Opinnäytetyö (AMK)

Tieto- ja viestintätekniikka

2020

Teemu Pusa

PILVIPOHJAINEN KORTINLUKIJA

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2020 | 30 sivua

Teemu Pusa

PILVIPOHJAINEN KORTINLUKIJA

Joukkoliikenteessä on pitkään käytetty etäluettavia älykortteja. Näiden korttien sisäiseen tallennustilaan tallennetaan käyttäjän lippujen tietoja. Kortille tallennettuja tietoja voidaan käyttää busseissa matkustusoikeuksien tarkistamiseen.

Opinnäytetyön tarkoituksena oli suunnitella olemassa olevalle kortinkirjoitustoiminnallisuudelle pilvipohjainen vaihtoehto helpottamaan siirtymävaihetta id-pohjaisten matkakorttien käyttöönottamiselle. Työssä toteutettiin pilvipohjainen kortinlukija ja sen asiakaskirjasto.

Työn toteutukseen käytettiin gRPC-kirjastoa yhteyden protokollana, Javascript-, Kotlin- ja C++-ohjelmointikieliä ohjelman toteutuksessa, MIFARE-tuoteperheen älykortteja ja korttien kanssa kommunikointiin APDU-viestejä ja PC/SC-rajapintaa.

Opinnäytetyö aloitettiin rakentamalla pilvipohjaisesta kortinlukijasta prototyyppiratkaisu, jonka avulla tarkistettiin, että työlle asetetut vaatimukset ovat realistisesti toteutettavissa. Tämän jälkeen rakennettiin kirjaston tuotantoversio.

ASIASANAT:

gRPC, MIFARE, PC/SC, APDU, Nodejs

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2020 | 30 pages

Teemu Pusa

CLOUD-BASED CARD READER

Contactless smart cards have long been used in public transport. Information on the purchased tickets is stored to the internal storage of these cards. Information stored to these cards can be used to check travel rights when using buses. The purpose of the thesis was to design a cloud-based card reader solution which works with an existing system and facilitates transition to the online based solution. A cloud-based card reader and its client library were implemented in this thesis.

The gRPC library was used as the connection protocol. Javascript-, Kotlin- and C++-programming languages were used in the implementation of the program. The MIFARE product family of smart cards for communication APDU messages and PC/SC interface were used

First a functional prototype of the cloud-based card reader was successfully built and used for testing that the requirements for the work were realistically feasible. After that, a production version was built and integrated to the existing system.

KEYWORDS:

gRPC, MIFARE, PC/SC, APDU, Nodejs

SISÄLTÖ

KUVAT

1 JOHDANTO

Perinteisesti busseissa käytettävissä bussikorteissa tieto siitä, minkälaisia matkustusoikeuksia käyttäjällä on kirjoitetaan kortille. Fyysiselle kortille kirjoittamisessa on kuitenkin ongelmana, että niiden sisältöä voi olla hankala muuttaa. Esimerkiksi tilanteessa, jossa käyttäjä ostaa netin kautta lippuja niiden saaminen kortille voi olla hankalaa ja kestää jonkin aikaa. Tämä tekee myös myyntilaitteista monimutkaisia, kun ottaa huomioon, että tuotetiedot pitää tuoda laitteille ja myyntidata lähettää palvelimelle. Tästä syystä nykyään kehitetään palvelinpohjaisia myyntiohjelmistoja, jolloin online-myynneissä ja -leimauksissa kutsut lähetetään suoraan palvelimelle ja ne muuttavat palvelimen tietokannassa olevia kortin tietoja. Tämä helpottaa monimutkaisten myynti- ja leimauslogiikkojen toteuttamista ja päivittämistä. Siirtyminen online-pohjaisuuteen ei ole kuitenkaan helppo askel, sillä myyntijärjestelmät ovat jo tuotantokäytössä ja siirtovaiheen pitäisi olla mahdollisimman häiriötön. Tämän takia kehitetään vaihtoehtoista ratkaisua, jonka avulla voitaisiin siirtyä pilvipohjaisuuteen helpommin tulevaisuudessa.

Käytännössä fyysisen ja online-myynnin suurin ero on se, että vain fyysisessä myynnissä tiedot kirjoitetaan myös kortille. Tietojen säilöminen kortille ei ole kuitenkaan aivan yksinkertainen prosessi. Kortilla oleva tallennustila on rajallinen, ja kortille tallennettavien tietojen muuttaminen on hankalaa. Kortin käsittelyyn tarvitaan myös avaimia eikä ole varmaa mihin ne säilötään. Kortti on eräänlainen tietokone, sitä voidaan ohjata APDU-komentojen avulla. Jos avainten käyttäminen tapahtuisi palvelimen puolella, niitä ei tarvitsisi lähettää asiakkaalle ollenkaan. Kyky lähettää APDU-komentoja laitteelle, jossa lukija on kiinni, vaatii kuitenkin muutamien ongelmien ratkaisemisen. Kun asiakas laittaa kortin lukijatasolle tulisi sisällön lukemisen tapahtua mahdollisimman nopeasti. Ongelmia kuitenkin tuottaa palvelimen ja asiakkaan välillä oleva viive. Pienikin viive aiheuttaa suuria eroja nopeudessa riippuen komentojen lähetystavasta.

Työn tarkoituksena on kehittää APDU-komentojen kommunikaatioväylä palvelimen ja asiakkaan välille edellä mainitut ongelmat huomioiden.

2 VAATIMUKSET

Jotta kirjastoa on mahdollista käyttää tuotannossa, sen pitää täyttää muutamia sille asetettuja vaatimuksia. Tässä luvussa käydään läpi kirjastolle asetettuja vaatimuksia.

2.1 Taaksepäin yhteensopivuus

Kirjaston on tarkoitus olla täysin yhteen sopiva vanhan järjestelmän kanssa. Tämän takia kirjaston pitää tukea vähintään samoja älykorttityyppejä kuin vanha järjestelmä. Vähintään tuettavia kortti tyyppejä ovat MIFARE tuoteperheen Classic- ja DESFire-kortit. Korteille tallennettava binäärirakenne on tarkkaan suunniteltu ja kirjaston pitää pystyä tallentamaan täysin samanlainen rakenne kortille. Tämän binäärirakenteen käsittely, on kirjaston yksi vaikeimmista tehtävistä, mutta se hoituu olemassa olevan kirjaston avulla.

2.2 Toiminta pilvipohjaisen myyntiohjelmiston kanssa

Kirjaston tärkein vaatimus on sen saumaton toiminta olemassa olevan palvelinpohjaisen myyntijärjestelmän kanssa. Myyntiohjelman käyttöliittymän kannalta halutaan, että fyysisten korttien käyttäminen olisi mahdollisimman samanlaista kuin online-pohjaisten tilien. Tämä mahdollistaa myös fyysisten ja online-pohjaisten tuotteiden käyttämisen samalla tilillä. Käytännössä tämä tarkoittaa, että kirjaston pitää tarjota rajapinnat, joiden avulla myyntiohjelmisto voi kontrolloida pilvipohjaista kortinlukijaa.

2.3 Kehityksen yksinkertaistaminen

Kirjaston yksi tarkoitus on yksinkertaistaa asiakasohjelmien kehitystä. Koska monimutkainen kortin käsittelylogiikka on siirretty palvelimen puolelle, asiakasohjelmien ei tarvitse toteuttaa sitä. Tämän takia asiakaskirjaston tarvitsee toteuttaa vain yhteyden vaatima toiminnallisuus. Kirjastosta tarvitaan toteutukset Android- ja Windows-laitteille. Vaikka alustoille tehdään ohjelmia eri ohjelmointikielillä, on asiakaskirjasto niin helppo toteuttaa, ettei se ole ongelma.

2.4 Nopeus

Kortin lukeminen tapahtuu aina ensimmäisenä, kun kortille tehdään jotakin. Lukuoperaation nopeus antaa käyttäjälle ensimmäisen kuvan koko järjestelmän nopeudesta. Asiakkaat eivät halua odottaa useita sekunteja operaatioiden suoriutumista. On siis tärkeää, että kirjasto toimii riittävän nopeasti. Kokonaisviive koostuu monesta osasta. Ensimmäinen kokonaisviiveeseen vaikuttava tekijä on kortin ja lukijan välinen viive, johon vaikuttaa kortinlukijan- sekä kortintyyppi. Toinen merkittävästi viiveeseen vaikuttava tekijä on asiakkaan ja palvelimen välillä oleva viive. Kortin käsittely palvelimella on niin nopeaa, ettei se vaikuta merkittävästi kokonaisviiveeseen.

Viiveen kannalta on myös tärkeätä valita oikeat tekniikat, joilla viive saadaan minimoitua. Yhteyden kannalta on myös tärkeätä, että se on käyttökelpoinen myös google-pilvipalvelussa pyörivälle ohjelmalle. Kokonaisviiveen saa suurin piirtein laskemalla lukukomentojen suorittamiseen menevän ajan ja lisäämään tähän pari kertaa palvelimen ja asiakkaan välisen viiveen verran aikaa. Perinteisen järjestelmän viive on yli 200 ms, joten tämä antaa vertailuarvon, jonka lähelle pitäisi pyrkiä.

3 KÄYTETYT TEKNOLOGIAT

Kirjasto tulee osaksi olemassa olevaa myyntijärjestelmää, jonka takia sen tarvitsee olla toteutettu yhteensopivien tekniikoiden avulla. Myyntijärjestelmän palvelinpuolen ohjelmaa ajetaan Googlessa GKE-klusterin sisällä, joka on Googlen hallinnoima Kubernetes-klusteri. Palvelinohjelmat paketoidaan Docker-kontin sisälle. Kun mahdollista, kirjaston toteuttamisessa käytetään pääasiallisesti Typescript-ohjelmointikieltä, koska myyntiohjelma on toteutettu samalla kielellä. Tämän luvun aikana käydään lyhyesti lävitse kirjaston toteutukseen käytettyjä tekniikoita.

3.1 Docker-ohjelma

Docker-ohjelmisto, jolla ohjelmat ja niiden riippuvuudet voidaan paketoida muotoon, jossa niitä voidaan ajaa helposti millä tahansa linux-koneella. Docker on suosittu työkalu etenkin palvelinpuolen ohjelmistojen keskuudessa.

3.2 Kubernetes-ohjelmisto

Kubernetes on Googlen sisäisen Borg-nimisen järjestelmän pohjalta Google kehittämä avoimen lähdekoodin klusterin hallintasovellus. Ohjelma paketoidaan ensimmäiseksi Docker-kontin sisälle ja tämän jälkeen luodaan deployment-tiedostoja, jossa tätä luotua konttia käytetään. Kubernetes siis hallitsee docker: rilla tehtyjä ohjelmia ja käynnistää ne uudelleen, jos ne kaatuvat.

3.3 Protobuf-kirjasto

Protobuf on Googlen kehittelemä kirjasto, jonka tarkoituksena on rakeenteellisten tietorakenteiden serialisointi binäärirakenteiksi. Protobuf on monikäyttöinen kirjasto, sitä voidaan käyttää monissa eri käyttötarkoituksissa. Yksi käyttötarkoitus on esimerkiksi rakenteellisten viestien muuntaminen pieneen tehokkaaseen muotoon, vaikka TCP/IP-

kerroksen ylitse lähettämistä varten. Voidaan sanoa, että Protobuf toimii vaihtoehtona nykyisin perinteiselle JSON-formaatille, mutta kummallakin teknologialla on omat heikkoutensa ja vahvuutensa. JSON-formaatissa ei ole periteisesti minkäänlaista määritystä rakenteesta, vaan se on itseään kuvaava. Siksi työskentely on nopeaa, mutta mikään ei takaa, että osapuolet ymmärtävät toisiaan ja lähettävät samankaltaisia viestejä. Formaatti vie myös paljon tilaa tietosisällön määrään nähden. Protobuf-formaatti käyttää .proto-tiedostoja kuvaamaan tietosisällön rakenteen. Tämän määritys tiedoston avulla voidaan generoida eri kielille protoc-kääntäjän avulla ohjelmakoodia. Muodostunut ohjelmakoodi voi olla vaikka C++-kielen luokka ja rakenteen jäseniä käsitellään get- ja set-metodien avulla ja binäärirakenne luodaan jonkun toisen metodin avulla. Vaikka Protobuf tekeekin tehokkaita ja pieniä binäärirakenteita niin se tekee virheiden selvittämisestä hankalaa, koska viestin binäärirakenteet eivät ole ihmisen ymmärrettävässä muodossa. [4]

Protobuf-määritys Person-viestille, jolla on kaksi jäsentä string-tyyppinen name ja uint32-tyyppinen age. Tiedoston alussa oleva ”syntax = ”proto3”;” kertoo, että käytetään protobuffin kolmatta standardia. (Kuva 1)

example-proto.proto

```
1  syntax = "proto3";
2
3  message Person {
4    string name = 1;
5    uint32 age = 2;
6  }
7
```

Kuva 1 Esimerkki yksinkertaisesta proto-tiedostosta.

3.4 gRPC-kirjasto

gRPC on Googlen sisäisen tekniikan perustalta Googlen tekemä kirjasto, jonka avulla on helppo tehdä RPC-kutsuja eri kielillä tehtyjen ohjelmien välillä. Kirjasto on toteutettu HTTP/2-protokollan päälle, jonka takia se tukee myös kaksisuuntaisten stream yhteyksien luomista, mikä on oleellinen ominaisuus tämän järjestelmän kannalta. gRPC käyttää palvelun määrittämiseen, proto-formaattia, joka on sama kuin protobuffissa, mutta siihen on laajennettu Service-käsite. Tämä tarkoittaa käytännössä, että kun palvelu on esitelty

.proto-tiedostossa näistä luodaan automaattisesti yksinkertaiset toteutukset halutuille kielille. [3]

Yksinkertaisen Example-palvelun määrittely. Palvelu sisältää yhden Hello-metodin, joka ottaa parametriksi HelloRequest-tyypin ja palauttaa HelloResponse-tyypin. (Kuva 2)

example-grpc-service.proto

```
1   syntax = "proto3";
2
3   package ExampleService;
4
5   message HelloRequest {
6       string name = 1;
7   }
8
9   message HelloResponse {
10      string greetingText = 1;
11  }
12
13  service Example {
14      rpc Hello(HelloRequest) returns(HelloResponse) {}
15  }
```

Kuva 2 Esimerkki yksinkertaisesta grpc-palvelun määrityksestä.

3.5 Nodejs-ohjelma

Nodejs on avoimen lähdekoodin alustariippumaton Javascript-ohjelmointiympäristö. Nodejs-avulla voidaan ajaa Javascript-ohjelmia Windows- ja Linux-käyttöjärjestelmillä, mutta pääasiallisesti sitä käytetään palvelinohjelmistojen kehitykseen Javascript-kielellä. Perinteisesti Javascript ajetaan jonkun virtuaalikoneen sisällä. Nodejs käyttää V8 nimistä virtuaalikonetta. V8 on täydellinen hiekkalaatikko, ja sillä ei ole yhteyttä kernelin operaatioihin. Nodejs toteuttaa rajapinnan Javascriptin ja käyttöjärjestelmän kernelin välille. Tämän avulla Javascript-koodista voidaan luoda esimerkiksi tiedostojärjestelmään tiedostoja. [9]

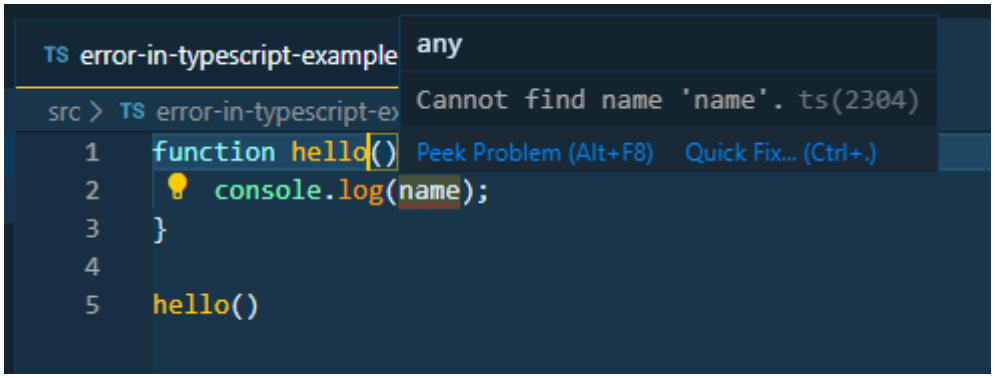
3.6 Typescript-kieli

Javascript on dynaaminen tyypittämätön ohjelmointikieli, mikä tekee kielen kanssa prototyypittämisestä nopeaa. Tämä tuo kuitenkin mukanaan muutamia ongelmia ohjelman koon kasvaessa. Koska Javascript-tulkki ei tarkista kielen syntaksi-virheitä monet ohjelmoijan tekemät virheet jäävät ajon aikaisiksi virheiksi. Ajon aikaiset ohjelmavirheet ovat pahempia kuin käännöksen aikaiset virheet, koska ne saattavat tulla esille vasta ohjelman tuotantokäytössä. Typescript on Microsoftin kehittämä ylemmän tason kieli Javascriptille. Se tarjoaa mahdollisuuden ohjelman vaiheittaiselle tyypitykselle. Kieli on suunniteltu ja suunnitellaan niin, että se seuraa aina tiukasti Javascriptin virallista määritystä. Tämä tarkoittaa käytännössä, että kieleen ei pyritä tekemään ominaisuuksia, joita Javascriptin kehityssuunnitelmissa ei ole mainittu. Typescript-kääntäjä tarkistaa ohjelmakoodin ja muuntaa sen perinteiseksi Javascriptiksi. [8]

Ohjelmaa, joka antaa virheen ”ReferenceError: name is not defined”, joka tarkoittaa, että name-muuttaa ei ole määritelty. (Kuva 3)

Kuva 3 Esimerkki virheellisestä Javascript-ohjelmasta.

Ohjelmassa on sama virhe kuin edellisessä, mutta kuten huomataan, kääntäjä kertoo tämän tyyppiset virheet jo alussa. (Kuva 4)



Kuva 4 Esimerkki virheellisestä Typescript-ohjelmasta.

3.7 MIFARE

MIFARE on NXP Semiconductors omistama tavaramerkki, johon kuuluu sarja kontaktittomia älykortteja ja läheisyyskortteja. Brändillä on useita tuotteita, jotka perustuvat ISO/IEC 14443 standardiin. [6]

MIFARE Classic on muistilaite, jossa muisti on jaettu segmentteihin ja lohkoihin yksinkertaisen avainpohjaisen valvonnan avulla. MIFARE Classic-kortteja on olemassa eri kokoisia. MIFARE Classic 1k:ssa on 1 024 t tallennustilaa, joka on jaettu 16 sektoriin ja jokainen sektori on suojattu kahdella avaimella, joiden nimet ovat A ja B. Jokainen avain voidaan ohjelmoida sallimaan operaatioita kuten kirjoitus, lukeminen ja arvo lohkon arvon nostaminen. MIFARE Classic 4k:ssa 4 096 t tallennustilaa, joka on jaettu 40 sektoriin, joista 32 on samankokoisia kuin 1k kortilla ja 8 yli nelinkertaisena kokona. MIFARE mini tarjoaa 320 t tallennustilaa jaettuna viiteen sektoriin. Normaalitilanteessa jokaisen Classic-korttityypin sektorin 16 viimeisintä tavua on varattu avaimille ja valvonnan säännöille. Jokaisen kortin 16 ensimmäistä tavua on varattu kortin sarjanumerolle ja joillekin muille valmistajan tiedoille. Nämä 16 tavua ovat vain luku tilassa. Näiden asioiden takia kortin oikea tallennuskapasiteetti on huomattavasti pienempi kuin on ilmoitettu. Esimerkiksi 1k kortilla käytettävissä oleva tallennuskapasiteetti putoaa 752 tavuun. [6]

MIFARE DESfire on huomattavasti kehittyneempi kuin classic ja se sisältää tehtaalta tullessaan NPX tekemän yleiskäyttöisen käyttöjärjestelmän. Tämä käyttöjärjestelmä tarjoaa yksinkertaisen tiedostojärjestelmän, johon on mahdollista luoda kansioita ja tiedostoja. MIFARE DESfire kortteja on kolmen tyyppisiä D40, EV1 ja EV2. D40 on näistä ensimmäisenä julkaistu versio, joka tukee vain Triple-DES salausta ja tarjoaa neljä

kilotavua tallennustilaa. EV1 tarjoaa 2 kt, 4 kt, tai 8 kt tallennustilaa. EV2 on näistä korttityypeistä uusin ja tarjoaa 2 kt, 4 kt, 8 kt, 16 kt tai 32 kt tallennustilaa. [6]

3.8 Application protocol data unit (APDU)

APDU on älykorttien kanssa käytetty viestintäyksikkö älykorttilukijan ja älykortin välillä. APDU-komentojen rakenne on määritelty standardissa ISO/SEC 7816-4. On olemassa kahden kategorian APDU-komentoja ja vastauksia. Komento on se, jonka lukija lähettää kortille ja siihen kuuluu neljän tavun pakollinen otsikko ja 0-65 535 tavua dataa. Vastaus on kortin lähettämä vastaus komennolle ja se sisältää 0-65 536 tavua dataa ja lopuksi kaksi pakollista tilatavua. Tilatavujen mahdolliset arvot riippuvat komennon tyypistä ja luokasta, mutta oletusluokan kanssa 0x90 0x00 merkitsee komennon onnistumista. [7]

APDU komennon otsikko alkaa yhden tavun mittaisella komennon luokalla. Komennon luokan oletus arvo on 0xFF, jolloin komento tulkataan lukijan puolesta ja sitä ei lähetetä kortille. Seuraava tavu on komennon tyyppi. Tämän mahdolliset arvot riippuvat siitä, mikä on komennon luokka. Tämän jälkeiset kaksi tavua ovat komennon parametreja. Nämä parametrit ovat monikäyttöisiä ja niiden mahdolliset arvot riippuvat komento luokan ja komento tyypin arvoista. [5]

Seuraavaksi tulee APDU komennon valinnaiset arvot. Ensimmäisenä tulee 1-3 tavun pituinen komennon datan pituus. Toisena tulee yleinen data komennolle ja sen pituus pitää olla sama kuin edellisen ilmoittama pituus. Lopuksi tulee 1-3 tavun kokoinen odotettu maksimi pituus vastaukselle. [7]

4 KIRJASTON SUUNNITTELUPERIAATTEET

Kirjaston yksi tärkeimpiä vaatimuksia on nopeus. Nopeuden saavuttaminen tilanteen takia ei ole kuitenkaan yksiselitteistä. Nopeuden saavuttaminen vaatii joidenkin kompromissien tekemistä. Yhden komennon ajaminen kortille kestää muutamia millisekunteja komennon mukaan. Komennot ajetaan kortille yksi kerrallansa ja suuri osa ajasta menee tähän. Saksassa sijaitsevan palvelimen viive on 40 ms, kunhan käytössä on hyvät tietoliikenneyhteydet. DESFire-tyyppisen kortin lukemiseen tarvitaan 22 komentoa. Jos nämä komennot lähetettäisiin yksi kerrallaan asiakkaalle, kestäisi kortin lukeminen ilman kortin kirjoitusnopeuden huomioon ottamista 880 millisekuntia. Kun tähän lisätään vielä kortin kirjoitusnopeus, niin lopullinen aika menee jo yli yhden sekunnin.

Kortin kirjoitusnopeuteen ei voida vaikuttaa, mutta aikaan, joka menee komentojen lähettämiseen palvelimelta asiakkaalle, voidaan vaikuttaa. Palvelimen ja asiakkaan välinen viive pysyy samana riippumatta siitä, lähetetäänkö asiakkaalle yksi APDU-komento vai monta. Ratkaisu on siis paketoida lähetettävät komennot mahdollisimman suuriin paketteihin ja lähettää ne yhdessä viestissä asiakkaalle. Asiakas suorittaa nämä komennot siinä järjestyksessä kuin ne tulivat ja kokoaa vastauksista samankaltaisen paketin. Kaikki komennot suoritettuaan asiakas lähettää paketin takaisin palvelimelle tutkittavaksi. Näin viive saadaan laskettua suurimmassa osassa tilanteista alle 300 ms.

Ratkaisu ei kuitenkaan ole täysin yksiselitteinen ja siinä on muutamia ongelmia. Koska protokolla on suunniteltu niin, että asiakasohjelmista tulee mahdollisimman yksinkertaisia ne eivät ymmärrä kortille lähetettyjä viestejä tai kortin vastauksia. Tämä tulee ongelmaksi, kun esimerkiksi 22 lähetetystä komennosta esimerkiksi viides komento epäonnistuu. Sen jälkeen tulevat komennot suoritetaan, vaikka niitä ei pitäisi. Tämä virhetilanne täytyy kuitenkin hyväksyä, sillä nopeus ja asiakasohjelmien yksinkertaisuus ovat tärkeämpiä ominaisuuksia.

5 TUETUT KORTTITYYPIT

Pilvilukija tukee kahdenlaisia korttityyppejä MIFARE DESfire ja MIFARE Classic. Näiden käyttämisessä on merkittäviä eroja. Koska järjestelmän tarvitsee olla yhteensopiva vanhojen käytössä olevien järjestelmien kanssa. Tämä aiheuta tilanteita, joissa tarvitsee tehdä kompromisseja eri korttityyppien puutteiden takia.

5.1 MIFARE Classic

MIFARE Classic 1k -kortti koostuu 16 sektorista ja jokaisessa sektorissa on neljä 16 t:n kokoista blokkia. Sektorin kolme ensimmäistä blokkia on varattu datan tallentamiseen ja neljäs on varattu avaimia varten. Classic-kortin käyttäminen perustuu näiden 16 t:n kokoisten lohkojen lukemiseen ja kirjoittamiseen. Classic kortin luku ja kirjoitus operaatiot tapahtuvat aina 16 t:n kokoisina osina ja ei ole mahdollista kirjoittaa tai lukea pienempiä tai suurempia osia kerrallaan. Classic-kortilla jokaisen sektorin kirjoittaminen ja lukeminen on mahdollista suojata eri avaimella.

Kortin lukeminen tai kirjoittaminen aloitetaan lataamalla ensimmäiseksi haluttu avain kortille. Kun kortin johonkin sektoriin halutaan kirjoittaa tai sen sisältöä halutaan lukea, tarvitsee autentikoida jokin sen sektorin lohkoista. Kun yksi kolmesta on autentikoitu niin kaikille saman sektorin lohkoille voi kirjoittaa tai niitä voi lukea. Tämän jälkeen kortille voidaan lähettää kirjoitus tai lukukomento. Autentikoinnissa on myös tärkeää tietää, minkä tyyppinen avain on ja millä numerolla avain on ladattu kortille. Avaintyyppi kuvaa, onko se ladattu kortille A- vai B-tyyppisenä. Kortille on mahdollista asettaa avaimet niin, että lukemiseen ja kirjoittamiseen käytetään eri avaimia esimerkiksi lukemiseen A-tyyppistä ja kirjoittamiseen B-tyyppistä, mutta nämä voivat olla myös sama avain. Koska kortille voidaan kirjoittaa ja sieltä lukea vain 16 tavun kokoisia osioita koko sektorin lukeminen tarkoittaa kolmen lukukomennon lähettämistä kortille. MIFARE Classic-kortilla ei ole tukea transaktioihin perustuvaa tiedon yhtenäisyyden takaamista. Siksi kortin sisältö saattaa jäädä rikkinäiseen tilaan, eli että kun halutaan kirjoittaa kolmen lohkon verran tietoa, mutta kortti otetaan pois lukijasta ennen kuin viimeisimmän komennon suorittaminen on täysin valmis. Tällöin kortti jää rikkinäiseen tilaan. Tällaisia tilanteita ei ole mahdollista huomata kortin puolesta vaan kortin käyttäjän on itse lisättävä tietoa siitä, milloin

kortti on oikeassa tilassa ja käyttäjä voi tarkistaa tämän. Yksi tyyli on lisätä viimeisimpään lohkoon crc-tunniste, jonka avulla voidaan laskea korttia luettaessa, onko se oikein laskettu. Jos crc-tunniste ei täsmää on ainoa tapa olettaa, että koko sisältö on väärää.

5.2 MIFARE DESFire

DESFire-kortin käyttäminen on monimutkaisempaa kuin Classic-kortin, mutta se on myös monipuolisempi ja turvallisempi. DESFire-kortti koostuu ohjelmista ja näiden sisällä olevista tiedostoista. DESFire-kortti toimii samankaltaisesti kuin tiedostorakenne eli ensin valitaan ohjelma samalla tavalla kuin kansio tiedostojärjestelmässä ja sen jälkeen voidaan sen sisällä olevia kansioita muokata tai niitä voidaan tehdä lisää.

Ohjelmilla on kolme tavua pitkä tunniste, jota käytetään, kun ohjelma valitaan. Tämä tunniste annetaan myös, kun ohjelma luodaan. DESFire kortilla voi olla tyypin mukaan monta eri ohjelmaa ja nämä voivat olla vaikka eri toimijoiden käyttämiä. Yleisesti on kuitenkin hyvä idea, että ainakin uusien ohjelmien luominen estetään tuntemattomilta käyttäjiltä. Kortin juurikansio on eräänlainen pääohjelma ja on aina kortin tullessa kenttää automaattisesti valittuna. Juuriohjelmaa koskee samat autentikaatiosäännöt kuin muissa ohjelmissa. Tämä tarkoitta, että juuritason ohjelman avaimet voidaan vaihtaa oletusavaimista, jolloin tuntemattomat käyttäjät eivät voi poistaa tai lisäillä omia sovelluksia kortille. Yhteen ohjelmaan mahtuu korttityypin mukaan useita tiedostoja.

Tieto säilötään kortilla tiedostoihin. Toisin kuin Classic-korteilla DESFire-kortille on mahdollista kirjoittaa erikokoisia tietomääriä mielivaltaisiin kohtiin tiedostoissa. Jokaisella tiedostolla on yhden tavun mittainen tunniste, jota käytetään, kun tiedostoon halutaan kirjoittaa. Tiedostolle voidaan määrittää mitä avainta sen halutaan käyttävän kortille ladatuista avaimista. Tiedoston koko ja sen tyyppi määritellään tiedoston luomisen yhteydessä.

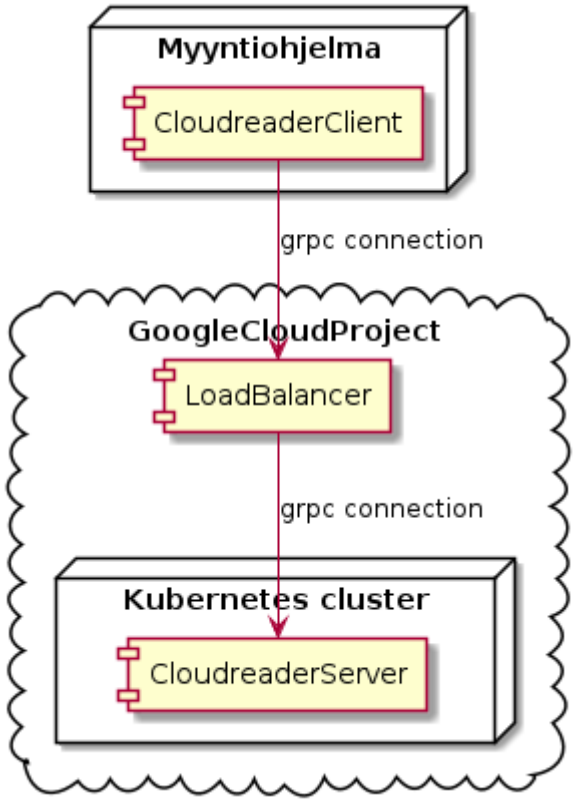
DESFire-kortille voidaan ladata useita avaimia. Avaimien vaihtaminen vaatii autentikaatioita ohjelman pääavaimen avulla.

DESFire kortin tiedostoihin tehtävien muutosten eheys varmistetaan commitointi-toiminnallisuuden avulla. Tämän avulla varmistetaan, että tehtyjen kirjoituskomentojen tekemät

muutoksen kortin tietosisältöön tallentuvat kortille vasta commit-komennon jälkeen. Jos kortti poistetaan lukijasta, kortin tietosisältö jää sellaiseen tilaan, joka se oli ennen muutosoperaatioita.

6 ARKKITEHTUURI

Arkkitehtuuri koostuu asiakasohjelmasta ja palvelinohjelmasta. Asiakasohjelma on mahdollisimman yksinkertainen ja se toteuttaa vain tarvittavan toiminnallisuuden. Asiakaskirjastosta on toteutukset Androidille ja Windows-laitteille. (Kuva 5)

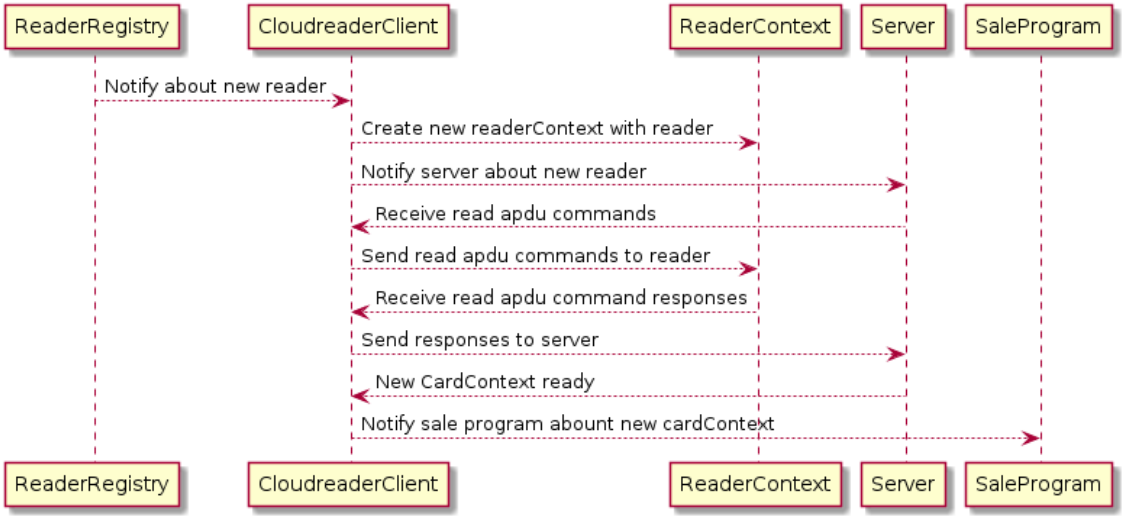


Kuva 5 Järjestelmän arkkitehtuuri.

6.1 Asiakaskirjasto

Asiakaskirjastosta löytyy kaksi eri toteutusta. Toinen on toteutettu Nodejs:llä työasemalle ja toinen Java- ja Kotlin -ohjelmointikielillä Android laitteille. Näiden perustoimintaperiaate on samanlainen, vaikka niiden toteuttamiseen ei käytetä samaa ohjelmointikieltä. Työasemien asiakaskirjasto on suunniteltu niin, että se tukee monen lukijan samanaikaista käyttämistä.

Asiakaskirjaston yhteys avataan open-komennon avulla. Tämän open-komennon mukana välitetään oikea autentikaatio token, jonka palvelin tarkistaa julkisella avaimella. Kun yhteys on avattu, aina kun lukija kytketään kiinni tai irrotetaan, syntyy tapahtuma, joka lähetetään palvelimelle. Jokaista uutta lukijaa varten luodaan uusi UUID-tunniste ja tämä lähetään aina kutsun mukana, jos tehdään operaatioita, jotka koskevat kyseistä lukijaa. Luettujen korttien kanssa hyödynnetään samankaltaista tekniikkaa. Kortin tullessa kortinlukijan kenttään siitä saadaan ATR-tunnus ja samalla luodaan korttia varten UUID-tunniste. Nämä lähetetään palvelimelle ja niitä käytetään, kun halutaan tehdä korttia koskevia operaatioita. Asiakasohjelmiston on tarkoitus ylläpitää näitä kontekseja muistissa ja ohjata palvelimelta tulevat APDU-komennot oikealle lukijalle. (Kuva 6)



Kuva 6 Esimerkki kortin lukutilanteesta asiakaskirjaston kannalta.

6.1.1 Pöytäkonetoteutus

Asiakaskirjasto on Typescriptillä toteutettu käyttöliittymäriippumaton Nodejs:n päälle pyöriväksi tarkoitettu kirjasto. Kortinlukijan kanssa kommunikointiin kirjasto käyttää npm:stä löytyvä nfc-pcsc-kirjasto. Käytännössä se on virallisten node addon-rajapintojen avulla luotu kääre kirjasto PC/SC-rajapinnan päälle. Työasema ympäristöissä on mahdollista, että koneeseen kytketään samanaikaisesti monta eri kortinlukijaa, joten kirjasto on toteutettu niin, että se tukee monen lukijan käyttöä.

6.1.2 Android toteutus

Cloudreader-kirjasto on toteutettu androidille käyttämällä Java- ja Kotlin-ohjelmointikieltä. Android-toteutukssa käytetään laitteiden sisällä olevaa kortinlukupiiriä ja sitä ohjataan järjestelmän tarjoavien funktioiden avulla. Android-laitteissa on vain yksi kortinlukija, joten Android-toteutuksen kirjastosta ei tarvitse tukea monen kortinlukijan käyttämistä. Android-laitteissa testattiin myös ulkoisia bluetooth-lukijoita, mutta niiden komentojen suoritusnopeus oli niin hidas, mikä teki niistä käytännössä käyttökelvottomia tähän käyttötarkoitukseen.

6.1.3 Rajapinta

Kirjaston Android- ja työasematoteutuksia on tarkoitus käyttää alustariippumattomasti saman myyntiohjelman kanssa, minkä takia niiden tarvitsee toteuttaa samankaltaiset rajapinnat. Rajapinta sisältävät sekä callback-tyyppisiä ja normaalisti kutsuttavia metodeita.

Kun yhteys saadaan muodostettua palvelimelle, kutsutaan onConnect callbackiä.

Kun yhteys palvelimeen katkeaa, kutsutaan onDisconnect on callbackiä.

Kun uusi korttikonteksti on palvelimen puolelle muodostettu, kutsutaan onNewCardContext on callbackiä.

Joissakin tilanteissa komennon suorittaminen kortilla epäonnistuu ja on tärkeää, että kirjaston käyttäjä saa tästä tiedon esimerkiksi latausnäkymistä poistumista varten. Kun palvelin huomaa jonkin operaation epäonnistuneen, kutsutaan onCardOperationFailed callbackiä.

Kun kortti poistetaan lukijasta, kirjasto ilmoittaa käyttäjää oncardContextRemoved-callbackin avulla. Kutsussa annetaan parametrina sen korttikontekstin id, joka poistui lukijasta.

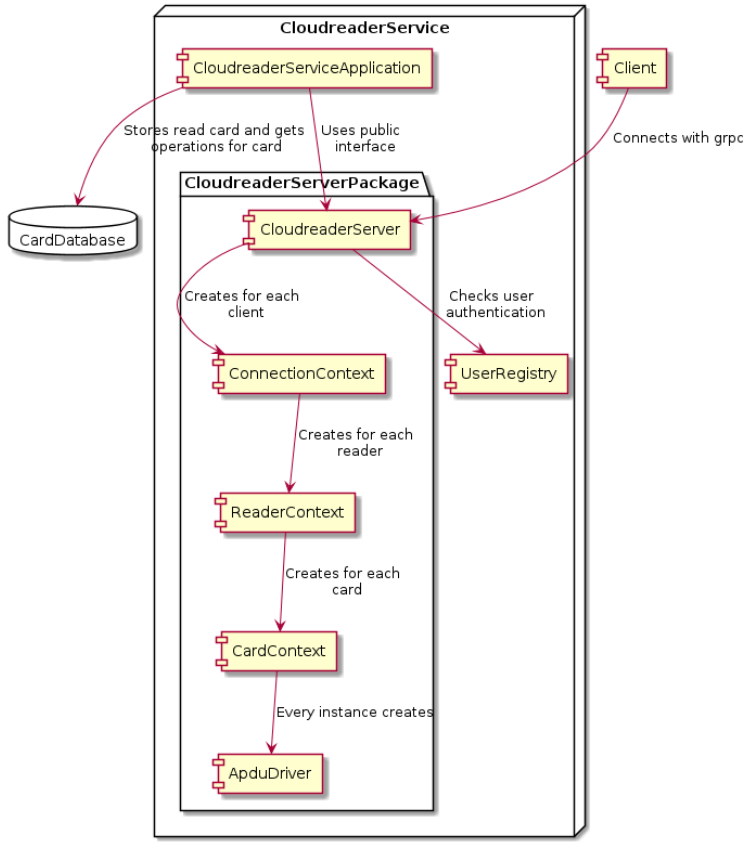
Kun uusi lukija tunnistetaan, kutsutaan onReaderConnected-callbackiä, jonka parametrina tulee sille generoitu readerContextId.

Kun lukijan tunnistetaan poistuvan, kutsutaan onReaderDisconnected-callback sen lukijan readerContextId, jota se koskee.

6.2 Palvelinohjelmisto

Palvelinpuolen osuus pilvilukijasta toteutetaan kirjastona, jota käytetään myyntiohjelmiston palvelussa. Kirjasto on toteutettu pääosin Typescript-kielellä, mutta APDU-ajurit korteille on toteutettu C++-kielellä.

Palvelinohjelma koostuu pilvilukijapalvelusta ja sen sisällä pyöritettävästä palvelinkirjastosta. Kirjasto vastaanottaa asiakkailta tulevat viestit ja käsittelee ne. Pilvilukijapalvelu toteuttaa kirjaston vaatiman rajapinnan, johon kuuluu käyttäjäoikeuksien tarkistaminen, korttikuvien tallennus tietokantaan ja haluttujen operaatioiden välitys kirjastolle. (Kuva 7)

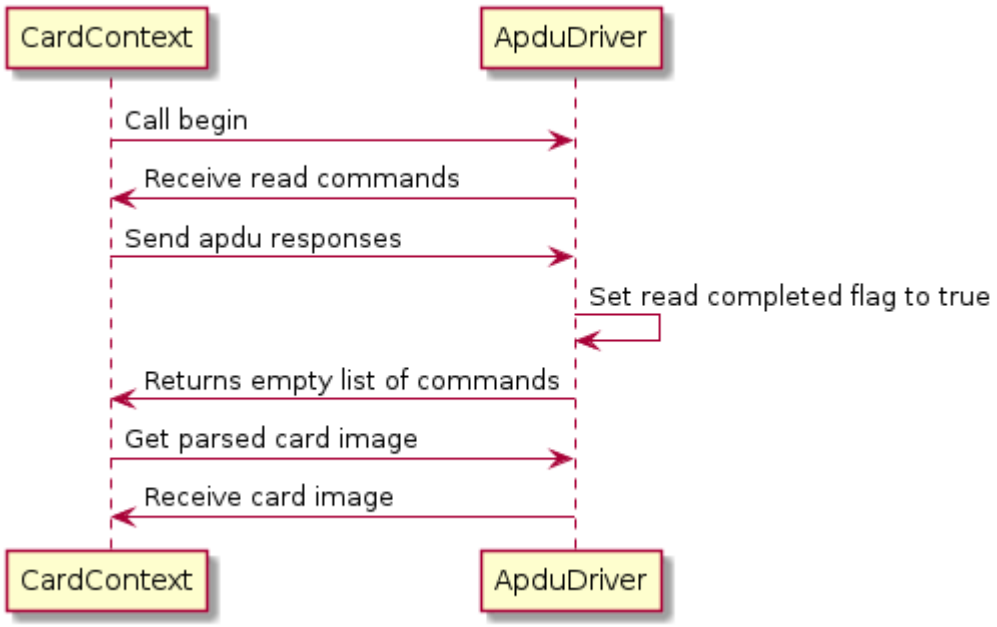


Kuva 7 Komponenttikaavio palvelinkirjaston arkkitehtuurista.

6.2.1 APDU-ajurit

APDU-ajureita on toteutettu kaksi, molemmat näistä toteuttavat abstraktin-luokan ApduDriver. Ajurit on toteutettu C++-kielellä, niille on luotu node addon kanssa rajapinta Javascriptin puolelle. ApduDriver-luokalla on staattinen create-metodi, jolle annetaan parametrina kortin atr-tunniste ja binääridataa, jossa on avaimet. create-metodin toteutus päättelee atr-tunnisteen avulla, onko kyseessä MIFARE Classic vai MIFARE DESFire -kortti ja luo sopivan ajuritoteutuksen instanssin.

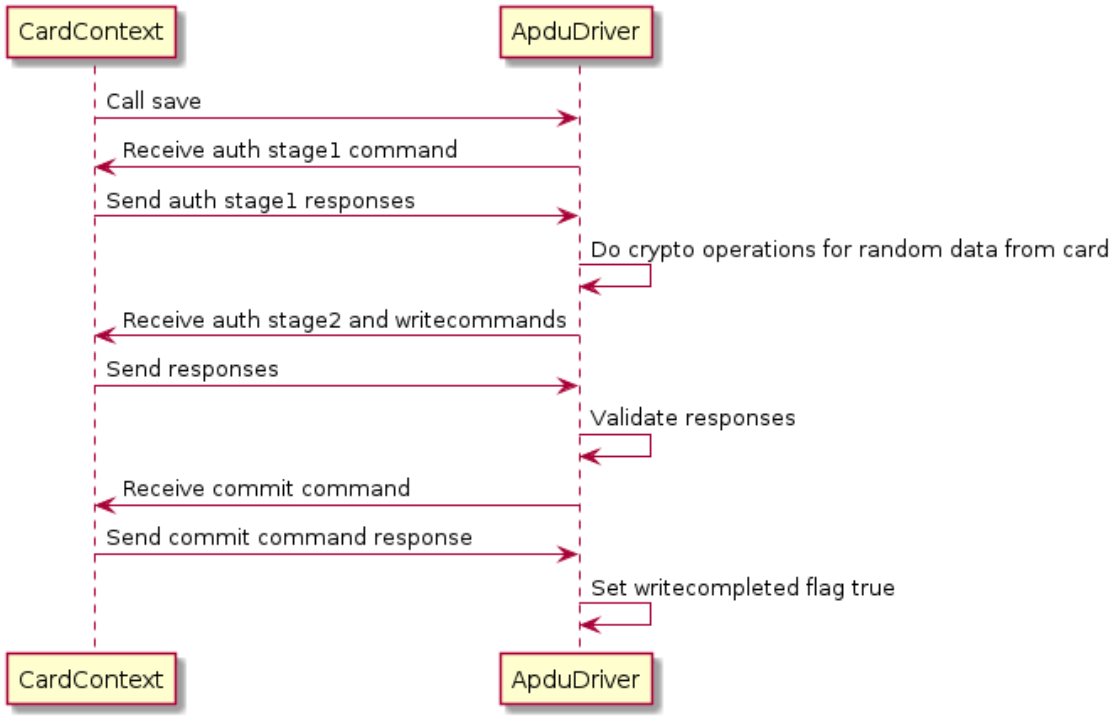
Kortin lukeminen on aina ensimmäinen operaatio, mitä kortille tehdään. Lukukomentojen sisältö käsitellään ja sisältö kopioidaan kortin käsittely rakenteen sisälle. Lukuoperaatio vaatii kokonanisuudessaan vain yhden menopaluun verran liikennettä, joten se on todella nopea ja suoraviivainen. (Kuva 8)



Kuva 8 Esimerkki lukutilanteesta APDU-ajurin kanssa.

MIFARE Classic -kortille kirjoittaminen on monimutkaisuudeltaan ajurin kannalta samaa tasoa kuin sen lukeminen, mutta DESFire -kortilla kirjoitusoperaatio on huomattavasti monimutkaisempi. Classic-kortilla authentikaatio voidaan suorittaa yhdellä kerralla, mutta DESFire -kortilla tämä on kaksivaiheinen. DESFirekortille kirjoitus aloitetaan samalla tavalla kuin Classic-kortille kirjoittaminenkin eli kutsumalla APDU-ajurin save-metodia. Kutsu antaa vastauksena autentikoinnin ensimmäisen vaiheen komennon. Kun komento on suoritettu, vastaus lähetetään takaisin ajurille. Ajuri tekee sille annettujen

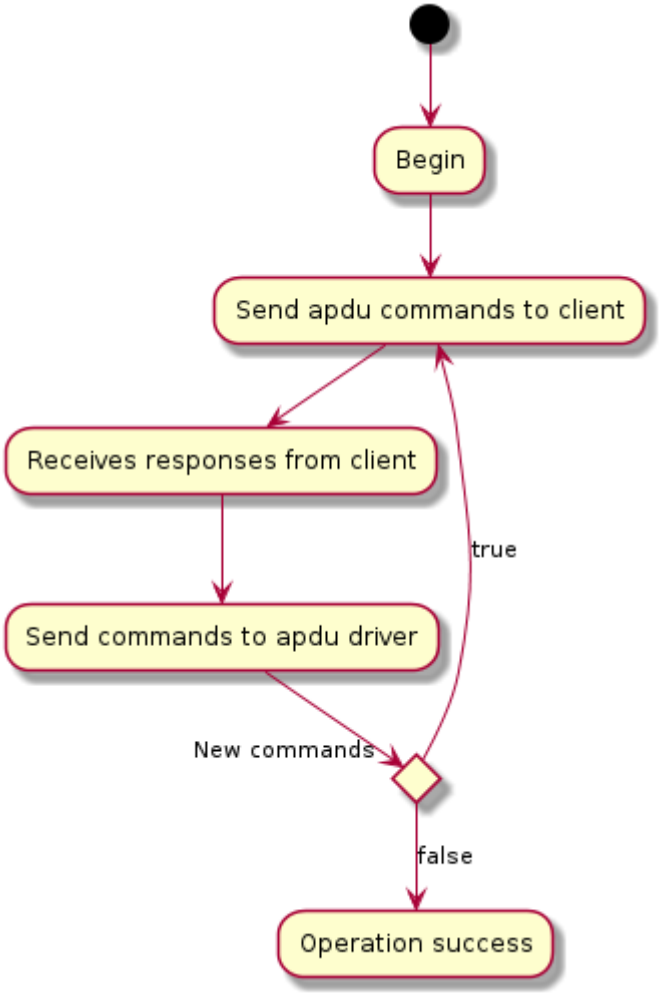
avaimien kanssa salausoperaation tälle datalle ja salattu data lähetetään takaisin kortille. Kortti tekee samanaikaisesti saman operaation ja tarkistaa menikö operaatio oikein. Jos meni, autentikaatio on onnistunut ja seuraavat kirjoituskomennot onnistuvat. Kirjoitus-operaatioiden validoinnin jälkeen kortille lähetetään commit-komento, joka tekee muutoksista pysyviä kortilla. (Kuva 9)



Kuva 9 DESFire-kortin tallennusoperaation sekvenssikaavio.

APDU-ajurilla on sendResponses-metodi, jolle annetaan parametrina lista puskureista, joissa on kortin antamat vastaukset komennoille. Metodi palauttaa listan komennoista, joita se haluaa kortille seuraavaksi lähetettävän. Komentojen lähettelyä ja uusien vastaanottamista jatketaan niin kauan, kunnes sendResponses palauttaa tyhjän listan, jolloin sen mielestä ollaan valmiita. (Kuva 10)

APDU-ajurin toteutus on täysin synkroninen.

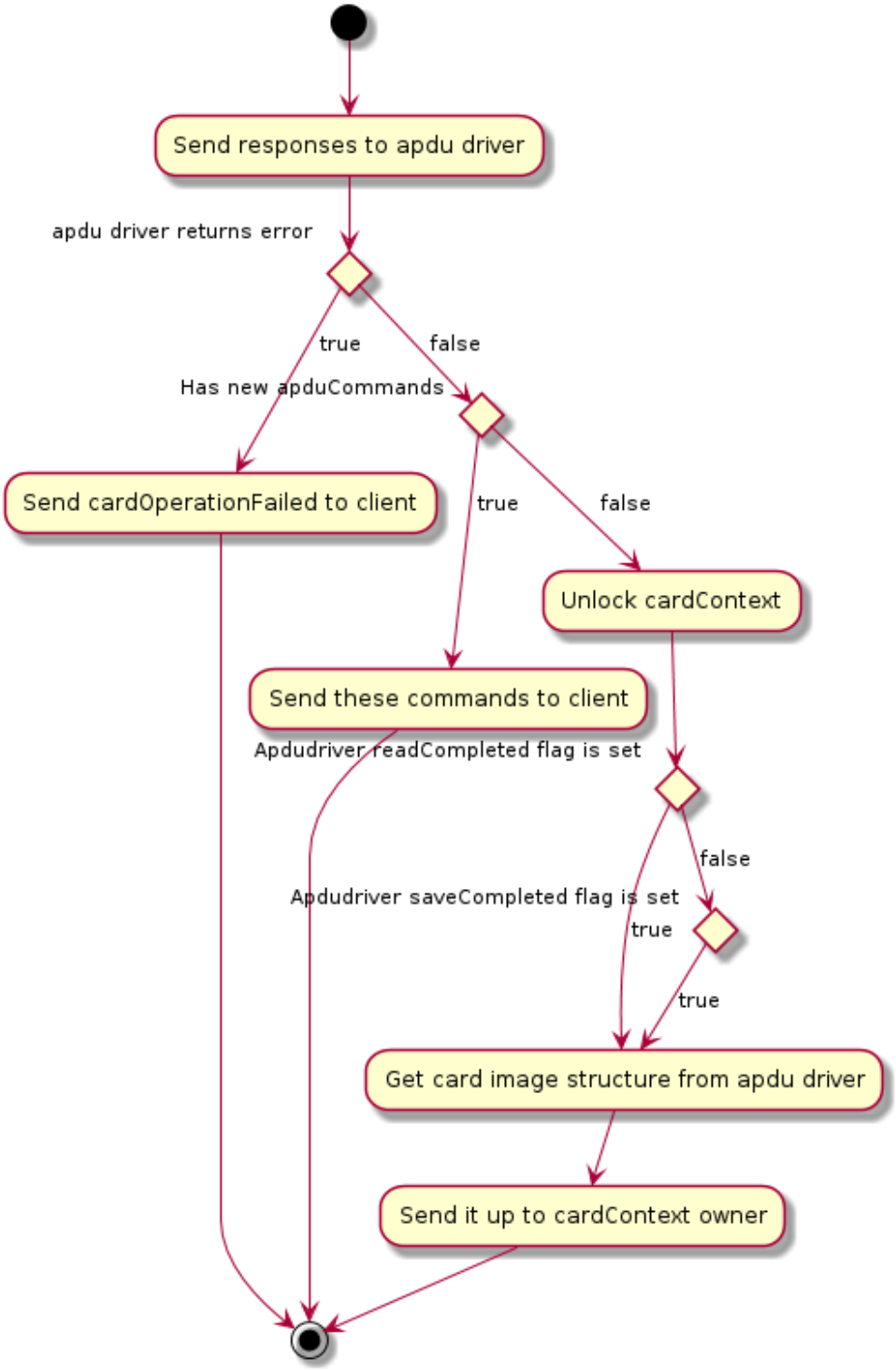


Kuva 10 Vuokaavio APDU-ajurin käytöstä.

6.2.2 Korttikonteksti

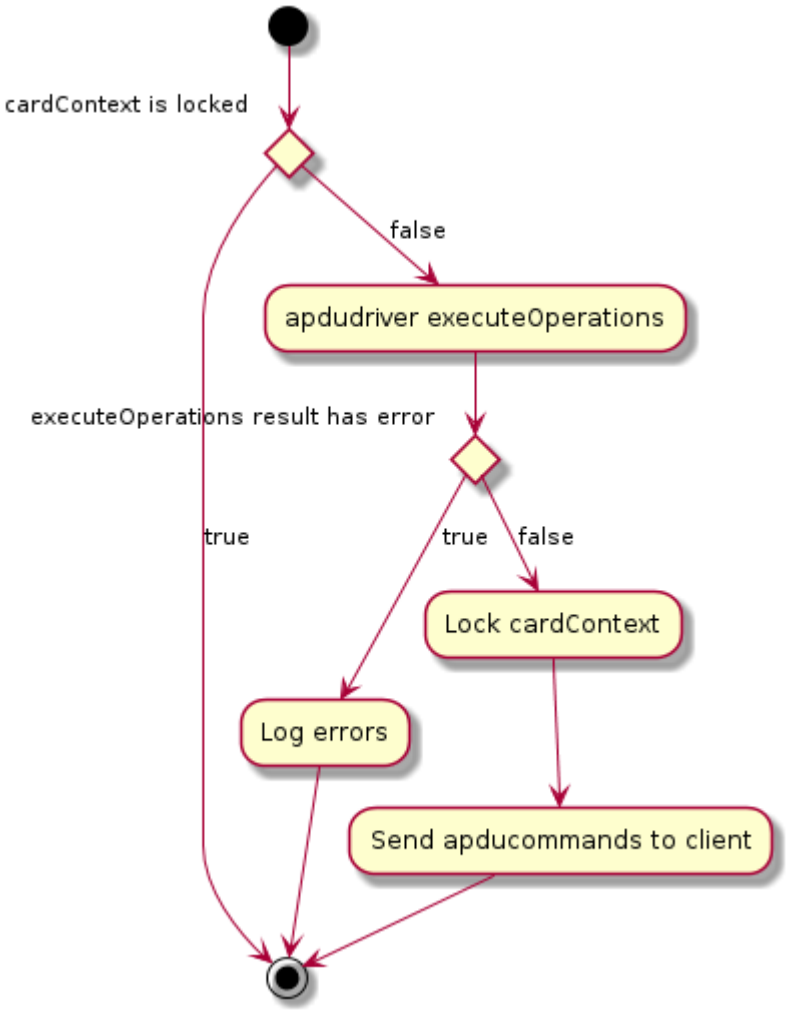
Korttikonteksti on luokka, josta luodaan instanssi jokaiselle kortille, jonka tiedot palvelimelle lähetetään. Korttikonteksti luo itselleen uuden APDU-ajuri instanssin ja pitää sitä omassa tilassaan tallessa. Korttikonteksti on APDU-ajurin wrapper-luokan kaltainen, mutta se tarjoaa kuitenkin omaa toiminnallisuuttakin.

Korttikontekstilla on APDU-ajurin tapaan handleApduResponses-metodi. Sille annetaan parametrina suoritettujen komentojen vastaukset. (Kuva 11)



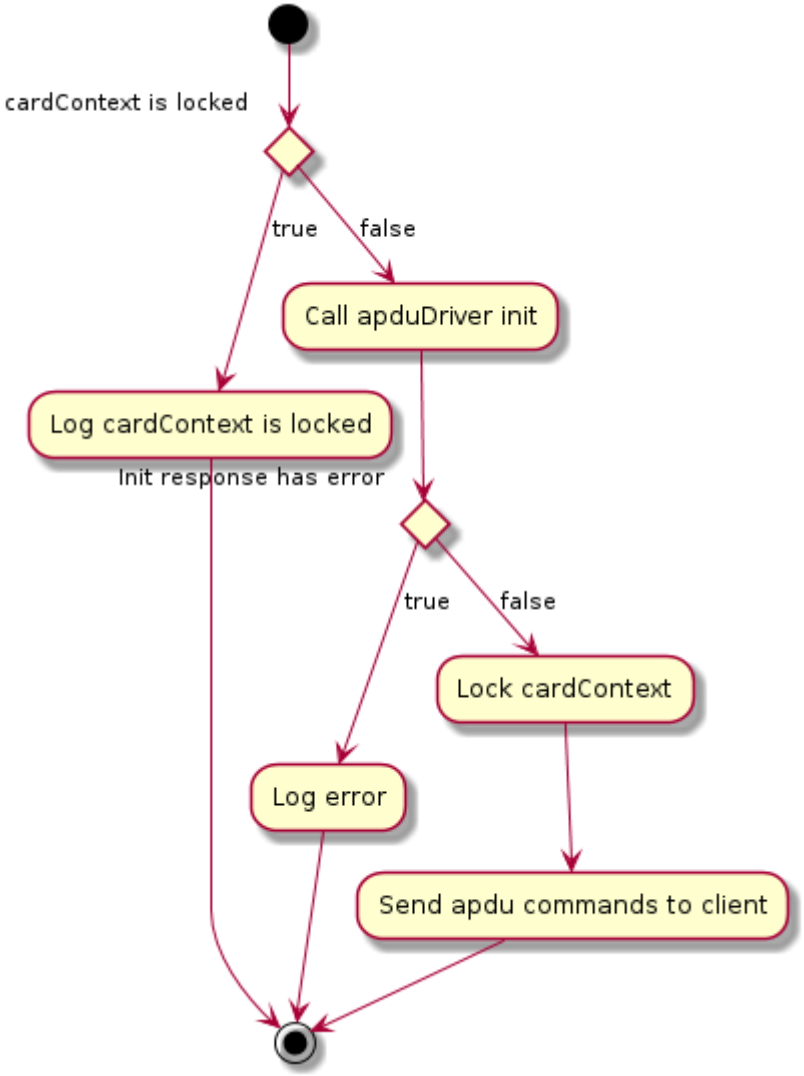
Kuva 11 Vuokaavio korttikontekstin handleResponses-metodin toiminnasta.

Korttikontekstilla on executeOperations-metodi, joka suorittaa sille annetut operaatiot ja käytännössä metodi ei itse toteuta näitä vaan käyttää siihen APDU-ajuri instanssin toteutuksia. (Kuva 12)



Kuva 12 Korttikontekstin executeOperations-vuokaavio.

Korttikontekstilla on metodi init-metodi, joka palautta listan komennoista, jotka kortin alustamisen aloittamiseen tarvitaan. (Kuva 13)



Kuva 13 Korttikontekstin init-metodin vuokaavio.

6.2.3 Lukijakonteksti

Lukijakonteksti luodaan, kun palvelimelle lähetetään tieto uudesta lukijasta. Lukijakonteksti käsittelee uusien korttien tapahtumat ja luo niistä uuden korttikontekstin ja lisää sen omaan tilaansa. Lukijakontekstin tehtävä on myös välittää APDU-komentojen vastaukset korttikontekstille ja korttikonteksti tapahtumien vieminen ylöspäin. Hyvä esimerkki tästä on uuden korttikuvan tietorakenteen vieminen ylöspäin yhteyden kontekstille. Lukijakontekstilla on huomattavasti vähemmän tehtäviä kuin korttikontekstilla ja siitä syystä sen toteutus on myös yksinkertainen.

6.2.4 Yhteyden konteksti

Yhteyden konteksti luodaan jokaiselle palvelimelle tulevalle pilvilukijayhteydelle. Se ylläpitää yhteydelle ilmoitettuja lukijakonteksteja ja tekee mappaukset lukijasta korttikonteksteille. Yhteyden konteksti luo jokaiselle palvelimelle ilmoitetulle uudelle lukijalle uuden lukijakontekstin ja säilöö sen tilaansa.

Yhteyden konteksti on se kerros, joka ymmärtää yhteyden gRPC-määrityksen. Sen tehtävänä on käsitellä asiakkaalta tulevat viestit ja tehdä oikeanlaiset operaatiot näiden perusteella.

Yhteyden kontekstin tehtävänä on myös välittää joitakin viestejä ylöspäin edelliselle kerrokselle callbackin avulla.

6.2.5 Pilvilukijanpalvelin

Pilvilukijanpalvelin on luokka, josta pilvilukijaa käyttävä ohjelma tekee itselleen instanssin sekä palvelinpuolen kirjaston pääluokka. gRPC-palvelun sisääntulopiste luodaan pilvilukijapalvelin luokan sisällä. Tämän luokan tehtävänä on palvelimeen yhdistäneiden asiakasyhteyksien ylläpitäminen. Käytännössä tämä tarkoittaa, että aina kun uusi yhteys huomataan, luodaan uusi yhteyden konteksti ja säilötään se luokan omaan tilaan. Luokalla on getCardContext-metodi, jolla korttikontekstitunnisteen avulla voidaan kyseisen tunnisteen osoittama konteksti. Tätä metodia käytetään esimerkiksi silloin, kun kirjaston käyttäjä haluaa suoritta jotakin korttikontekstia vasten operaatioita. Luokalla on myös metodi, jonka avulla koko gRPC-palvelu voidaan sammuttaa.

7 YHTEYS ASIAKKAAN JA PALVELIMEN VÄLILLÄ

Yhteyden protokollana käytetään gRPC-nimistä protokollaa. Protokolla valittiin sen tehokkuuden ja kaksisuuntaisuuden takia. Tämä protokollamääritys sisältää vain createConnection-metodin, joka luo yhteyden, jota käytetään APDU-komentojen ja vastausten lähetykseen palvelimen ja asiakaskirjaston välillä. Yhteys määritellään ja tyypitetään proto-tiedoston avulla.

Kaksisuuntaisen yhteys luodaan mahdollisimman pian asiakaskirjaston instanssin luomisen jälkeen. Tällä tavoin uusien yhteyksien luomisista johtuvat kättelyoperaatiot eivät hidasta varsinaisten lukuoperaatioiden tekemistä.

Oikeassa toteutuksessa asiakasohjelman ja palvelimen välissä on Googlen-kuormantasaaja, joka aiheuttaa sen, että protokollassa täytyy olla määritys ping-viesteille. Ping-viesti on vain viesti, jossa on yksi boolean-tyyppinen ping-kenttä. Näitä ping-viestejä lähettämällä estetään Googlen kuormantasaajaa katkaisemasta gRPC-yhteyttä. Kaksisuuntaisessa yhteydessä ping-viestejä tarvitsee lähettää kumpaankin suuntaan.

Yhteyden schema koostuu kahdesta viestityypistä palvelimelle lähetettävästä ja asiakkaalle lähetettävästä. Molemmat viestityypit sisältävät readerContextId ja cardContextId kentät. Nämä molemmat kentät asetetaan, kun halutaan tehdä operaatioita, jotka koskettavat tiettyä korttia. (Kuva 14)

cloudreader-service.proto

```
syntax = "proto3";

package Cloudreader;

message ReaderNotify {
    string name = 2;
}

message CardNotify {
    bytes atr = 3;
}

message MessageToServer {
    string readerContextId = 1;
    string cardContextId = 2;
    CardNotify cardNotify = 3;
    ReaderNotify readerNotify = 4;
    repeated bytes apduResponses = 5;
    bool readerDeparted = 6;
    bool cardDeparted = 7;
    bool ping = 8;
}

message MessageToClient {
    string readerContextId = 1;
    string cardContextId = 2;
    repeated bytes apduCommands = 3;
    bool cardReadSuccess = 4;
    bool ping = 5;
    bool cardIsCorrupted = 6;
}

service CloudreaderService {
    rpc createConnection(stream MessageToServer) returns (stream MessageToClient) {}
}

```

Kuva 14 Pilvulukijan gRPC määritys.

7.1 Viesti palvelimelle

CardNotify kertoo palvelimelle, että uusi kortti on lukijassa ja on valmis ottamaan vastaan komentoja. Viesti sisältää kortin atr-tunnisteen, jolla päätellään, minkä tyyppinen kortti ilmestyi lukijaan. ReaderNotify kertoo palvelimelle, että uusi lukija on kytketty. Tämä viesti sisältää vain lukijalle määritetyn nimen. Kortin suorittamien komentojen vastaukset tulevat palvelimelle apduResponses kentän mukana. Kun kortinlukija irrotetaan työasemasta, asiakas lähettää viestin palvelimelle, jossa readerDeparted-boolean-tyyppinen kenttä on tosi. Kun kortti on poistunut kortinlukijan kentästä ja sille ei voi enää lähettää komentoja, asiakas lähettää palvelimelle viestin, jossa cardDeparted-kenttä on tosi.

7.2 Viesti asiakkaalle

Palvelin ilmoittaa apduCommands-kentän avulla asiakkaalle, mitä komentoja se haluaa kortille lähetettävän. Palvelin ilmoittaa cardReadSuccess-kentän avulla asiakkaalle, että korttikonteksti on palvelimen puolella valmis ja sitä voidaan hyödyntää myyntien tekemiseen kortille. Palvelin ilmoittaa asiakkaalle cardIsCorrupted-kentän avulla, että kortin tietosisältö on korruptoitunut. Tämä yleensä tarkoittaa, että crc-tunnisteet kortin binääriesityksessä eivät täsmää.

8 KORTTIKOMMUNIKAATIO

Pilvilukijan kannalta on tärkeää valita oikea kommunikaatio-protokolla, joka on mahdollisimman yleiskäyttöinen asiakaskirjaston ja palvelimen välille ja joka toimii mahdollisimman pienin muutoksin usean korttityypin kanssa. APDU on kommunikaatioyksikkö älykorttilukijan ja älykortin välillä. MIFARE DESfire on älykortti, joka tukee suoraan APDU-kerroksen kommunikaatiota. APDU tuntuukin järkevältä ratkaisulta tämän kannalta. Kirjaston vaatimuksissa on olemassa olevien korttityyppien tukeminen ja MIFARE Classic on yksi niistä. MIFARE Classic on vain muistilaite, joka ei tue APDU-kerroksen komentojen käsittelyä. PS/SC-standardia tukevat lukijat sisältävät APDU-tulkin, joka osaa muuntaa nämä yksilölliset APDU-komennot Classic-kortille sopiviksi. Tällöin myös MIFARE Classic-tyyppisen kortin kanssa kommunikointiin voidaan käyttää APDU-kerroksen komentoja. Ainoana poikkeuksena tähän on Android-järjestelmä, jonka sisäinen lukija ei tue PS/SC-standardia. Tämän takia Androidille toteutetun asiakaskirjaston tarvitsee itse käsitellä nämä APDU-komennot ja muuntaa ne Classic-kortin tukemiksi operaatioiksi. Android-järjestelmä tarjoaa funktioita Classic-tyyppisten korttien käsittelyyn. Käytännössä tämä tarkoittaa, että kirjasto tutkii APDU-komennon ja päättelee sen avulla, minkä funktiokutsun tekee ja muodostaa sen paluuarvon perusteella oikeanlaisen APDU-vastauksen.

9 JOHTOPÄÄTÖKSET

Työn tavoitteena oli tehdä online-pohjaiselle myyntijärjestelmälle mahdollisuus tehdä myyntejä fyysisille korteille. Uuden järjestelmän tarvitsee olla yhteen sopiva vanhan offline-pohjaisen myyntijärjestelmän kanssa. Uuden järjestelmän luku- ja kirjoitusnopeudet pitää olla riittävän nopeita. Koska korttien käsittely tapahtuu palvelimen puolella, se on luonnollisesti hitaampaa kuin paikallinen korttien käsittely. Tämä johtuu palvelimen- ja asiakasohjelman välisestä viiveestä.

Työn vaatimusten määrittely oli helppoa ja tapahtui olemassa olevan laajan tietämyksen avulla. Määrittelyiden jälkeen oli kuitenkin epäselvää, onko vaatimukset realistisesti toteutettavissa. Epäselvyyden takia rakennettiin ensin prototyyppiversio, jonka avulla mitattiin lukunopeus ja varmistettiin sen riittävyys. Tämä oli hyvä ratkaisu, koska tällä tavoin saatiin varmistettua, että järjestelmän arkkitehtuuri on toteutettavissa ilman, että väärän arkkitehtuurin kanssa tuhlaantuu liikaa aikaa.

Tuotantoversion rakentaminen sujui kokonaisuudessaan hyvin ja siitä tuli toimiva tuote. Työ jaettiin pienempiin osiin, jotka suoritettiin osissa. Kun yksi osa saatiin valmiiksi, tarkistettiin sen toiminta ja varmistettiin, että se täyttää sille asetetut vaatimukset. Suorittamalla työn tällä tavalla ei tullut tilannetta vastaan, jossa rakennettiin ominaisuuksia väärän suunnitelman kanssa liian pitkälle. Toimintatapa muistuttaa ketterän kehityksen toimintatapaa, vaikka projektissa ei sellaista virallisesti käytettykään.

Projektin alussa viivettä pidettiin merkittävänä ongelmana, mutta pienten optimointien jälkeen se ei ollut ongelma. Järjestelmä helpottaa tulevaisuudessa puhtaaseen online-pohjaisuuteen siirtymistä ja helpottaa myös muutosten tekemistä korttien käsittelyyn, koska muutosten tekeminen palvelinohjelmaan on helpompaa kuin niiden jakaminen laitteille.

LÄHTEET

[1] NXP Semiconductors N.V. 2015. MIFARE DESFire EV1 contactless multi-application IC. Viitattu 18.4.2020 Saatavilla: https://www.nxp.com/docs/en/data-sheet/MF3ICDX21_41_81_SDS.pdf

[2] NXP B.V. 2019. MIFARE DESFire EV2 contactless multi-application IC. Viitattu 18.4.2020. Saatavilla: https://www.nxp.com/docs/en/data-sheet/MF3DX2_MF3DHX2_SDS.pdf

[3] gRPC Authors, 2020. Grpc documentation. Viitattu 26.4.2020. Saatavilla: https://grpc.io/docs/

[4] Protocol Buffers, 2020. Viitattu 26.4.2020 Saatavilla: https://developers.google.com/protocol-buffers

[5] Springcard PC/SC readers - CSB6 group, 2012. Viitattu 26.4.2020. Saatavilla: http://files.springcard.com/pub/pmd841p-fa.pdf

[6] MIFARE. Viitattu 27.4.2020. Saatavilla: https://en.wikipedia.org/wiki/MIFARE

[7] APDU. Viitattu 27.4.2020. Saatavilla: https://en.wikipedia.org/wiki/Smart_card_application_protocol_data_unit

[8] Typescript, 2020. Viitattu 28.4.2020. Saatavilla: https://www.typescript-lang.org/docs/home.html

[9] Nodejs, 2020. Viitattu 28.4.2020. Saatavilla: https://en.wikipedia.org/wiki/Node.js