StatusTasks-luokkien korjaus

Luvussa 5 esitelty StatusTasks-koodiluokkien ongelma voitaisiin korjata asiakassovelluksessa noin yhden tai kahden henkilötyöviikon aikana. Koska neljä

kuudesta tilatehtävien käsittelijästä palvelimella tukee jo useiden tilojen lähettämistä yhden pyynnön sisällä, riittäisi StatusTask-koodiluokkien osalta pienen muutoksen tekeminen asiakassovellukseen käsittelyn hyödyntämiseksi.

Kuten koodiesimerkistä 3 nähdään, asiakassovellus käy läpi raportoimista vaativien tehtävien tiloja ja lähettää nämä palvelimelle yksi kerrallaan. Koodia olisi mahdollista muokata niin, että siitä poistettaisiin for-silmukka, joka iteroi jokaisen tilan erikseen. Iteroinnin sijaan jokaisessa StatusTask-luokassa luotaisiin yksittäinen viesti, jonka viestikenttään lisättäisiin kaikki raportoimattomat tilat.

Kaksi palvelimen luokkaa, jotka eivät tällä hetkellä kykene käsittelemään useita tiloja yhdessä pyynnössä, voitaisiin myös muokata käsittelemään kaikki tilat kerralla. Toimintalogiikka voitaisiin kopioida muista palvelimen tiloja käsittelevistä luokista, kuten palvelimen AppDeploymentStatusHandler-luokasta (Esimerkkikoodi 7). Tämän jälkeen vastaava käsittely asiakassovelluksessa voitaisiin tehdä sama muutos kuin muihinkin StatusTask-koodiluokkiin. Tämä muutos vähentäisi edestakaisia pyyntöjä huomattavasti tilanteissa, joissa useita suoritettujen tehtävien tiloja raportoidaan peräkkäin.

6.2 Mahdolliset suuret muutokset

Kuten luvussa 5 tuotiin esille, asiakassovelluksen kommunikaation toiminnassa olisi parantamisen varaa. Pienempimuotoisten korjausten ohella olisi mahdollisesti tarjolla lisää hyötyjä, jos sovelluksen kommunikaation toimintaan tehtäisiin suurempia muutoksia. Suurempien muutosten vaikutusten ja kannattavuuden täydellinen selvittäminen vaatii kuitenkin syvällistä tutkimusta ja laajamuotoista testausta, mikä halutessa voitaisiin suorittaa tämän työn ajatuksiin pohjaten.

Loogisena muutoksena olisi sallia asiakassovelluksen lähettää useita HTTP-pyyntöjä samanaikaisesti. Nykyisellään synkronointitapahtuma keskeytyy, kun palvelimelta odotetaan vastausta yksittäisiin viesteihin. Jos jokainen lähetetty pyyntö ei pysäyttäisi synkronoinnin suorittamista vastauksen saapumiseen asti,

synkronoinnit valmistuisivat laitteella parhaimmillaan monia sekunteja nopeammin laitteen suorittaessa synkronoinnin eri osia rinnakkain.

Asiakassovelluksen kannalta useiden HTTP-viestien lähettämisen salliminen samanaikaisesti ei vaikuta suurelta muutokselta. Kuitenkin kun synkronointiprosessi olettaa koodiluokkien instanssien suoritusta tietyssä järjestyksessä, ongelmia muutoksista voi syntyä. Esimerkiksi laitteen rekisteröityessä Miradoren laitehallintaan ensimmäisenä palvelimelle lähetetään laitteen inventaario. Tähän viestiin kerätään laitteen kaikki tunnisteet, ja palvelimella niiden avulla viimeistellään laitteen rekisteröinti. Synkronointitapahtuman seuraavia osia ei voida suorittaa, jos tätä laitteen rekisteröintiä ei ole viimeistelty.

Huomioon muutosta tehdessä täytyisi ottaa synkronoinnin osien itsensä uudelleenkäynnistämisen käsittely. Esimerkiksi, jos viisi koodiluokkaa haluaisi käynnistää synkronoinnin uudelleen samanaikaisesti, nykyisellä toteutuksella asiakassovellus tekisi turhaa työtä, jos synkronoinnin eri osat suoritettaisiin yhtä aikaa. Tämä mahdollisesti aiheuttaisi myös muita ongelmia.

Jos useiden pyyntöjen samanaikaisen lähettäminen toteutettaisiin, palvelimen käsittelemä HTTP-pyyntöjen kokonaismäärä pysyisi samana. Uutena ongelmana voisi mahdollisesti ilmetä palvelimen kyvyttömyys käsitellä muuttunutta lähetystiheyttä. Nyt pyynnöt ovat tulleet asiakassovellukselta yksi kerrallaan. Uudella toteutuksella palvelimelle saapuisi mahdollisesti samanaikaisesti moninkertainen määrä HTTP-pyyntöjä aikaisempaan verrattuna. Tällöin palvelimelle varatut resurssit eivät välttämättä riittäisi prosessoimaan viestejä, ja sen toimintakyky voisi heiketä.

Ottaen huomioon muutosten vaativan paljon työtä sekä niiden mahdollisesti synnyttämät haitat vaikuttaa samanaikaisten pyyntöjen salliminen epäsuotuisalta. Sen sijaan HTTP-pyyntöjen minimointi olisi mahdollisesti parempi synkronoinnin optimointiratkaisu. Tällöin palvelimen kokonaispyyntömäärä ja luvussa 5 esitellyn HTTP-protokollan kuormituksen vaikutus vähenisivät.

HTTP-pyyntöjen minimointi edellyttäisi merkittäviä muutoksia sekä asiakassovellukseen että palvelimen komponentteihin. Tiivistetysti sovelluksen nykyinen tapa käsitellä HTTP-pyyntöjä synkronointien aikana tulisi uudistaa siten, että niiden sisältö yhdistetään suuremmiksi yksittäisiksi pyynnöiksi mahdollisuuksien mukaan. Esimerkiksi aiemmin esitellyn virheenkorjauksen sijaan StatusTask-luokissa synkronointitapahtuman kaikkien osien keräämä tieto voitaisiin yhdistää yhdeksi isoksi pyynnöksi, jolla raportoidaan kaikkien eri osa-alueiden tehtävien tilat. Suurien pyyntöjen tukeminen vaatisi myös palvelimen käsittelyn muutoksia, kun kaiken oletetaan nykyään toimivan yksittäisten pyyntöjen kautta. Muutoksen jälkeen synkronoinnin HTTP-pyyntöjen määrä vähenisi merkittävästi, sillä nykyisin jokaisesta laitteella suoritetusta yksittäisestä tehtävästä lähetetään oma pyyntönsä.

Myös uusien toimenpiteiden haku palvelimelta laitteelle voitaisiin toteuttaa yksittäisellä HTTP-pyynnöllä. Tällä hetkellä erilaiset tehtävät ja konfiguraatiot lukeutuvat kahdeksaan kategoriaan. Esimerkiksi uusien sovelluksen asennukset ja laitteen toiminnallisuuksien konfiguroinnit ovat kumpikin omia kategorioitaan. Asiakassovellus lähettää kyselyviestin palvelimelle, jolloin palvelin palauttaa yhden kategorian kaikki toimenpiteet, jotka laitteelle on asetettu. Esimerkiksi kaikki asennettavat sovellukset palautetaan laitteelle kerralla. Saadakseen seuraavan kategorian toimenpiteet sovellus lähettää uuden pyynnön. Pyyntöjen lähetys toistuu, kunnes kaikki uudet toimenpiteet ovat laitteella.

Palvelin voisi kategoria kerrallaan toteutuksen sijaan palauttaa kaikki jonossa olevat uudet tehtävät ja konfiguraatiot yhden vastauksen sisällä. Käsittelyn uusiminen vaatisi melkoisesti muutoksia niin asiakassovellukseen kuin palvelimeenkin synkronoinnin siirtyessä toimimaan useiden pienien viestien sijaan yksittäisten suurien pyyntöjen avulla.

Asiakassovelluksen keskeisimmän osan, synkronointitapahtumien, muuttuessa uusilta vaikeuksilta olisi vaikea välttyä. Sovelluksen toiminta rakentuu vahvasti nykyisen synkronoinnin toteutuksen ympärille, joten laajoja uudistuksia sovelluksen komponentteihin olisi toteutettava.

Esitellyt HTTP-pyyntöjen muutokset voivat teoreettisesti olla tehokkaampi ratkaisu synkronoinnin kannalta. Synkronoinnit tapahtuisivat pienemmällä pyyntömäärällä verrattuna nykyiseen vähentäen jokaisen HTTP-pyynnön mukana tulevaa kuormitusta. Kuitenkin kun huomioidaan muutosten vaatima työmäärä ja saatavat konkreettiset hyödyt, synkronoinnin suuret muutokset eivät toistaiseksi vaikuta houkuttelevilta.

Loppukäyttäjälle näkyvä hyöty saataisiin synkronointien suoriutuessa nopeammin. Synkronoinnin suoritusaika nykyään ei kuitenkaan ole pitkä, yleensä synkronoinnit kestävät muutamia sekunteja laitteen verkkoyhteyden ollessa hyvä. Suurin osa synkronoinneista tapahtuu lisäksi laitteella käyttäjän huomaamatta, joten suurin osa asiakaskunnasta ei välttämättä huomaisi juurikaan eroa aiempaan.

Komponentteja uudelleen kirjoittaessa asiakassovelluksen koodikantaa voitaisiin samalla mahdollisesti uudistaa. Samalla komponentteja olisi mahdollista tuottaa nykyistä helpommin testattavaa koodia, joka mahdollistaisi paremman automaatiotestauksen toteuttamisen, vaikka käyttäjille näkyvät hyödyt jäisivätkin tässä pienemmiksi. Toisaalta taas testauksen parantuessa todennäköisyys ohjelmiston virheille pienenee, jolloin käyttäjät kohtaisivat vähemmän ongelmia ohjelmiston käytössä.

HTTP-pyyntöjen minimointi voi olla hyödyllistä myös palvelimen toiminnan kannalta Miradoren laitemäärän jatkaessa kasvuaan. Mikäli jossain vaiheessa pyyntöjen määrää halutaan syystä tai toisesta vähentää, Android-asiakassovelluksen osalta se olisi teknisesti mahdollista. Muutokset voivat toisaalta vaikuttaa kokonaisuuteen haitallisestikin. Kuvaillut muutokset vaativat paljon uutta koodia, jolloin uusia ohjelmointivirheitä voi syntyä. Myös muuttunut viestinnän luonne asiakassovelluksen ja palvelimen välillä voi aiheuttaa ongelmia, esimerkiksi viestien koon muuttuessa ja viestin keskimääräisen käsittelyn keston kasvaessa. Laajamuotoinen ja hyvin suunniteltu testaaminen ongelmien välttämiseksi olisi tärkeää, jos suurempia muutoksia alettaisiin harkita.

7 Tulokset ja yhteenveto

Työn tarkoituksena oli tutkia Miradoren Android MDM -asiakassovelluksen ja palvelimen välistä kommunikaatiota ja sen komponentteja etsien mahdollisia kehityskohteita ja esittää niihin ratkaisuja. Android-asiakassovelluksen toimintaa ja kommunikaatiota palvelimen kanssa esiteltiin esimerkein ja havainnollistettiin kuvien avulla. Kommunikaation toteutusta analysoitiin, ja siitä löytyneitä kehityskohteita tuotiin esille. Löytyneisiin pienempiin kehityskohteisiin ehdotettiin käytännön ratkaisuja. Lopuksi pohdittiin vielä suurien muutosten toteutusta ja vaikutuksia.

Kommunikaation toteutuksesta löytyi selkeitä, pieniä kehityskohteita, joiden korjaaminen olisi helpohkoa ja vaikutus selkeä. Esimerkiksi oletettavasti epähuomiossa toteutettu asiakassovelluksen StatusTask-luokkien tilojen yksittäin käsittely saataisiin huomattavasti tehokkaammaksi pienellä vaivalla.

Suurempien synkronointitapahtumien muutosten osalta tarvittava työn määrä ei todennäköisesti tuottaisi riittävästi hyötyjä, jotta toteutus olisi kannattavaa. Toisaalta, jos tulevaisuudessa esimerkiksi huomataan suuren pyyntömäärän aiheuttavan ongelmia, voidaan näiden muutosten tekeminen ottaa harkintaan.

Kaiken kaikkiaan asiakassovelluksen kommunikaatio palvelimen kanssa pienistä kehityskohteista huolimatta on kuitenkin toimivaa. Nykytoteutus palvelee käyttötarpeita hyvin, eikä sen toiminnassa ole huomattu suuria ongelmia tuotantokäytössä. Ehdotettujen pienien muutosten toteutumisen jälkeen voitaisiin todeta asiakassovelluksen ja palvelimen välisen kommunikaation, nykytoteutuksen puitteissa, olevan tehokasta.

Lähteet

1 Mohiuddin, Khaja Taiyab. 2023. Mobile Device Management and Their Security Concerns. International Research Journal of Engineering and Technology (IRJET). Vol. 10, No. 10, s. 834–840.

2 Kumar, L. K. Suresh. 2024. Clustered Based Technological and Organizational Aspects of Mobile Device Management and Their Security Concerns. The Bioscan. 19(2): S.I(1), s. 722–728.

3 Mobile Device Management. 2025. Verkkoaineisto. Miradore. <https://www.miradore.com/device-management/mobile-device-management-mdm/>. Luettu 5.2.2025.

4 Miradore client for Android. 2022. Verkkoaineisto. Miradore. <https://www.miradore.com/knowledge/android/miradore-client-for-android/>. Luettu 5.2.2025.

5 Android Jetpack. 2017. Verkkoaineisto. Android Developers. <https://developer.android.com/jetpack/androidx/releases/archive/arch#100_-_november_6_2017>. Päivitetty 3.1.2024. Luettu 24.7.2024.

6 Deploying a configuration profile. 2022. Verkkoaineisto. Miradore. <https://www.miradore.com/knowledge/features/deploying-configuration-profile/>. Luettu 10.2.2025.

7 Schedule tasks with WorkManager. Verkkoaineisto. Android Developers. <https://developer.android.com/topic/libraries/architecture/workmanager>. Luettu 25.7.2024.

8 Firebase Cloud Messaging. Verkkoaineisto. Google Firebase. <https://firebase.google.com/products/cloud-messaging>. Luettu 27.5.2024.

9 HTTP Reference. Verkkoaineisto. Mozilla. <https://developer.mozilla.org/en-US/docs/Web/HTTP>. Luettu 5.4.2024.

10 Zero-touch enrollment for IT admins. Verkkoaineisto. Android Enterprise. <https://support.google.com/work/android/answer/7514005?hl=en>. Luettu 23.7.2024.

11 Android Zero Touch data sheet. 2020. Verkkoaineisto. Android Enterprise. <https://storage.googleapis.com/android-com/resources/enterprise/pdfs/Zero-touch-data-sheet-2020.pdf>. Luettu 23.7.2024.

12 Java.net. Verkkoaineisto. Android Developers. <https://developer.android.com/reference/java/net/package-summary>. Luettu 31.7.2024.

13 Java, HttpClient. Verkkoaineisto. Oracle. <https://docs.oracle.com/en/java/javase/11/docs/api/java.net.http/java/net/http/HttpClient.html>. Luettu 31.7.2024.

14 Managed Google Play applications in Miradore. 2022. Verkkoaineisto. Miradore. <https://www.miradore.com/knowledge/android/how-to-add-managed-google-play-applications/>. Luettu 9.9.2024.

15 TCP (Transmission Control Protocol) – The transmission protocol explained. 2020. Verkkoaineisto. Ionos. <https://www.ionos.com/digitalguide/server/know-how/introduction-to-tcp/>. Luettu 6.8.2024.

16 Git. Verkkoaineisto. Git. <https://git-scm.com/>. Luettu 20.9.2024.

17 Azure. Verkkoaineisto. Microsoft. <https://azure.microsoft.com/en-us>. Luettu 26.8.2024.

18 Google Play EMM API. Verkkoaineisto. Android Enterprise. <https://developers.google.com/android/work/play/emm-api>. Luettu 11.9.2024.

19 Samsung Knox. Verkkoaineisto. Samsung. <https://www.samsungknox.com/en>. Luettu 27.8.2024.

20 Android Device Policy Manager. Verkkoaineisto. Android Developers. <https://developer.android.com/reference/android/app/admin/DevicePolicyManager#setResetPasswordToken(android.content.ComponentName,%20byte[])>. Luettu 27.8.2024.

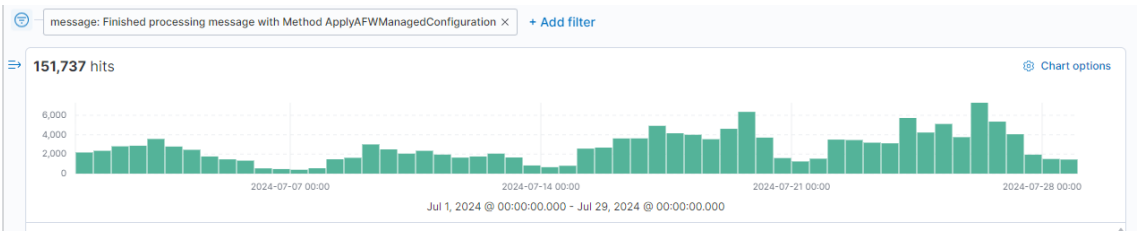
21 Firebase Cloud messaging HTTP API reference. Verkkoaineisto. Google Firebase. <https://firebase.google.com/docs/cloud-messaging/http-server-ref>. Luettu 20.9.2024.

22 Build app server send requests. Verkkoaineisto. Google Firebase. <https://firebase.google.com/docs/cloud-messaging/send-message#send-messages-to-multiple-devices>. Luettu 23.9.2024.

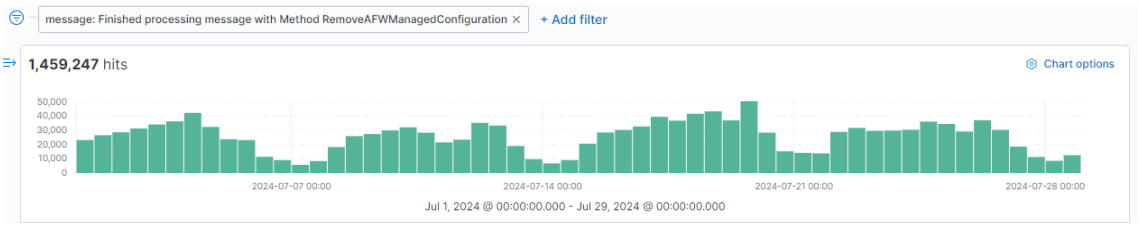
23 Best practices when sending FCM messages at scale. Verkkoaineisto. Google Firebase. <https://firebase.google.com/docs/cloud-messaging/scale-fcm#create-rollout>. Luettu 23.9.2024.

NotificationService-komponentin Managed Google Play -viestit

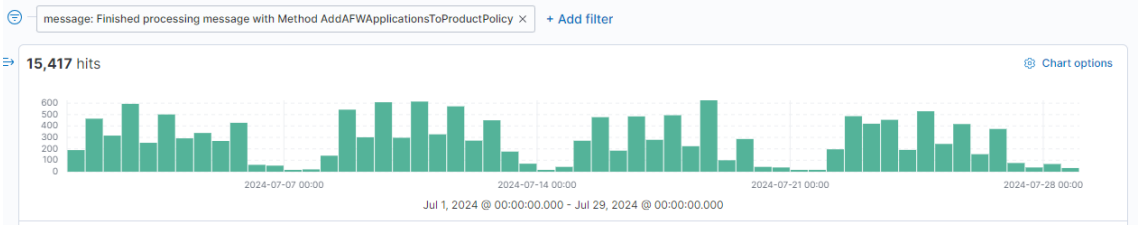
NotificationService-komponentin käsittelemät viestit viestityypeittäin Managed Google Playn osalta heinäkuun 2024 ensimmäisten neljän viikon ajalta.



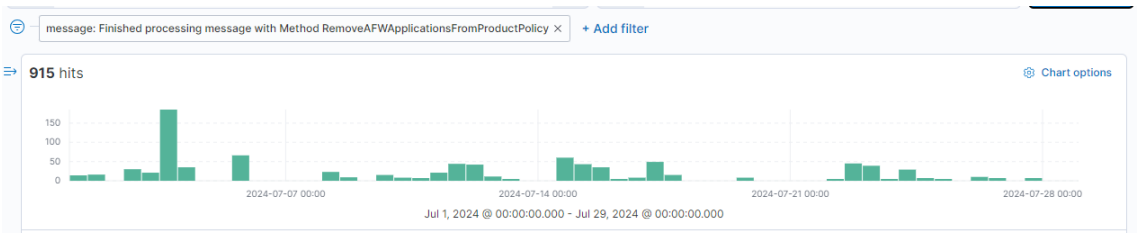
Kuva 1. Käsitellyt ApplyAFWManagedConfiguration-viestit.



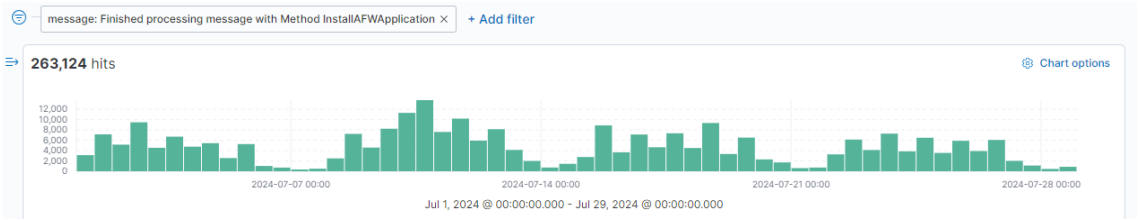
Kuva 2. Käsitellyt RemoveAFWManagedConfiguration-viestit.



Kuva 3. Käsitellyt AddAFWApplicationsToProductPolicy-viestit.



Kuva 4. Käsitellyt RemoveAFWApplicationsFromProductPolicy-viestit.



Kuva 5. Käsitellyt InstallAFWApplication-viestit.