TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MAURI MUSTONEN

SÄHKÖASEMAN ÄLYKKÄÄN ELEKTRONIIKKALAITTEEN VIESTIEN TILAUS JA PROSESSOINTI

Diplomityö

Tarkastaja: Professori Kari Systä

Tarkastaja ja aihe hyväksytty 8. elokuuta 2018

TIIVISTELMÄ

MAURI MUSTONEN: sähköaseman älykkään elektroniikkalaitteen viestien tilaus ja prosessointi
Tampereen teknillinen yliopisto
Diplomityö, 58 sivua, 5 liitesivua
Toukokuu 2018
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Kari Systä

Avainsanat: IEC 61850, MMS, AMQP, sähköasema, älykäs elektroniikkalaite, ohjelmistokehitys

Sähkönjakeluverkko on tärkeä osa nykyistä yhteiskuntaa ja sen päivittäistä toimintaa. Sähköverkko koostuu sähköntuotantolaitoksista, sähkölinjoista ja sähköasemista. Sähköverkon eri komponenttien avulla sähkö toimitetaan tuontantolaitoksesta kuluttajille. Sähköasemat ja niiden automatisointi ovat tärkeässä roolissa verkon yleisen toiminnan ja turvallisuuden takaamiseksi. Tässä diplomityössä keskitytään suunnittelemaan ja toteuttamaan yksittäinen ohjelmistokomponentti osaksi isompaa sähköasemiin liittyvää järjestelmää. Suunniteltavan komponentin tarkoituksena on tilata tietoa sähköasemalta verkon yli ja saada jaettua tämä tieto järjestelmän muille komponenteille. Sähköasemalta tuleva tieto on esimerkiksi mittaustietoa ja mittaustiedosta kiinnostunut järjestelmän komponentti tarvitsee tämän tiedon käyttöliittymässä näyttämiseen.

Sähköasemilta tieto tilataan *älykkäiltä elektroniikkalaitteilta* (engl. *Intelligent Electronic Device*, *IED*). IED:t ovat sähköaseman automaatiolaitteita, jotka on kytketty aseman verkkoon. Näistä käytetään myös nimitystä suojarele. IED-laitteiden kommunikointiin liittyy vahvasti maailmanlaajuinen *IEC 61850* -standardi (engl. *International Electrotechnical Commission*). Standardi määrittää kuinka IED-laitteet kommunikoivat verkon yli ja mekanismit kuinka ulkopuolinen ohjelma voi tilata siltä viestejä.

Ennen työn aloitusta ohjelmasta oli toteutettu demo, joka todisti kokonaisuuden toimivuuden. Demototeutuksessa oli kuitenkin ongelmia, jotka estivät sen käytön luotettavasti tuotannossa. Tässä työssä demoa käytettiin pohjana uuden version suunnittelulle. Demosta analysoitiin sen ongelmia ja mistä ne johtuivat. Näitä tietoja käytettiin uuden komponentin suunnitteluun liittyvissä päätöksissä.

Tuloksena työstä oli muusta järjetelmästä riippumaton ohjelmistokomponentti, joka pystyi tilaamaan viesteja IED-laitteelta IEC 61850 -standardin mukaisesti. Komponentti kykeni prosessoimaan ja jakamaan tilatut viestit järjestelmän muiden komponenttien kanssa. Komponentti päätyi tuotantoon osaksi muuta järjestelmää.

ABSTRACT

MAURI MUSTONEN: Substation's intelligent electronic device messages subscription and processing
Tampere University of Technology
Master of Science thesis, 58 pages, 5 Appendix pages
May 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Kari Systä

Keywords: IEC 61850, MMS, AMQP, IED, substation, intelligent electronic device, software development

Nowadays an electric grid is an important part of our society. It consists of power plants, power lines and substations. With these components electricity can be delivered from power plants to the end users. Substations and their automation play an important role in guaranteeing power grid safety and functionality. The focus of this master thesis is to plan and implement a software component to be a part of the bigger system which is related to substations and their management. The implemented component should be able to subscribe information from the substation and share that with other parts of the system. The information from the substation can include different types of data such as measurement data which for example can be shown on the user interface.

The information is subscribed in substations from an *Intelligent Electronic Device*, *IED* for short. An IED is an automation device which controls the other physical devices of the substation. IEDs are also connected to the substation's local network. IED can also be called with the name protection relay. The *International Electrotechnical Commission* has defined a worldwide standard called *IEC 61850* which defines the rules how IED devices should communicate with each other over the substation network. This standard also defines rules for how a software outside the substation network can subscribe information from the IED.

Before this thesis started, a proof of concept software component had already been developed. However, this component had many problems and so it was not used as a part of the system in production. Analyzing these problems was a part of this thesis. The new knowledge was then used to plan an improved version of the software component.

As a result from this thesis was a software component independent from other parts of the system. Implemented component was also able to subscribe information from the IED according to the IEC 61850 standard and share it with the other parts of the system. The component ended up being a part of the bigger system in the production environment.

ALKUSANAT

Toteutin tämän diplomityöni yritykselle nimeltä Alsus Oy. Alsus oli sen hetkinen työpaikkani vuonna 2018. Diplomintyön aihe liittyi sopivasti sen hetkisiin työtehtäviin ja sisälsi todella paljon oman mielenkiinnon kohteita. Työtehtävistä syntyi idea diplomityöstä, se muokkautui hieman ja lopulta siitä tuli tämän diplomityön aihe. Diplomityöhön liittyvän ohjelmistokehityksen aloitin jo helmikuussa 2018. Ohjelmisto valmistui toukokuussa ja siitä eteenpäin olen käyttänyt aikana työn ohessa diplomityön kirjoittamiseen.

Diplomityössäni aiheena oli suunnitella ja kehittää yksittäinen ohjelmistokomponentti osaksi isompaa järjestelmää. Järjestelmä liittyi sähköasemiin ja niiden tarkkailuun. Komponentin tarkoituksena oli tilata tietoa verkon yli sähköasemilta ja jakaa tieto järjestelmän muiden komponenttien kanssa. Tämä diplomityö on tarkoitettu luettavaksi niille, jotka ovat kiinnostuneita sähköasemiin liittyvistä ohjelmistoista tai kuinka isompaa järjestelmää hajautetaan ja tietoa siinä jaetaan.

Haluan kiittää Alsus Oy -yritystä aiheesta ja mielenkiintoisista työtehtävistä, jotka mahdollistivat tämän diplomityön. Lisäksi, että sain käyttää työaikaani vapaasti diplomityön tekemiseen ja kirjoittamiseen. Yrityksen puolelta haluan erityisesti kiittää henkilöitä Jouni Renfors ja Samuli Vainio, jotka kannustivat minua ja antoivat palautetta tämän diplomityön tekemiseen. Kiitän työni ohjaajaa professori Kari Systää työni luotettavasta ja todella hyvästä ohjaamisesta. Haluan myös kiittää läheisiä ystäviäni, joiden kanssa pidimme paljon yhteisiä kirjoitushetkiä ja rakentavia keskusteluja diplomityön tekemisestä. Ilman niitä diplomityöni kirjoitusprosessi olisi venynyt pidemmäksi. Lisäksi kiitän perhettäni tuesta ja motivaatiosta koko opiskelujen aikana mitä heiltä sain. Lopuksi kiitän muita tärkeitä ystäviäni, jotka auttoivat minua diplomityössäni oikolukemalla ja motivoimalla minua.

Tampereella, 9.8.2018

Mauri Mustonen

SISÄLLYSLUETTELO

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

ACSI	engl. *Abstract Communication Service Interface*, IEC 61850 -standardin käyttämä lyhenne kuvaamaan palveluiden abstraktimalleja
AMQP	engl. *Advanced Message Queuing Protocol* on avoin standardi viestien välitykseen eri osapuolien kesken
BRCB	engl. *Buffered Report Control Block* on IEC 61850 -standardissa puskuroitu viestien tilaamisesta vastaava luokka
CDC	engl. *Common Data Class* on IEC 61850 -standardissa joukko uudelleenkäytettäviä dataobjekin luokkia
CMV	engl. *Complex Measured Value* on IEC 61850 -standardissa dataobjektin luokkatyyppi
DA	engl. *Data Attribute* on IEC 61850 -standardissa käsite abstrahoimaan jokin sähköaseman laitteen mitattava arvo (esim. jännite)
dchg	engl. *data change* on IEC 61850 -standardissa oleva liipaisimen tyyppi
DO	engl. *Data Object* on IEC 61850 -standardissa käsite abstrahoimaan joukko samaan kuuluvia data-attribuutteja
DPC	engl. *Controllable Double Point* on IEC 61850 -standardissa dataobjektin luokkatyyppi nimeltään Pos
dupd	engl. *data update* on IEC 61850 -standardissa oleva liipaisimen tyyppi
FC	engl. *Functional Constraint* on IEC 61850 -standardissa käsite viitattujen data-attribuuttien rajoittamiseen
FCD	engl. *Functional Constrained Data* on IEC 61850 -standardissa viitteen tyyppi rajoittaamaan viitattuja data-attribuutteja hierarkiassa ensimmäisestä dataobjektista alaspäin
FCDA	engl. *Functional Constrained Data Attribute* on IEC 61850 -standardissa viitteen tyyppi rajoittamaan data-attribuutteja hierarkiassa muusta kuin ensimmäisestä dataobjektista alaspäin
FFI	engl. *Foreign Function Interface*, mekanismi, jolla ohjelma voi kutsua toisella kielellä toteutettuja funktiota
GI	engl. *General Interrogation* on IEC 61850 -standardissa oleva liipaisimen tyyppi
GIL	engl. *Global Interpreter Lock*, Ruby-kielen tulkissa oleva globaali tulkkilukitus, joka rajoittaa yhden säikeen suoritukseen kerrallaan
GVL	engl. *Global Virtual Machine Lock* on sama kuin GIL, mutta eri nimellä
HAL	engl. *Hardware Abstraction Layer* on laitteistoabstraktiotaso abstraktoimaan laitteen toiminnallisuuden lähdekoodista
IEC	engl. *International Electrotechnical Commission*, on sähköalan kansainvälinen standardiorganisaatio
IEC 61850	maailmanlaajuinen sähköasemien IED-laitteiden kommunikoinnin määrittävä standardi
IED	engl. *Intelligent Electronic Device*, sähköaseman älykäs elektroniikkalaite (myös nimellä turvarele), joka toteuttaa aseman automaatiota
IP	engl. *Internet Protocol* on protokolla verkkoliikenteessä joka huolehtii pakettien perille toimittamisesta
JRuby	on Ruby-kielen tulkki Ruby-koodin suoritukseen Java-virtuaalikoneella

JSON	engl. *JavaScript Object Notation* on JavaScript-kielessä käytetty notaatio objektista ja sen sisällöstä
JVM	engl. *Java Virtual Machine* on Java-kielen virtuaalikone Java-koodin suoritukseen
LD	engl. *Logical Device* on IEC 61850 -standardissa käsite abstrahoimaan joukko fyysisestä laitteesta joukko loogisesti yhteen kuuluvia laitteita
LN	engl. *Logical Node* on IEC 61850 -standardissa käsite abstrahoimaan fyysinen laite loogisen laitteen ryhmästä
mag	on dataobjektin instanssin data-attribuutti nimeltään mag (engl. magnitude)
MMS	engl. *Manufacturing Message Specification* on maailmanlaajuinen standardi reaaliaikaiseen kommunikointiin verkon yli eri laitteiden välillä
MMXU	engl. *measurement* on IEC 61850 -standardissa loogisen noodin luokka mallintamaan mitattuja arvoja
MRI	engl. *Matz's Ruby Interpreter* on Ruby-kielen tulkki
MV	engl. *Measured Value* on IEC 61850 -standardissa on dataobjektin luokkatyyppi
OptFlds	engl. *Optional Fields* on attribuutti viestin vaihtoehtoisten kenttien määritykseen
ORM	engl. *Object-relational Mapping* on relaatiotietokannan taulujen ja rivien abstrahointi oliopohjaisesti
PD	engl. *Physical Device* on IEC 61850 -standardissa käytetty käsite abstrahoimaan sähköaseman fyysinen laite
phsA	dataobjektin instanssi nimeltään phsA (engl. phase A) ja tyyppiä CMV
PhV	dataobjektin instanssi nimeltään phV (engl. phase to ground voltage) ja tyyppiä WYE
Pos	dataobjektin instanssi tyyppiä DPC ja nimeltä Pos (engl. position)
q	dataobjektin instanssin data-attribuutti nimeltään q (engl. quality)
qchg	engl. *quality change* on IEC 61850 -standardissa oleva liipaisimen tyyppi
RCB	engl. *Report Control Block*, raporttien konfigurointiin ja tilaukseen tarkoitettu luokkatyyppi IED-laitteelle
RoR	engl. *Ruby on Rails* on kehys web-sovellusten kehittämiseen Ruby-kielellä
SCSM	engl. *Specific Communication Service Mapping* on IEC 61850 -standardin abstrahoitujen mallien toteuttaminen jollakin tekniikalla
stVal	dataobjektin instanssin data-attribuutti nimeltään stVal (engl. status value)
t	dataobjektin instanssin data-attribuutti nimeltään t (engl. timestamp)
TCP/IP	engl. *Transmission Control Protocol/Internet Protocol*, on joukko standardeja verkkoliikenteen määrityksiin
TotW	dataobjektin instanssi tyyppiltään MV ja nimeltä TotW (engl. total active power)
TrgOp	engl. *Trigger Options* on IEC 61850 -standardissa käytetty lyhenne määritetyille liipaisimille
URCB	engl. *Unbuffered Report Control Block* on IEC 61850 -standardissa ei puskuroitu viestien tilaamisesta vastaava luokka
WYE	engl. *Phase to ground/neutral related measured values of a three-phase system* on IEC 61850 -standardissa dataobjektin luokkatyyppi

XCBR	on IEC 61850 -standardissa luokka mallintamaan sähkölinjan katkaisijaa (engl. circuit breaker)
XML	engl. *Extensible Markup Language* on laajennettava merkintäkieli, joka on ihmis- ja koneluettava
YARV	engl. *Yet another Ruby VM* on Ruby-kielen toinen tulkki, jonka tarkoitus on korvata MRI-tulkki

1. JOHDANTO

Sähköverkko koostuu tuotantolaitoksista, sähkölinjoista ja sähköasemista. Sähköasemilla on erilaisia tehtäviä verkossa. Näitä ovat esimerkiksi jännitteen muuntaminen, verkon jakaminen ja sen toiminnan tarkkailu. Lisäksi nykypäivänä asemien toiminnallisuutta voidaan seurata ja ohjata etäohjauksella. Sähköaseman yksi tärkeä tehtävä on suojata ja tarkkailla verkon toimivuutta, ja vikatilanteessa esimerkiksi katkaista linjasta virrat pois. Tällainen vikatilanne on esimerkiksi kaapelin poikkimeno, joka aiheuttaa vaarallisen oikosulkutilanteen.

Tässä diplomityössä on tarkoituksena suunnitella ja toteuttaa ohjelmistokomponentti osaksi isompaa sähköasemiin liittyvää järjestelmää. Komponentin tavoitteena on tilata tietoa verkon yli sähköaseman automaatiolaitteelta ja jakaa tieto järjestelmän muiden komponenttien kanssa. Tieto sähköasemilta tilataan tilaaja-julkaisija-arkkitehtuurimallin mukaan. Tieto voi esimerkiksi sisältää mittaustietoa jännitteestä tai fyysisen katkaisijan tilasta. Komponentin täytyy prosesoida saapunut tieto ja jakaa se siitä kiinnostuneen järjestelmän komponentin kanssa. Esimerkkinä mittaustiedosta kiinnostunut komponentti tarvitsee tiedon käyttöliittymässä olevan mittarin päivittämiseen.

Tieto tilataan sähköasemilla olevilta *älykkäiltä eletroniikkalaitteilta* (engl. *Intelligent Electronic Device*, lyhennetään *IED*). IED-laite on sähköaseman automaatiolaite jota kutsutaan myös nimellä suojarele. IED-laite voidaan kytkeä ja konfiguroida toteuttamaan monta aseman eri funktionaalisuutta ja ne on myös kytketty aseman verkkoon. IED:t voivat kommunikoida paikallisverkon yli aseman muun laitteiston ja IED-laitteiden kanssa, ja näin toteuttaa aseman toiminnallisuutta. Nykypäivänä verkon nopeus mahdollistaa reaaliaikaisen kommunikoinnin asemalla sen eri laitteiden välillä. IED-laitteet voivat myös kommunikoida aseman paikallisverkosta ulospäin, esimerkiksi keskitettyyn ohjauskeskukseen. Tämän yhteyden kautta tässä työssä toteutetun ohjelmistokomponentin on tarkoitus tilata tieto verkon yli muun järjestelmän käyttöön. Yksi IED-laite voidaan esimerkiksi konfiguroida hoitamaan sähkölinjan kytkimenä oloa, joka myös tarkkailee linjan toimintaa mittaamalla konfiguroituja arvoja, kuten jännitettä ja virtaa. Vikatilanteen sattuessa IED katkaisee linjan virrasta suurempien tuhojen välttämiseksi. Linjan korjauksen jälkeen virta kytketään takaisin päälle. [5]

IED-laitteet noudattavat kommunikoinnissa maailmanlaajuisesti määritettyä *IEC 61850* -standardia (engl. *International Electrotechnical Commission*). Standardin tarkoituksena on määrittää yhteinen kommunikointiprotokolla ja säännöt aseman kaikkien eri laitteiden välille. Tarkoituksena on ehkäistä jokaista valmistajaa tuottamasta omia versioita ja protokollia omille laitteilleen. Standardia noudattamalla eri IED-laitteet pystyvät kommunikoimaan keskenään yhteisillä säännöillä [23, s. 624]. Standardi määrittää tiedon tilaamisen

mekanismit, jolla aseman ulkopuolinen ohjelma voi tilata tiedot verkon yli. Nämä määritykset ovat tämän työn kannalta tärkein osa standardia ja on mekanismi millä komponentti tiedon asemalta tilaa. Standardi on määritetty niin, että se voi toimia monella eri teknisellä toteutuksella. Tässä työssä standardin määrityksiä käytetään *TCP/IP*-protokollaperheen päällä.

Työn tekijä oli jo ennen työn aloitusta Alsus Oy:ssä toteuttanut yksinkertaisen demoversion ohjelmasta (engl. proof of concept). Toteutus oli puutteellinen ja siinä oli toimintahäiriöitä, jotka estivät sen käytön tuotannossa luotettavasti. Demo todisti eri osien toimivuuden mahdollisuuden ja opetti tekijälle standardia. Tässä työssä demoa käytetään pohjana uuden toteutuksen suunnittelulle. Työssä analysoidaan sen toimintahäiriöitä ja mitkä ne aiheutti. Näitä tietoja käytetään pohjana uuden toteutuksen liittyvien päätöksien tekemiseen.

IEC 61850 -standardin ymmärtäminen on osa toteutusta, joten sitä käsitellään tässä työssä ennen suunnittelua ja toteutusta. Toteutettu komponentti tilasi ja jakoi tiedot onnistuneesti muiden järjestelmän komponenttien kanssa. Toteutuksessa tiedon jakamiseen käytettiin *AMQP*-standardiin (engl. *Advanced Message Queuing Protocol*) pohjautuvaa *RabbitMQ*-välityspalvelinta. AMQP-standardi käsitellään myös ennen varsinaista suunnittelua ja toteutusta.

Tämän työn tutkimustyön osuus on miettiä ja tutkia uuden toteutuksen arkkitehtuuria ja toteutusta. Arkkitehtuurin täytyy ottaa huomioon IEC 61850 -standardi ja sen asettamat rajoitteet. Tarkoitus on täyttää kaikki uudelle toteutukselle asetetut vaatimukset ja estää demoversioon liittyvät toimintahäiriöt. Työlle asetetaan tutkimuskysymyksiä, joita peilataan työn lopussa saavutettuihin tuloksiin ja pohditaan kuinka hyvin niihin päästiin. Työlle asetettiin seuraavat tutkimuskysymykset:

- *Mitkä ohjelmiston arkkitehtuurin suunnittelumallit (engl. design patterns) olisivat sopivia tämän kaltaisen ongelman ratkaisemiseen?*
- *Kuinka järjestelmä hajautetaan niin että tiedon siirto eri osapuolten välillä on mahdollista ja joustavaa?*
- *Mitkä olivat syyt demoversion toimintahäiriöihin ja kuinka nämä estetään uudessa toteutuksessa?*
- *Järjestelmän hajautuksessa, mikä olisi sopiva tiedon jakamisen muoto eri osapuolten välillä?*

2. TAUSTA

Tässä osiossa lukijaa perehdytetään työn kannalta tärkeään teoriaan ja taustatietoon. Osuuden kokonaan lukemalla lukija ymmärtää, mitä IEC 61850 -standardi tarkoittaa sähköasemien kannalta ja mihin sitä käytetään. Lisäksi kuinka standardi määrittää viestien tilauksen mekanismit ulkopuoliselle ohjelmalle ja mitä siihen liittyy. Standardi on todella laaja ja kappaleessa 2.1 siitä käsitellään vain tämän työn kannalta oleellinen asia. Loput asiat lukija voi tarkistaa standardista. Tässä työssä toteutettu komponentti jakoi viestit järjestelmälle käyttämällä AMQP-standardiin pohjautuvaa välittäjäpalvelinta. Kappaleessa 2.2 perehdytään AMQP-standardiin ja kuinka välittäjäpalvelin sen pohjalta toimii.

2.1 IEC 61850 -standardi yhteiseen kommunikointiin

Sähköasemilla nykypäivänä käytössä olevilla IED-laitteilla toteutetaan aseman toiminnallisuuden funktioita. Aseman toiminnallisuuteen liittyy sen kontrollointi ja suojaus. Aseman komponenttien suojauksen lisäksi, siihen kuuluu myös asemalta lähtevät sähkölinjat. Hyvä esimerkki sähköaseman suojauksesta on korkeajännitelinjan katkaisija, joka katkaisee virran linjasta vikatilanteissa. Tällainen vikatilanne on esimerkiksi linjan poikkimeno kaatuneen puun tai pylvään takia. Fyysistä katkaisijaa ohjaa aseman automatiikka, joka toteutetaan IED-laitteilla. IED-laite voi olla kytketty fyysisesti ohjattavaan laitteeseen [11, s. 63–64]. Koko sähköaseman toiminnallisuus koostuu monesta eri funktiosta, jotka on jaettu monelle IED-laitteelle. Jotta systeemi pystyy toimimaan, täytyy IED-laitteiden kommunikoida keskenään ja vaihtaa informaatiota toistensa kanssa. IED-laitteiden täytyy myös kommunikoida asemalta ulospäin erilliselle ohjausasemalle monitorointia ja etäohjausta varten [5, s. 1]. On selvää, että monimutkaisen systeemin ja monen valmistajan kesken tarvitaan yhteiset säännöt kommunikointia varten.

Maailmanlaajuisesti asetettu IEC 61850 -standardi määrittää sähköaseman sisäisen kommunikoinnin säännöt IED-laitteiden välillä. Standardi määrittää myös säännöt asemalta lähtevään liikenteeseen, kuten toiselle sähköasemalle ja ohjausasemalle [11, s. 10]. Ilman yhteistä standardia, jokainen valmistaja olisi vapaa toteuttamaan omat säännöt ja protokollat kommunikointiin. Seurauksena olisi, että laitteet eivät olisi keskenään yhteensopivia eri valmistajien kesken. Standardin tarkoitus on poistaa yhteensopivuusongelmat ja määrittää yhteiset säännöt kommunikoinnin toteuttamiseen [18, s. 1].

Tärkeä ja iso osa standardia on sähköaseman systeemin funktioiden abstrahointi mallien kautta. Standardi määrittää tarkasti kuinka abstraktit mallit määritellään aseman oikeista laiteista ja niiden ominaisuuksista. Tarkoituksena on tehdä mallit tekniikasta ja toteutuksesta riippumattomaksi. Tämän jälkeen määritellään kuinka mallit toteutetaan erikseen

toimivaksi jollekin tekniikalle. Abstrahoituja malleja käytetään myös määrittämään sähköaseman IED-laitteiden ja aseman muiden osien konfigurointi. Tekniikasta riippumattomien mallien ansiosta standardi on pohjana tulevaisuuden laajennoksille ja tekniikoille. Uusien tekniikoiden ilmaantuessa, voidaan standardiin lisätä osa, joka toteuttaa abstraktimallit kyseiselle tekniikalle [5, s. 2]. Tässä työssä standardin malleja ja palveluita käytettiin *MMS*-protokollan (engl. *Manufacturing Message Specification*) toteutuksella. MMS-protokolla on maailmanlaajuinen *ISO 9506* -standardi, joka on määritetty toimivaksi TCP/IP:n pinon päällä [24]. Jokainen verkkoon kytkety IED-laite tarvitsee IP-osoitteen kommunikointiin.

2.1.1 Standardin eri osat ja niiden merkitykset

IEC 61850 -standardi on laaja kokonaisuus. Tämän takia se on pilkottu erillisiin dokumentteihin, joista jokainen käsittelee omaa asiaansa. Historian saatossa standardiin on lisätty uusia dokumentteja laajentamaan standardia [16, 25] [9, s. 13]. Tämän työn kirjoitushetkellä standardiin kuului lisäksi paljon muitakin dokumentteja, esimerkiksi uusiin toteutuksiin muille tekniikoille ja vesivoimalaitoksien mallintamiseen liittyviä dokumentteja. Laajuudesta huolimatta standardin voi esittää 10:llä eri pääkohdalla ja näiden alakohdilla. Taulukossa 1 on esitetty standardin pääkohdan dokumentit ja niiden alkuperäiset englanninkieliset otsikot [16]. Tässä työssä tullaan viittaamaan standardin eri osiin, jotta lukija voi tarvittaessa etsiä tietoa asiasta tarkemmin.

Taulukko 1. *IEC 61850 -standardin pääkohtien ja niiden alakohtien dokumentit, (pohjautuu taulukkoon [23, s. 2]).*

Osa	Otsikko englanniksi
1	Introduction and overview
2	Glossary
3	General requirements
4	System and project management
5	Communication requirements for functions and device models
6	Configuration description language for communication in power utility automation systems related to IEDs
7-1	Basic communication structure - Principles and models
7-2	Basic information and communication structure - Abstract communication service interface (ACSI)
7-3	Basic communication structure - Common data classes
7-4	Basic communication structure - Compatible logical node classes and data object classes
8-1	Specific communication service mapping (SCSM) - Mappings to MMS (ISO 9506-1 and ISO 9506-2) and to ISO/IEC 8802-3
9-2	Specific communication service mapping (SCSM) - Sampled values over ISO/IEC 8802-3
9-3	Precision time protocol profile for power utility automation
10	Conformance testing

Standardin ensimmäiset osat 1–5 kattavat yleistä kuvaa standardista ja sen vaatimuksista.

Osiossa 6 käsitellään IED-laitteiden konfigurointiin käytetty *XML* (engl. *Extensible Markup Language*) -pohjainen kieli [10, s. 7–8]. Tämä osuus ei ole tämän työn kannalta tärkeä ja sitä ei sen tarkemmin käsitellä. Osat 7-1–7-4 käsittelevät standardin abstraktia mallia, niiden palveluita ja kuinka se rakentuu. Abstrahoidut palvelut ja mallit standardissa lyhennetään *ACSI* (engl. *Abstract Communication Service Interface*), ja samaa lyhennettä käytetään tässä työssä [11, s. 72]. Osissa 8–9 ja niiden alakohdissa käsitellään abstraktimallien toteuttamista erillisille protokollille, jolloin malleista tulee kyseisestä tekniikasta riippuvaisia. Tässä työssä käytettiin osaa 8-1, joka toteuttaa abstrahoidut mallit MMS-protokollalle. Osa 10 käsittelee testausmenetelmiä, joilla voidaan varmistaa standardin määritysten noudattaminen. Tämä osuus ei myöskään ole tämän työn kannalta tärkeä, ja sitä ei sen tarkemmin käsitellä. [11, s. 15]

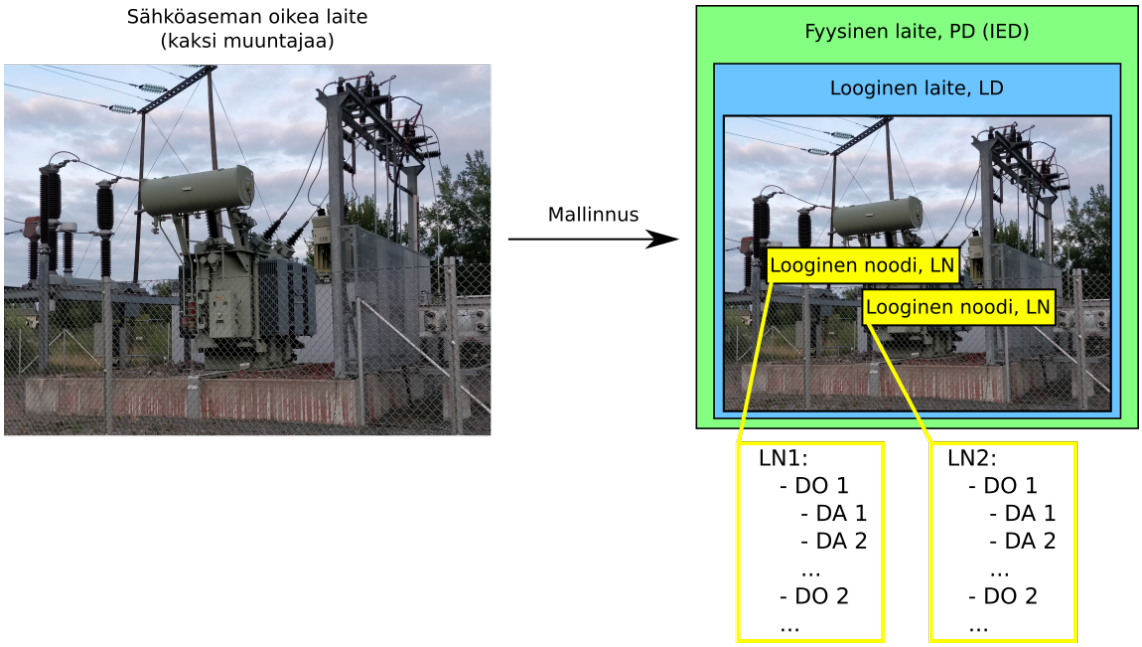
2.1.2 Abstraktimallin käsitteet ja niiden käyttö

IEC 61850 -standardin lähtökohtana on pilkkoa koko sähköaseman toiminnallisuuden funktiot pieniksi yksilöiksi. Pilkotut yksilöt abstrahoidaan ja pidetään sopivan kokoisina, jotta ne voidaan konfiguroida esitettäväksi erillisellä IED-laiteella. Yksi aseman funktio voidaan hajauttaa monelle eri IED-laitteelle. Esimerkiksi linjan suojaukseen liittyvät komponentit, katkaisija (engl. circuit breaker) ja ylivirtasuoja (engl. overcurrent protection). Toimiakseen yhdessä, laitteiden täytyy vaihtaa informaatiota keskenään verkon yli [11, s. 31]. Standardi määrittää seuraavat käsitteet sähköaseman funktioiden mallintamiseen:

- *fyysinen laite* (engl. *physical device*, lyhennetään *PD*),
- *looginen laite* (engl. *logical device*, lyhennetään *LD*),
- *looginen noodi* (engl. *logical node*, lyhennetään *LN*),
- *dataobjekti* (engl. *data object*, lyhennetään *DO*),
- *data-attribuutti* (engl. *data attribute*, lyhennetään *DA*).

Yllä listatut käsiteet muodostavat mallista hierarkisen puurakenteen ja ne on listattu hierarkisessa järjestyksessä. Hierarkian juurena on fyysinen laite, sen alla voi olla yksi tai useampi looginen laite, loogisen laitteen alla yksi tai useampi looginen noodi jne. Käsitteillä standardissa virtualisoidaan aseman funktiot, esimerkiksi suojaus. Kuvassa 1 on esitetty, kuinka sähköaseman fyysiset laitteet voidaan mallintaa standardin määrittämillä käsitteillä. Samaa periaatetta käytetään kaikille aseman laitteille. Kuvassa 1 ensin uloimpana on fyysinen laite, joka ohjaa aseman oikeita laitteita ja tarkkailee niiden toimintaa. Tämä laite voi olla IED-laite, joka on myös samalla kytketty aseman verkkoon ja sillä on IP-osoite. Yksi IED-laite voi olla samaan aikaan kytkettynä aseman moneen muuhun oikeaan laitteeseen. Tämän jälkeen mallinnetaan aseman joukko laitteita loogiseksi laitteeksi. Tällainen voi esimerkiksi olla tietyn jännitetason (engl. bay) komponentit, kuten katkaisijat, muuntajat jne. Kuvassa kaksi muuntajaa on mallinnettu yhdeksi loogiseksi laitteeksi, koska ne kuuluvat samaan jännitetasoon. Looginen laite koostuu loogisista noodeista. Looginen noodi mallintaa jotakin aseman ohjattavaa yksittäistä laitetta.

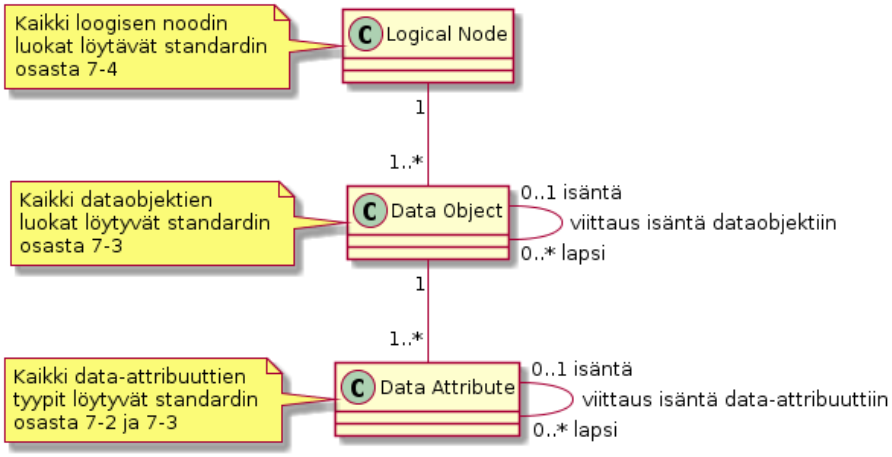
Kuvassa kaksi muuntajaa mallinnetaan loogisiksi noodeiksi. Jotta oikeaa fyysistä muuntajaa voidaan kuvata mallilla, täytyy siitä pystyä esittämään mitattavia tai kuvaavia arvoja. Tällaisia arvoja ovat esimerkiksi mitatut jännitteen arvot. Näihin tarkoituksiin käytetään käsitteitä dataobjekti ja data-attribuutti. Looginen noodi koostuu dataobjekteista ja dataobjekti koostuu data-attribuuteista. Data-attribuutti esittää yhtä mitattavaa tai kuvaavaa arvoa laitteesta, esimerkiksi sen hetkinen jännite tai laitteen tila. Dataobjekti on tapa koostaa yhteen kuuluvat data-attribuutit saman käsitteen alle, esimerkiksi mittaukseen tai ohjaukseen liittyvät data-attribuutit. [7, s. 2] [9, s. 24]



Kuva 1. *Sähköaseman fyysisten laiteiden abstrahointi IEC 61850 -standardin käsitteillä (pohjautuu kuvaan [11, s. 17]).*

IEC 61850 -standardin käsitteiden avulla sähköaseman laitteet ja funktiot voidaan esittää malleilla. Malleja voidaan käyttää IED-laitteiden konfiguroinnin määrittämiseen ja tietona, jotka voidaan siirtää verkon yli laitteelta toiselle. Jotta käsitteitä voidaan käyttää konfigurointiin ja kommunikointiin, standardi määrittää lisää tarkkuutta käsitteisiin ja kuinka niitä käytetään. MMS-protokollan kanssa fyysinen laite -käsite yksilöidään IP-osoiteella. Tämä käsite on olemassa standardissa, jotta oikea laite voidaan pitää abstraktina toteutettavasta tekniikasta. Muut käsitteet, eli looginen laite, looginen noodi, dataobjekti ja data-attribuutti määritellään standardissa luokilla tai tyypeillä. Kuvassa 2 olevassa luokkakaaviossa on esitetty kuinka käsitteet muodostavat hierarkian toisistaan ja mistä standardin osasta käsitteen mallit löytyvät. Huomiona että dataobjektit ja data-attribuutit voivat viitata itseensä. Esimerkkinä dataobjekti voi sisältää myös alidataobjektin, joka taas sisältää data-attribuutteja. [12, s. 20–22]

Looginen laite ja noodi yksilöidään nimillä, jotka ovat yksilöllisiä IED-laitteessa. Standardi asettaa rajoitteita nimeämiseen kuten pituuden. Looginen noodi esitetään IED-laitteella jonkin standardissa määritetyn luokan instanssina. Standardin osassa 7-4 määritellään val-



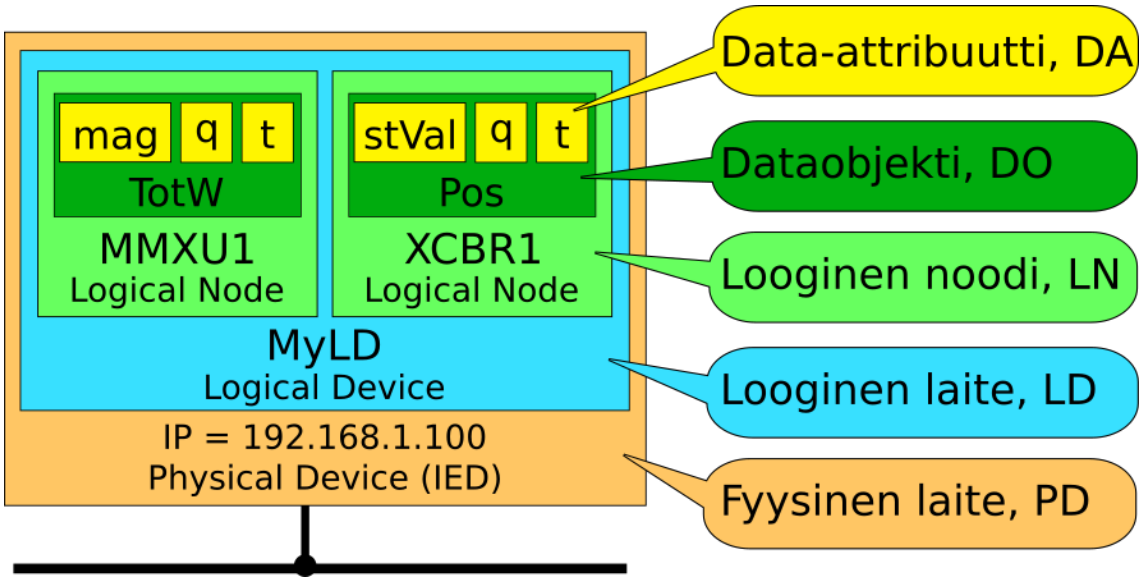
Kuva 2. *IEC 61850 -standardissa määritettyjen luokkien hierarkia (pohjautuu kuvaan [12, s. 17]).*

miita luokkia käytettäväksi eri laitteiden esittämiseen. Esimerkiksi katkaisija on määritelty luokkaan tyyppiltään *XCBR* (engl. circuit breaker) [14, s. 105–106]. Sähköaseman insinööri, joka konfiguroi IED-laitteen, määrittää konfiguraatiotiedostossa, että kytketty katkaisija esitetään XCBR-luokan instanssina ja nimeää sen standardin ohjeiden mukaan. Näin IED-laite tietää mitä laitetta se esittää ja ohjaa. IED-laitteessa kaikki eri luokkien instanssit yksilöidään nimillä ja niitä käytetään kun olioon viitataan esimerkiksi palvelukutsulla tai konfiguraatiolla. Looginen noodi koostui dataobjekteista. Standardissa dataobjektit ovat myös määritetty luokkilla, joista tehdään instansseja. Loogisen noodin luokan tyyppi määrittää mitä dataobjektin luokkia instantioidaan, ja millä nimellä ne esitettään. Toisin sanoen, aseman insinööri voi valita käytettävän loogisen noodin instansin nimen, mutta ei voi valita sen dataobjektin nimiä. Standardi määrittää dataobjektien luokkien tyypit standardissa osassa 7-3. Dataobjekti koostuu data-attribuuteista, kuten loogisen noodin luokka koostuu dataobjekteista. Dataobjektin luokka määrittää käytettävät data-attribuutit ja niiden nimet. Kuitenkaan tällä kertaa data-attribuutti ei välttämättä ole luokka. Data-attribuutit voivat olla primitiivisiä tyyppejä, kuten *integer* ja *float*. Ne voivat myös olla ns. *rakennettuja data-attribuutteja* (engl. *constructed attribute classes*), jotka pitävät sisällään tarkempia data-attribuutteja. Hyvä esimerkki on data-attribuutti nimeltään *q*, jonka tyyppi on *Quality*. Standardin mukaan tällä tyypillä on vielä aliattribuutteina mm. *validity*, *detailQual* jne [13, s. 11]. Tämä on esitetty kuvassa 2 data-attribuutin itseensä viittauksella. Myös dataobjekti voi sisältää alidataobjekteja, jonka alla on taas omat data-attribuutit. Kappaleessa 2.1.3 käydään tarkemmin läpi kuinka luokkien hierarkia standardissa rakentuu. [9, 11, 12, 13]

2.1.3 Loogisen noodin luokkien ja attribuuttien rakentuminen

IEC 61850 -standardissa kaikki luokat määritellään taulukoilla, joissa on standardoitu kentän nimi, tyyppi, selitys ja onko se valinnainen. Tässä osuudessa mennään syvemmälle luokkien määritykseen. Lisäksi esitetään esimerkkinä kuinka standardin pohjalta

instansioitu looginen noodi ja sen alla olevat dataobjektit ja data-attribuutit rakentuvat. Esimerkissä käytetään kuvan 3 rakennetta. Nimet ja luokkien instanssit konfiguroidaan IED-laitteelle XML-pohjaisella konfiguraatiotiedostolla. Tämä määritellään standardin osassa 6. Kuvassa 3 fyysinen laite on IED-laite ja siihen verkossa viitataan IP-osoitteella 192.192.1.100. IED-laitteelle on konfiguroitu looginen laite nimeltä MyLD. Eri loogiset laitteet IED-laitteella yksilöi vain sen nimi. Loogisella laitteella on kaksi instanssia loogisen noodin luokista nimillä *MMXU1* ja *XCBR1*. MMXU1-instanssi on tyyppiä *MMXU* (engl. measurement) [14, s. 57–58] ja XCBR1 on tyyppiä *XCBR* (engl. circuit breaker). Kyseessä on siis vastaavasti mittaukseen liittyvä laite ja aikaisemmin mainittu linjan katkaisija. XCBR1 loogisella noodilla on dataobjekti nimeltään *Pos* (engl. position), joka on tyyppiä *DPC* (engl. controllable double point). Ja MMXU1 nimeltään *TotW* (engl. total active power), joka on tyyppiä *MV* (engl. measured value). Loogisilla noodeilla on määritetty enemmänkin dataobjekteja eri nimillä, mutta kuvassa 3 on esitetty vain yhdet yksinkertaisuuden takia. Pos-dataobjektilla on data-attribuutit nimeltään *stVal*, *q* ja *t*. Ja TotW-dataobjektilla on data-attribuutit *mag*, *q* ja *t*. Esimerkin data-attribuutti q on tyyppiä Quality, jolla on alidata-attribuutteja ja attribuutti StVal on tyyppiä boolean. [13, 14]



Kuva 3. *Standardin käsitteiden hierarkinen rakenne ja niiden nimeämisen esimerkki (pohjautuu kuvaan [9, s. 24]).*

Standardissa osassa 7-4 on lista kaikista sen määrittämistä loogisen noodin luokista eri tarkoituksiin. Taulukossa 2 on esitetty XCBR-luokan määritys. Taulukosta voi nähdä luokan instanssille määritetyt kenttien nimet ja viimeinen sarake M/O/C, kertoo onko kenttä pakollinen (Mandatory, M), valinnainen (Optional, O), vai ehdollinen (Conditional, C). Taulukosta voi nähdä kuvan 3 esimerkin XCBR1-instanssin dataobjektin nimeltä Pos ja sen tyypin DPC. Standardissa dataobjektien luokkia kutsutaan *yleisiksi luokiksi* (engl. *Common Data Class*, lyhennetään *CDC*). Dataobjektin luokkia voidaan käyttää rakentamaan monta eri loogisen noodin luokkaa. Tämän takia dataobjektin luokkia standardissa kutsutaan nimellä *yleiset luokat*. Dataobjektin luokkien on tarkoitus kerätä yhteen samaan

asiaan liittyvät data-attribuutit. CDC-luokkien määritykset löytyvät standardin osasta 7-3 [9, s. 26]. Joillakin CDC-luokkien attribuutteina voi olla vielä muita CDC-luokkia. Tällöin standardissa puhutaan *yleisistä aliluokista* (engl. *sub data object*). Tämä esiteltiin myös aikaisemmin kuvassa 2 olevalla dataobjektin itseensä viittauksella. Esimerkkinä tästä on CDC-luokka *WYE*, jolla on attribuuttina *phsA* niminen kenttä, joka on tyyppiä *CMV*. CMV on CDC-luokka, jolla on taas omat data-attribuuttinsa. [12, s. 51,61] [13, s. 36]

***Taulukko 2**. IEC 61850 -standardin katkaisijaluokan XCBR -määritys (pohjautuu taulukkoon [14, s. 105–106]).*

Dataobjektin nimi	Englanniksi	CDC-luokka	M/O/C
Selitys			
EEName	External equipment name plate	DPL	O
Tila informaatio			
EEHealt	External equipment health	ENS	O
LocKey	Local or remote key	SPS	O
Loc	Local control behaviour	SPS	M
OpCnt	Operation counter	INS	M
CBOpCap	Circuit breaker operating capability	ENS	O
POWCap	Point on wave switching capability	ENS	O
MaxOpCap	Circuit breaker operating capability	INS	O
Dsc	Discrepancy	SPS	O
Mitatut arvot			
SumSwARs	Sum of switched amperes, resettable	BRC	O
Kontrollit			
LocSta	Switching authority at station level	SPC	O
Pos	Switch position	DPC	M
BlkOpn	Block opening	SPC	M
BlkCls	Block closing	SPC	M
ChaMotEna	Charger motor enabled	SPC	O
Asetukset			
CBTmms	Closing time of breaker	ING	O

Taulukossa 3 on DPC-luokan määritys. DPC-luokan instanssi esiintyy nimellä Pos ja on myös XCBR-luokan attribuutti. Taulukosta voi nähdä kuvan 3 esimerkissä esitetyt data-attribuutit stVal, q ja t ja niiden tyypit. Attribuuttien tyyppejä on paljon enemmänkin ja lukija voi tarvittaessa tarkistaa kaikki tyypit standardista. Tällä periaatteella standardi rakentaa kaikki muutkin luokat hierarkisesti. Määrityksien avulla voidaan selvittää mitä dataobjekteja looginen noodi sisältää, ja mitä data-attribuutteja dataobjekti sisältää. Taulukossa 3 on myös määritetty data-attribuuttien *funktionaaliset rajoitteet* (engl. *Functional Constraint*, lyhennetään *FC*), sekä mahdolliset *liipaisimet* (engl. *trigger options*, lyhennetään *TrgOp*). Funktionaaliset rajoitteet käsitellään tarkemmin kappaleessa 2.1.5 ja liipaisimet kappaleessa 2.1.6.

Kaikkien yllämainittujen luokkien kenttien määritysten lisäksi standardi määrittää palveluita jokaiselle luokkatyypille erikseen. Palvelut ovat abstrahoituja rajapintafunktiota, joille määritellään pyynnöt ja vastaukset. Standardin osa joka määrittää tekniikan toteu-

Taulukko 3. *IEC 61850 -standardin DPC-luokan määritys ja instanssin nimi on Pos (pohjautuu taulukkoon [13, s. 44]).*

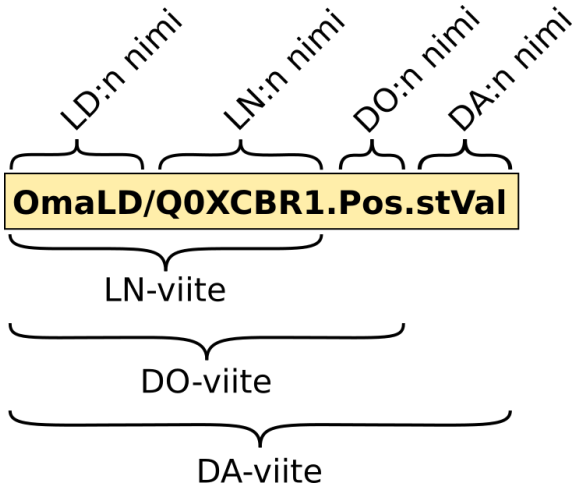
Data-attribuutin nimi	Tyyppi	FC	Liipaisin (TrgOp)
Tila ja ohjaus			
origin	Originator	ST	
ctlNum	INT8U	ST	
stVal	CODEC ENUM	ST	dchg
q	Quality	ST	qchg
t	TimeStamp	ST	
stSeld	BOOLEAN	ST	dchg
opRcvd	BOOLEAN	OR	dchg
opOk	BOOLEAN	OR	dchg
tOpOk	TimeStamp	OR	
Vaihtoehtoinen ja estäminen			
subEna	BOOLEAN	SV	
subVal	CODED ENUM	SV	
subQ	Quality	SV	
subID	VISIBLE STRING64	SV	
blkEna	BOOLEAN	BL	
Asetukset, selitys ja laajennos			
pulseConfig	PulseConfig	CF	dchg
ctlModel	CtlModels	CF	dchg
sboTimeOut	INT32U	CF	dchg
sboClass	SboClassses	CF	dchg
operTimeout	INT32U	CF	dchg
d	VISIBLE STRING255	DC	
dU	UNICODE STRING255	DC	
cdcNs	VISIBLE STRING255	EX	
cdcName	VISIBLE STRING255	EX	
dataNs	VISIBLE STRING255	EX	

tuksen, määrittää myös kuinka palvelut sillä toimivat. Tässä työssä esimerkkinä MMS-protokollan määritys, eli osa 8-1. Esimerkkinä palveluista kaikille dataobjekteille on mm. *GetDataValues*, joka palauttaa kaikki dataobjektin attribuuttien arvot. *SetDataValues* kirjoittaa annetut data-attribuuttien arvot. Ja *GetDataDirectory* palauttaa kaikki data-attribuuttien viitteet kyseisessä dataobjektista. Näitä ja muita abstrahoituja malleja viitataan standardissa lyhenteellä *ACSI* (engl. *Abstract Communication Service Interface*). [12, s. 15, 45–46] [11, s. 26]

2.1.4 Attribuuttien viittaus hierarkiassa

IEC 61850 -standardi määrittää erilaisia palvelukutsuja eri luokkatyypeille. Jotta kutsuja voitaisiin tehdä verkon yli IED-laitteelle ja sen arvoja lukea ja asettaa hierarkiassa, pitää tiettyyn data-attribuuttiin tai dataobjektiin voida viitata yksilöivästi. Siksi standardissa on määritetty viittausformaatti, jota käytetään kun IED-laitteelle tehdään kutsuja. Kutsussa olevan viitteen perusteella IED-laite tietää, mihin instanssiin kutsu kohdistuu ja pystyy

toimimaan sen mukaan. Tärkeää on myös mainita, että yhdellä määritetyllä kutsulla voidaan lukea tai kirjoitaa monta data-attribuuttia. Kutsuja ei ole rajoitettu käsittelemään yhtä data-attribuuttia kerrallaan. Viitteen lisäksi aikaisemmin mainittu funktionaalinen rajoite kertoo mihin data-attribuutteihin kutsu kohdistuu. Tämä tullaan käsittelemään tarkemmin kappaleessa 2.1.5. Kuvassa 4 on esitetty kuinka standardi määrittää viitteen muodostumisen loogisesta laitteesta data-attribuuttiin asti. [23, s. 625–626]



Kuva 4. *IEC 61850 -standardin määrittämä viitteen rakenne (pohjautuu kuvaan [11, s. 93]).*

Viite muodostuu suoraan laitteessa olevien luokkien instanssien nimien ja hierarkian mukaan. Loogisen laitteen (*LD*) ja loogisen noodin (*LN*) erottimena käytetään *kauttaviivaa* (/), ja muiden osien erottimena käytetään *pistettä* (.). Loogisella laitteella on aseman insinöörin määrittämä alle 65-merkkinen nimi. Muuten loogisen laitteen nimeen standardi ei puutu. Loogisen noodin instanssin nimi koostuu alku-, keski- ja loppuosasta. Alkuosan voi insinööri itse päättää. Esimerkkiksi kuvassa 4 loogisen noodin nimestä Q0 on alkuosa. Nimen täytyy alkaa kirjaimella, mutta se voi sisältää myös numeroita. Keskiosan täytyy olla loogisen luokan nimi, josta instanssi on tehty. Tässä tapauksessa jo aikaisemmin mainittu katkaisijan luokka, XCBR. Tämä osuus on aina 4 kirjainta pitkä ja aina isoilla kirjaimilla. Loppuosa on instanssin numeerinen arvo, joka ei sisällä kirjaimia. Insinööri voi itse päättää loppuosan, jonka ei tarvitse välttämättä olla juokseva numero. Esimerkiksi kuvassa 4 loogisen noodin nimen loppuosa on 1. Alku- ja loppuosan yhteenlaskettu merkkien pituus täytyy olla alle 13 merkkiä, eli koko loogisen noodin nimen pituus voi olla maksimissaa 17 merkkiä. Dataobjektien (*DO*) ja attribuuttien (*DA*) niminä käytetään standardin määrittämiä nimiä, jotka määritetään niitä vastaavissa luokissa osissa 7-3 ja 7-4 (katso taulukot 2 ja 3). Riippuen viittauksesta, näistä muodostuu loogisen noodin viite, dataobjektin viite ja data-attribuutin viite. Dataobjekti voi pitää sisällään toisen dataobjektin kuten aikaisemmin kuvassa 2 esitettiin. Viittausta jatketaan liittämällä instanssien nimiä toisiinsa pisteellä aina data-attribuuttiin asti. Samoin toimitaan kun data-attribuutti on tyypiltään rakennettu tyyppi, kuten Quality, jolla on alidata-attribuutteja. [12, s. 181–182] [11, s. 93–95]

Standardissa määritetään kaksi näkyvyysaluetta (engl. scope) viittaukselle, jotka ovat palvelin- ja looginen laite -näkyvyysalueet. Palvelin tarkoittaa tässä yhteydessä verkkoon kytkettyä laitetta, eli IED-laitetta. Palvelimen näkyvyysalueelle viitataan ottamalla viittauksesta pois loogisen laitteen nimi. Eli kuvassa 4 viittaus tulisi muotoon /Q0XCBR1.Pos.stVal. Edellä mainittua viittausta käytetään silloin, kun loogisen noodin instanssi sijaitsee loogisen laitteen ulkopuolella, mutta kuitenkin palvelimella. Loogisen laitteen näkyvyysalueessa viittaus sisältää loogisen laitteen nimen ennen kauttaviivaa, toisin kuin palvelimen näkyvyysalueessa. Esimerkiksi kuvassa 4 oleva viittaus OmaLD/Q0XCBR1.Pos.stVal. Loogisen laitteen näkyvyysaluetta käytetään silloin kun loogisen noodin instanssi sijaitsee loogisen laitteen sisällä sen hierarkiassa. Tässä työssä jatkossa käytetään pelkästään loogisen laitteen -näkyvyysaluetta. [12, s. 183]

Standardi määrittää viittausten maksimipituuden. Pituusmääritykset ovat voimassa kummallekin edellä mainitulle näkyvyysalueelle. Ennen kauttaviivaa saa olla maksimissaan 64 merkkiä. Tämän jälkeen kauttaviiva, josta seuraa uudelleen maksimissaan 64 merkkiä. Eli koko viittauksen maksimipituus saa olla enintään 129 merkkiä, kauttaviiva mukaan lukien. [12, s. 24,183]

2.1.5 Attribuuttien funktionaalinen rajoite ja niistä muodostetut datajoukot

Standardin CDC-luokat määrittävät käytettävät data-attribuutit (katso taulukko 3). Nämä luokat määrittävät myös jokaiselle data-attribuutille aikaisemmin mainitun *funktionaalisen rajoitteen* (engl. *functional constraint*, lyhennetään *FC*). Funktionaalinen rajoite kuvaa attribuutin käyttötarkoitusta, ja sen mitä palveluita attribuuttiin voidaan käyttää. Esimerkiksi kaikilla attribuuteilla, jotka liittyvät laitteen tilaan on funktionaalinen rajoite *ST* (engl. *status information*). Standardi määrittää paljon erilaisia funktionaalisia rajoitteita, jotka ovat kaikki kahden ison kirjaimen yhdistelmiä. Taulukossa 4 on esitetty joitain tärkeimpiä funktionaalisia rajoitteita. Funktionaalinen rajoite määrittää myös, onko attribuutti kirjoitettava tai luettava. [12, s. 53–55]

Taulukko 4. *Osa IEC 61850 -standardin määrittämistä funktionaalisista rajoitteista, lyhennetään FC (ote taulukosta [12, s. 54]).*

Lyhenne	Selite	Luettava	Kirjoitettava
ST	Laitteen tilatieto (status)	Kyllä	Ei
MX	Mittaustieto (measurands)	Kyllä	Ei
CF	Laitteen asetusarvo (configuration)	Kyllä	Kyllä
DC	Selitystieto (description)	Kyllä	Kyllä

Funktionaalista rajoitetta käytetään IED-laitteelle tehdyssä kutsussa viitteen kanssa rajoittamaan mitä data-attribuutteja tehty kutsu koskee. Tästä tulee nimi funktionaalinen rajoite. Funktionaalinen rajoite on pakollinen tieto kutsuissa, jotka lukevat tai kirjoittavat arvoja. Seuraavaksi esitetään esimerkki, kuinka yhdellä kutsulla viitataan moneen data-

attribuuttiin. Esimerkkinä otetaan kuvassa 4 olevasta viitteestä osa, joka viittaa dataobjektiin. Eli OmaLD/Q0XCBR1.Pos, jolloin viite on DO-viite. Kutsun vaikutusalue on aina hierarkiassa alaspäin. Eli nyt viitteellä viitataan Pos-dataobjektin kaikkiin alla oleviin data-attribuutteihin. Katso taulukko 3, jossa on esitetty kaikki Pos-dataobjektin alla olevat data-attribuutit. Huomiona, jos viittauksen alla olisi alidataobjekteja, niidenkin data-attribuutit kuuluisivat viittauksen piiriin. Viittauksen vaikutuksen voi siis ajatella jatkuvan viittauskohdasta alaspäin hierarkiassa kaikkiin ali-instansseihin. Funktionaalista rajoitetta käytetään rajoittamaan/suodattamaan kaikista viitatuista data-attribuuteista ne, jotka halutaan kirjoittaa tai lukea. Esimerkkinä jos kutsuun viitteellä OmaLD/Q0XCBR1.Pos lisättäisiin funktionaalinen rajoite ST, rajoitettaisiin kutsu koskemaan Pos-dataobjektin alidata-attribuuteista vain niitä attribuutteja, joilla on funktionaalinen rajoite ST. Taulukossa 5 on esitetty esimerkkinä ne data-attribuutit joita kutsu viittaisi. Taulukon attribuutit ovat samat kuin Pos-dataobjektin attribuutit taulukossa 3. Eli taulukon 5 mukaan, attribuutit olisivat origin, ctlNum, stVal, q, t ja stSeld. Muut data-attribuutit suodatetaan pois kutsun vaikutuksesta. Sama suodatus tapahtuu hierarkiassa alaspäin kaikille alidataobjekteille ja alidata-attribuuteille. Esimerkissä olevat attribuutit voisi vain lukea, ei kirjoittaa. Tämä sen takia, että taulukon 4 mukaan funktionaalinen rajoite ST sallii vain lukemisen. IEC 61850 -standardissa määritetään funktionaalinen rajoite XX, joka on sama kuin mikä tahansa muu funktionaalinen rajoite. Kuitenkin standardin osassa 8-1, joka tekee toteutuksen MMS-protokollalle, tämä ei ole tuettu toiminnalisuus. Eli toisin sanoen, jos MMS-protokollan kanssa halutaan lukea kaikki yhden dataobjektin data-attribuutit, joudutaan tekemään kutsu jokaista dataobjektin funktionaalista rajoitetta kohti. MMS-protokollan määrityksessä ei määritetä kutsua, jolla pystyisi lukemaan vain yhden data-objektin kaikki data-attribuutit. Monta erillistä kutsua tarvitaan. [12]

Viittauksen ja funktionaalisen rajoitteen avulla siis suodatetaan hierarkiassa alaspäin olevia dataobjekteja ja data-attribuutteja. IEC 61850 -standardissa on määritelty nimitykset käytettäväksi kun jotakin viittausta suodatetaan funktionaalisella rajoitteella. Nämä ovat *FCD* (engl. *Functional Constrained Data*) ja *FCDA* (engl. *Functional Constrained Data Attribute*). Nämä nimitykset ovat standardissa vain käsite ja niitä käytetään kun asiasta mainitaan. Taulukossa 6 on esitetty viittauksia eri tyyppisiin instansseihin funktionaalisella rajoitteella. Taulukosta selviää onko viitattu instanssi dataobjekti (DO) vai data-attribuutti (DA). Myös sen instanssin tyyppi ja käytetty nimitys viittaukselle FCD tai FCDA. Viitteestä käytetään FCD-nimitystä vain silloin kun hierarkian ensimmäistä dataobjekti rajoitetaan funktionaalisesti. FCDA-nimitys on käytössä kaikille muille viittauksille hierarkiassa alaspäin, joita rajoitetaan funktionaalisesti. Huomaa taulukossa 6 viittaus OmaLD/MMXU1.PhV.phsA, joka viittaa PhV dataobjekin alidataobjektiin. Tämä on FCDA-viittaus, vaikka kyseessä onkin dataobjekti. Ainoa ero FCD:n ja FCDA:n viittausten välillä on vain se, että FCD-viittaus on aina vain hierarkian ensimmäiseen dataobjektiin ja FCDA-viittaus siitä alaspäin hierarkiassa. [12, s. 55] [15, s. 63]

Funktionaalista rajoitetta käytetään viitteen kanssa suodattamaan viitatusta kohdasta alaspäin kaikki data-attribuutit. Tätä toiminnallisuutta käytetään hyväksi, kun tehdään kir-

Taulukko 5. *Pos-dataobjektista viitteellä OmaLD/Q0XCBR1.Pos ja funktionaalisella rajoitteella ST viitatut data-attribuutit.*

Data-attribuutin nimi	FC	Viittaa
origin	ST	kyllä
ctlNum	ST	kyllä
stVal	ST	kyllä
q	ST	kyllä
t	ST	kyllä
stSeld	ST	kyllä
opRcvd	OR	ei
opOk	OR	ei
tOpOk	OR	ei
subEna	SV	ei
subVal	SV	ei
subQ	SV	ei
subID	SV	ei
blkEna	BL	ei
pulseConfig	CF	ei
ctlModel	CF	ei
sboTimeOut	CF	ei
sboClass	CF	ei
operTimeout	CF	ei
d	DC	ei
dU	DC	ei
cdcNs	EX	ei
cdcName	EX	ei
dataNs	EX	ei

Taulukko 6. *Viitteen nimeäminen lyhenteellä funktionaalisen rajoitteen kanssa.*

FC	Viite	Instanssi	Tyyppi	Nimitys
ST	OmaLD/XCBR1.Pos	DO	DPC	FCD
ST	OmaLD/XCBR1.Pos.t	DA	TimeStamp	FCDA
ST	OmaLD/XCBR1.Pos.ctlNum	DA	INT8U	FCDA
MX	OmaLD/MMXU1.PhV	DO	WYE	FCD
MX	OmaLD/MMXU1.PhV.phsA	DO	CMV	FCDA
MX	OmaLD/MMXU1.PhV.phsA.t	DA	TimeStamp	FCDA

joittavia tai lukevia kutsuja ja rajoitetaan kutsulla vaikutettavia data-attribuutteja. Tätä samaa mekanismia käytetään hyväksi kun IED-laitteeseen määritellään *datajoukkoja*. IEC 61850 -standardissa datajoukko koostuu joukosta IED-laitteessa olemassa olevista data-attribuuteista. Datajoukko on tapa koostaa yhteen kiinnostavat data-attribuutit IED-laitteelta. Datajoukko nimetään ja sijoitetaan IED-laitteen hierarkiaan. Näin siihen voidaan viitata kutsuilla, kuten mihin tahansa muuhun hierarkian instanssiin. Datajoukot IED-laitteelle rakennetaan käyttämällä FCD- ja FCDA-viitteitä. Datajoukko koostuu siis joukosta FCD- ja FCDA -viitteitä. Jokaisella viitteellä on jokin funktionaalinen rajoite, joka suodattaa viitteen alla olevat attribuutit ja sisällyttää ne kyseiseen datajoukkoon. Esi-

merkkinä datajoukon rakentamisesta taulukon 6 viitteet. Näistä viitteistä voitaisiin rakentaa oikea standardin mukainen datajoukko, nimetä se nimellä Testi1, ja lisätä IED-laitteen hiearkiaan kohtaan OmaLD/LLN0.Testi1. Nyt datajoukkoon voisi viitata ja vaikka lukea kaikki sen arvot yhdellä kertaa. Jotta datajoukko saadaan näin tehtyä, tieto tästä pitäisi lisätä IED-laitteen asetustiedostoon. Datajoukkoja IED-laitteessa käytetään muodostamaan joukkoja tärkeistä data-attribuuteista, joita voidaan esimerkiksi lukea ja kirjoittaa yhdellä kutsulla. Datajoukkoja käytetään myös tilattavien *viestien sisältönä*. Viestejä voi standardin mukaan tilata vain datajoukoista olevista data-attribuuteista. [12, s. 61–68]

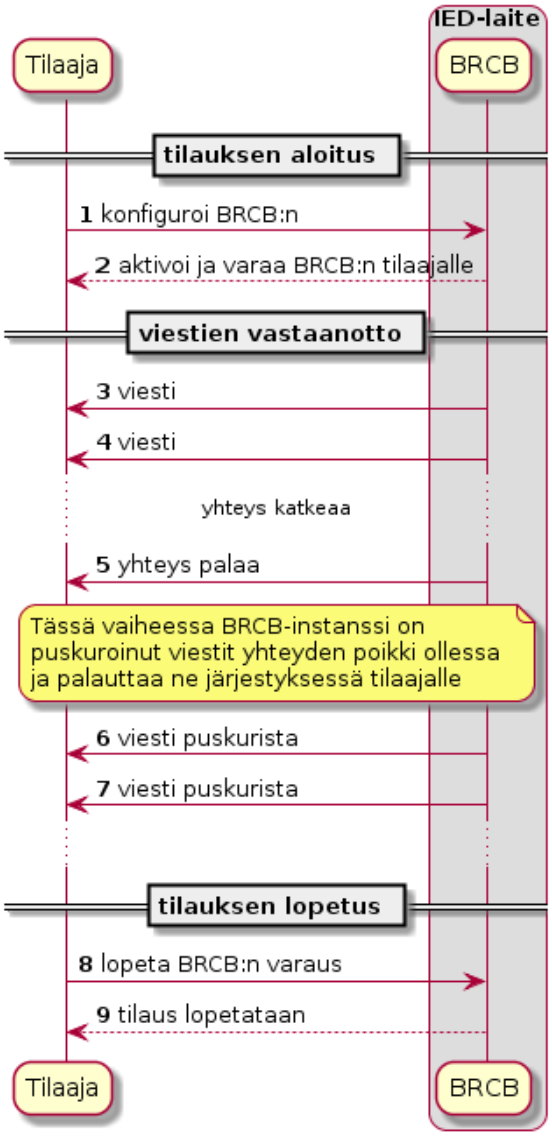
2.1.6 Viestien tilaus ja tilauksen konfigurointi

IEC 61850 -standardi määrittää, kuinka IED-laitteen ulkopuolinen ohjelma voi tilata kiinnostavien data-attribuuttien arvoja verkon yli. Viesti voidaan esimerkiksi lähettää tilaajalle, kun mitatun jännitteen arvo muuttuu. Kyseessä on tilaaja-julkaisija-arkkitehtuurimalli, jossa ulkopuolinen ohjelma on tilaaja ja IED-laite julkaisuja. Standardi määrittää, että viestejä voidaan tilata vain datajoukoissa viitatuilla data-attribuuteilta. Viestien lähetyksen tiheys riippuu siitä kuinka tilauksen yhteydessä tilaaja asettaa *liipaisimet*. Standardissa määritellään käytettäväksi erilaisia liipaisimia, joilla tilaaja voi muokata millä ehdoilla viesti pitäisi lähettää. Standardissa on myös määritetty mekanismit, joilla tilajaa voi pyytää kaikki arvot kerralla tai tilata jaksottaisia viestejä tietyn aikavälein. [11]

Standardissa määritetään luokka, jonka tehtävä on hoitaa tilausta ja sen asetuksia. Tässä kappaleessa käydään läpi luokan yleistä toiminnallisuutta. Kappaleessa 2.1.7 käsitellään luokan attribuutteja ja toimintaa tarkemmin. Niin kuin muutkin luokat standardissa, siitä tehdään instanssi, sille annetaan yksilöivä nimi ja se lisätään IED-laitteen hierarkiaan. Nämä määritellään IED-laitteen asetustiedostossa, kuten kaikki muutkin instanssit. Yksilöivän nimen avulla tilaaja voi viitata kutsulla instanssiin, muuttaa luokan asetuksia ja aloittaa tilauksen. Nämä luokat standardissa ovat *puskuroitu viestintäluokka* (engl. *Buffered Report Control Block*, lyhennetään *BRCB*) ja *ei puskuroitu luokka* (engl. *Unbuffered Report Control Block*, lyhennetään *URCB*). Tekstissä kumpaakin luokkaan viitatessa käytetään lyhennettä *RCB*. Ainoa ero luokkien toiminnan välillä on, että BRCB puskuroi viestejä jonkin aikaa yhteyden katkettua. Yhteyden palautuessa, se lähettää puskuroidut viestit järjestyksessä asiakkaalle. BRCB takaa viestien järjestyksen ja saatavuuden. URCB lähettää viestejä asiakkaalle ilman puskurointia ja viestit menetetään yhteyden katketessa. Standardissa määritetään, että yksi RCB-instanssi voi palvella vain yhtä tilaaja kerrallaan. IED-laitteeseen täytyy määrittää instansseja sen tilaajien määrän mukaan. [12, s. 93]

Sekvenssikaaviossa 5 on esitetty tilaajan ja IED-laitteella olevan BRCB-instanssin välinen viestien tilauksen prosessi. Kaaviossa ensin asiakas tilaa puskuroidun BRCB-instanssin (kohdat 1–2). Ensimmäisessä kutsussa tilaaja kirjoittaa BRCB-luokan arvot, kuten käytettävät liipaisimet jne. Kutsussa tilaajan on merkittävä RCB-instanssi varatuksi, jotta tilaus käynnistyy. BRCB aloittaa viestien julkaisun tilaajalle määritettyjen ehtojen mu-

kaan (kohdat 3–4). Jos tilaajan ja IED-laitteen välinen yhteys katkeaa, BRCB-instanssi puskuroi viestejä johonkin järkevään rajaan asti. Kun yhteys tilaajaan palaa, BRCB lähettää viestit järjestyksessä tilaajalle alkaen ensin puskurista (kohdat 5–7). Tilaaja voi lopettaa tilauksen ja instanssin varauksen merkitsemällä sen taas vapaaksi (kohdat 8–9). [11, s. 41–42]



Kuva 5. *Puskuroitu viestien tilausprosessi tilaajan ja IED-laitteella olevan BRCB-instanssin välillä (pohjautuu kuvaan [11, s. 42]).*

Standardissa määritetään, että viestejä voidaan tilata vain datajoukoista. IED-laitteen asetustiedostossa täytyy myös määrittää mitä datajoukkoa RCB-instanssi käyttää. Tämän jälkeen instanssi tarkkailee datajoukkon attribuuttien muutoksia ja lähettää viestin, jos tilaajan asettama liipaisin täsmää. Koska yksi RCB voi palvella vain yhtä tilaajaa kerrallaan, täytyy samaan datajoukkoon viitata monella eri RCB-instanssilla. Näin monta eri tilaajaa saavat viestin samasta tapahtumasta. [12, s. 93]

Standardissa on määritetty seuraavat liipaisimet data-attribuuteille, joita RCB tarkkailee

ja reagoi niihin:

- *datan muutos* (engl. *data change*, standardissa lyhenne *dchg*),
- *laadun muutos* (engl. *quality change*, standardissa lyhenne *qchg*), ja
- *datan päivitys* (engl. *data update*, standardissa lyhenne *dupd*).

Standardin luokkissa on määritetty mitä liipaisimia data-attribuutti tukee. Esimerkkinä aikaisemmin mainittu DPC-luokan määritys taulukossa 3, jossa TrgOp-sarake kertoo attribuutin liipaisimen. Datan muutos ja päivitys liipaisimien ero on, että datan päivitys liipaisee tapahtuman vaikka attribuutin uusi arvo olisi sama. Datan muutos ei liipaise tapahtumaa, jos uusi arvo on sama kuin edellinen arvo. Laadun muutos liipaisin tarkoittaa, että data-attribuuttiin liitetty laatuarvo muuttui. Laatuarvo kertoo tilaajalle ja arvojen lukijalle, voiko attribuuttien arvoihin luottaa. Laatuarvo on tyyppiä Quality ja tästä voi tarvittaessa lukea enemmän standardista. [11, s. 90]

2.1.7 Raportointi-luokan määritys ja toiminta

BRCB-luokalla on erilaisia attribuutteja, joita tilaaja voi kirjoittaa ja lukea ennen tilauksen aloittamista. BRCB ja URCB -luokat eivät eroa paljon attribuuteilla toisistaan, joten tässä kappaleessa keskitytään vain BRCB-luokan toimintaan. Tarkka määritys luokkien eroista löytyy standardin osasta 7-2. Taulukossa 7 on esitetty standardin määrittämän BRCB-luokan attribuutit, attribuutin nimi englanniksi ja sen selite. Taulukossa ei ole esitetty attribuuttien tyyppejä, koska ne voi lukija tarvittaessa tarkemmin lukea standardin omasta määrityksestä. Lisäksi tässä kappaleessa käydään läpi luokan attribuuttien toimintaa pääpiirteittäin ja loput tiedot lukija voi tarkistaa standardista. [12, s. 93–118].

Tilaaja voi vapaasti kirjoittaa ja lukea RCB-instanssin arvoja monella peräkkäisellä kutsulla ennen tilauksen aloittamista. Tärkein attribuutti on boolean tyyppinen *RptEna*. Kun attribuutti kirjoitetaan arvoon tosi, aloittaa instanssi tilauksen ja varaa sen kirjoittajalle. Tilaaja voi edelleen lukea ja kirjoittaa sen arvoja tilauksen ollessa päällä, mutta rajoitetusti. Joidenkin arvojen kirjoitus pitää tapahtua ennen tilausta tai samassa kutsussa kun RtpEna asetetaan arvoon tosi. Tilaaja lopettaa tilauksen jos yhteys on poikki tarpeeksi kauan tai RptEna kirjoitetaan arvoon epätosi.

RCB-luokan TrgOps-attribuutti on binääritietue, jossa yksittäinen bitti ilmaisee mikä liipaisin aiheuttaa viestin lähettämisen. Tällä attribuutilla tilaaja voi päättää mitä liipaisimia hän haluaa käyttää. TrgOps sisältää seuraavat liipaisimet:

- *datan muutos* (engl. *data change*, standardissa lyhenne *dchg*),
- *laadun muutos* (engl. *quality change*, standardissa lyhenne *qchg*),
- *datan päivitys* (engl. *data update*, standardissa lyhenne *dupd*),
- *yleinen kysely* (engl. *general-interrogation*, standardissa lyhenne *GI*), ja

Taulukko 7. *BRCB-luokan määritetyt attribuutit ja niiden selitteet (pohjautuu taulukkoon [12, s. 94]).*

Attribuutti	Englanniksi	Selite
BRCBName	BRCB name	Objektin nimi
BRCBRef	BRCB reference	Objektin viite
RptID	Report identifier	RCB-instanssin yksilöivä id lähetettyihin viesteihin, asiakas voi asettaa
RptEna	Report enable	Varaa RCB:n ja aloittaa viestien lähetyksen
DatSet	Data set reference	Tarkkailtavan datajoukon viite
ConfRev	Configuration revision	Juokseva konfiguraation numerointi, muutos kasvattaa numerointia
OptFlds	Optional fields	Mitä valinnaisia kenttiä viestiin lisätään
BufTm	Buffer time	Puskurointiaika, ennen viestin lähetystä. Tänä aikana tapahtuvat liipaisut yhdistetään samaan viestiin
SqNum	Sequence number	Juokseva lähetetyn viestin numerointi
TrgOps	Trigger options	Millä liipaisimilla viesti lähetetään
IntgPd	Integrity period	Periodisen viestien väli millisekunteina, arvolla 0 ei käytössä
GI	General-interrogation	Käynnistää yleiskyselyn, joka sisältää kaikki datajoukon attribuutit seuraavaan viestiin
PurgeBuf	Purge buffer	Puhdistaa lähettämättömät viestit puskurista
EntryID	Entry identifier	Puskurissa olevan viimeisimmän viestin id. Arvo 0 tarkoittaa tyhjää puskuria
TimeOfEntry	Time of entry	Puskurissa olevan viimeisimmän viestin aikaleima
ResvTms	Reservation time	Instanssin varausaika sekunteina kun yhteys katkeaa, arvo -1 tarkoittaa konfiguraation aikaista varausta ja 0 että ei varausta
Owner	Owner	Yksilöi varaavan asiakkaan, yleensä IP-osoite tai IED-laitteen nimi. Arvo 0 että RCB on vapaa tai ei omistajaa

- *jatkuva viestintä väliajoin* (engl. *intergrity*).

Kolme ensimmäistä liipaisinta dchg, qchg ja dupd ovat aikaisemmin kappaleessa 2.1.6 määrittettyjen data-attribuuttien liipaisimia. Asiakas voi tilata viestejä esimerkiksi vain datan muutoksista. RCB-luokka määrittää data-attribuuttien liipaisimien lisäksi vielä kaksi liipaisinta lisää, yleinen kysely ja jatkuva viestintä väliajoin. Yleinen kysely on viesti, johon RCB sisällyttää kaikki datajoukon attribuutit. Asiakas voi liipaista sen asettamalla luokan attribuutin GI arvoksi tosi ja TrgOps attribuutissa liipaisin on päällä. Tällöin RCB käynnistää viestin generoinnin ja lähettää sen asiakkaalle. Jos liipaisin ei ole päällä TrgOps attribuutissa, ja GI arvoksi asetetaan tosi. RCB ei generoi viestiä. Viestin lähetyksen jälkeen RCB itse asettaa GI:n arvoksi epätosi. Jatkuva viestintä liipaisin on jatkuvaa viestin lähettämistä tilaajalle väliajoin, johon sisältyy kaikki datajoukon attribuutit, kuten yleisessä kyselyssä. Toiminnon saa päälle kun asiakas asettaa RCB-luokassa attribuutit *IntgPd* arvoksi muu kuin 0, ja TrgOps-attribuutin arvossa kyseinen liipaisin on päällä. Attribuutti IntgPd kertoo minkä väliajoin viesti generoidaan ja lähetetään asiakkaalle. Jos IntgPd arvo on muu kuin 0 ja TrgOps attribuutissa liipaisin ei ole päällä, ei viestiä generoida ja lähetetä asiakkaalle väliajoin.

RCB-luokan attribuutin *OptFlds* avulla asiakas voi valita mitä vaihtoehtoisia kenttiä viestiin sisällytetään. Attribuutin OptFlds on binääritietue niin kuin ja TrgOps. Taulukossa 8 on esitetty sen asetettavat arvot [12, s. 98]. Taulukon yksittäinen kenttä vastaa OptFlds arvon yhtä bittiä. Bittien järjestys määräytyy tekniikalle toteutuksen perusteella. Esimerkiksi MMS-protokolla. Taulukon arvoilla tilaaja voi määrittää mitä lisätietoa viestiin sisällytetään. Esimerkiksi asettamalla reason-for-inclusion bitin päälle, liitetään viestin arvon yhteyteen miksi tämä arvo viestiin sisällytettiin. Viestin rakennetta ja kuinka OptFlds-attribuutin arvoilla sen sisältöön voi vaikuttaa käydään läpi tarkemmin kappaleessa 2.1.8.

Taulukko 8. *RCB-luokan OptFlds-attribuutin arvot ja niiden selitteet.*

Arvo	Selite
sequence-number	Jos tosi, sisällytä RCB-luokan attribuutti SqNum viestiin
report-time-stamp	Jos tosi, sisällytä RCB-luokan attribuutti TimeOfEntry viestiin
reason-for-inclusion	Jos tosi, sisällytä syy miksi arvo(t) sisällytettiin viestiin
data-set-name	Jos tosi, sisällytä RCB-luokan attribuutti DatSet viestiin
data-reference	Jos tosi, sisällytä datajoukon liipaisseen kohdan rakentamiseen käytetty FCD- tai FCDA-viite viestiin
buffer-overflow	Jos tosi, sisällytä viestiin tieto onko puskuri vuotanut yli kentällä BufOvfl (engl. buffer overflow)
entryID	Jos tosi, sisällytä RCB-luokan attribuutti EntryID viestiin
conf-revision	Jos tosi, sisällytä RCB-luokan attribuutti ConfRev viestiin

Lähetetyt viestit voivat sisältää vaihtelevan määrän sisällytettyjä arvoja. RCB-instanssi mittaa aikaa ensimmäisestä liipaisusta sen attribuutin *BufTm* verran ja tämän ajan jälkeen pakkaa kaikki liipaisseet attribuutit samaan viestiin. Tilaaja voi muuttaa arvoa jos haluaa käyttää pitempää tai lyhyempää puskurointiaikaa.

2.1.8 Viestin rakenne ja kuinka sen sisältö muodostuu

IED:n lähettämä viesti on rakenteeltaan hiukan monimutkainen ja lisäksi siihen vaikuttaa RCB-instanssin OptFlds-attribuutin asetetut bitit (taulukko 8). Tässä kappaleessa käsitellään viestin mallia, joka on tekniikasta riippumaton. Siitä ei tarvitse välittää minkälainen viestin rakenne on MMS-protokollan tasolla. Toteutetussa ohjelmassa käytettiin kirjastoa, joka hoitaa matalan tason asiat ja tarjoaa helppokäyttöisen rajapinnan viestin sisältöön. Kuitenkin viestin rakenteesta täytyy ymmärtää kuinka vaihtoehtoiset kentät siihen vaikuttavat ja kuinka attribuuttien arvot viestiin sisällytetään. Kuvassa 6 on esitetty standardin määrittämän viestin rakenne ja mitä kenttiä OptFlds-attribuutti kontrolloi. Viestin rakenteen voisi ajatella koostuvan kahdesta osasta. Ensin viestissä on yleinen tieto ja viimeisenä taulukko datajoukon alkioista 1–n:ään, jotka liipaisevat viestin lähetyksen.

Kuvassa 7 on esitetty yleinen kuva kahden viestin lähetyksestä liipaisun tapahtuessa. Kuvassa keskellä on kaksi BRCB-instanssia myBRCB01 ja myBRCB02, jotka tarkkailevat datajoukkoja Testi1 ja Testi2 vastaavasti. Kummatkin instanssit lähettävät viestin, jotka ovat kuvassa oikealle. BRCB-instansseista voi nähdä, mitä niille asetetut attribuuttien arvot ovat ja datajoukoista näkee mistä FCD- ja FCDA -viitteistä ne koostuvat. Kuvassa attribuutin MyLD/XCBR1.Pos.stVal arvo muuttuu ja tämä liipaisee viestin lähetyksen kummassakin BRCB-instanssissa. Viesteistä voi nähdä sen sisällön ja myös miten BRCB-instanssien OptFlds-attribuutin arvot vaikuttavat sen sisältöön. Lähetettyjen viestien rakennetta ja sisältöä voi verrata kuvassa 6 määritetyn viestin rakenteeseen. Kuvassa on esitetty myös kuinka BRCB-instansseihin viitataan MMS-protokollan tapauksessa. Tätä käsitellään tarkemmin kappaleessa 2.1.9.

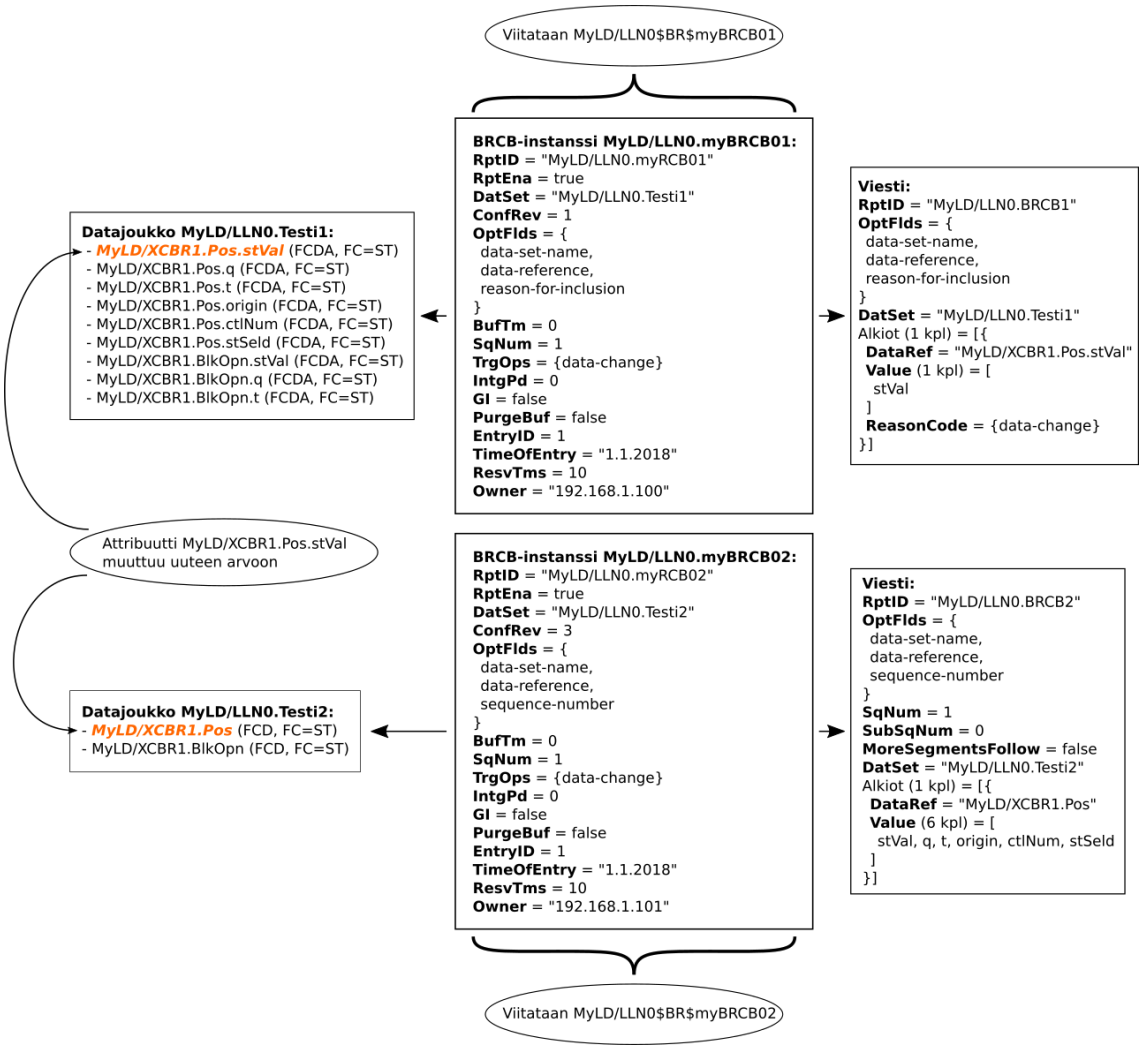
Viestissä *RptID*-kenttä sisältää viitteen RCB-instanssiin, mistä viesti on peräisin. OptFlds sisältää binääritietueen viestin vaihtoehtoisista kentistä. Viestin kenttiä *SqNum*, *SubSqNum* ja *MoreSegmentsFollow* käytetään kertomaan asiakkaalle, jos päätason viesti on liian pitkä ja se on pilkottu alaosiin. Kenttä SqNum on RCB-instanssin samanniminen kenttä ja toimii juoksevana numerointina päätason viesteille. Kenttä SubSqNum on juokseva numerointi alkaen nollasta, mikäli kyseessä on päätason viesti, eli saman SqNum arvon sisältävä viesti on pilkottu osiin. Kentän MoreSegmentsFollow ollessa tosi asiakas tietää että päätason viesti on pilkottu osiin ja seuraava osa on odotettavissa palvelimelta. Kun viestin kaikki osat on lähetetty, palvelin asettaa viimeisessä viestissä kentän MoreSegmentsFollow arvoksi epätosi ja seuraavassa päätason viestissä SubSqNum kentän arvoksi 0. Kenttä *DatSet* sisältää viitteen datajoukkoon mistä viesti on peräisin. Puskuroidussa BRCB-instanssissa kenttä *BufOvlf* kertoo onko viestipuskuri vuotanut yli. *ConfRev* kertoo juoksevan konfiguraation numeron, tämä tulee suoraan RCB-instanssin samannimisestä attribuutista. *TimeOfEntry* kertoo milloin viesti generoitiin IED-laitteen päässä. *EntryID* on viestin yksilöivä numerointi. Tämä kenttä tulee suoraan RCB-instanssin samannimisestä kentästä. Tämän jälkeen viestissä tulee taulukko, joka sisältää liipaisseet datajoukon alkiot. Jokainen taulukon alkio sisältää *Value*-kentän ja vaihtoehtoiset *DataRef* - ja *ReasonCode*-kentät. DataRef sisältää datajoukon FCD- tai FCDA-viitteen, joka

Viestin rakenteellinen sisältö		
Parametrin nimi	**Englanniksi**	**Selitys**
RptID	Report identifier	RCB-instanssin yksilöivä id.
OptFlds	Optional fields	Mitä optionaalisia kenttiä viestiin on sisällytetty
Jos sequence-number = tosi		
SqNum	Sequence number	Juokseva lähetetyn viestin numerointi
SubSqNum	Sub sequence number	Pilkotun viestin juokseva alinumerointi
MoreSegmentsFollow	More segments follow	Tosi jos samalla juoksevalla päänumerolla saapuu vielä lisää viestejä
Jos data-set-name = tosi		
DatSet	Data set	Tarkailtavan datajoukon viite
Jos buffer-overflow = tosi		
BufOvfl	Buffer overflow	Jos arvo on tosi, on viestien puskurit vuotaneet yli
Jos conf-revision = tosi		
ConfRev	Configure revision	Juokseva konfiguraation numerointi
Viestin data		
Jos report-time-stamp = tosi		
TimeOfEntry	Time of entry	Aikaleima milloin viesti generoitiin
Jos entryID = tosi		
EntryID	Entry id	Viestin yksilöivä numero
Liipaissut datajoukon alkio [1..n]		
Jos data-reference = tosi		
DataRef	Data reference	Liipaisseen datajoukon alkion FCD- tai FCDA -viite
Value	Value	Sisältää arvon tai arvot liipaisseesta datajoukon alkoiosta.
Jos reason-for-inclusion = tosi		
ReasonCode	Reason code	Syykoodi miksi tämä datajoukon kohta on sisällytetty viestiin

Kuva 6. *Standardin määrittämä lähetetyn viestin rakenne (pohjautuu kuvaan [12, s. 104]).*

liipaisi tapahtuman. ReasonCode-kenttä kertoo mikä RCB-instanssin TrgOps-attribuutilla asetetuista liipaisimista liipaisi tapahtuman ja aiheutti alkion sisällytyksen viestiin. Kentän mahdolliset arvot ovat samat kuin RCB-instanssin TrgOps-attribuutin arvot.

Value-kentän arvosta on tärkeä ymmärtää, että se voi sisältää yhden tai monta data-attribuutin arvoa. Tämä riippuu siitä, viittaako datajoukon liipaissut alkion FCD- vai FCDA-



Kuva 7. *BRCB-instanssi tarkkailee sille määritettyä datajoukoa ja generoi viestin tapahtuman liipaistessa.*

viitteellä useaan data-attribuuttiin. Viittauksen ollessa FCDA-viite, joka viittaa vain yhteen data-attribuutiin, sisältää Value-kenttä vain kyseisen data-attribuutin arvon. Jos viittaus on FCD- tai FCDA-viite joka viittaa moneen attribuuttiin hierarkiassa alaspäin. Sisältää Value-kenttä kaikki nämä arvot, vaikka niistä olisi liipaissut vain yksi attribuutti. FCD- ja FCDA-viittauksen toimintaa ja mitä attribuutteja se viittaa hierarkiassa alaspäin, käydään läpi kappaleessa 2.1.5. Esimerkki tästä on kuvassa 7, jossa liipaisu yhdessä attribuutissa aiheuttaa eri määrän arvoja kumpaankin viestiin. Tähän vaikuttaa kuinka liipaisevaan attribuuttiin on viitattu datajoukossa. Kuvassa 7 Testi1 attribuuttiin MyLD/XCBR1.Pos.stVal on viitattu FCDA-viitteellä, jossa funktionaalinen rajoite on ST. Eli FCDA-viite viittaa vain stVal-attribuuttiin, ei muihin. Tämän takia myBRCB01-instanssilta tuleva viestin Value-kenttä sisältää vain stVal-attribuutin arvon. Datajoukon Testi2 FCD-viite MyLD/XCBR1.Pos funktionaalisella rajoitteella ST sisältää kaikki Pos-instanssin ST data-attribuutit. Tämä sisältää myös muuttuneen MyLD/XCBR1.Pos.stVal data-attribuutin. Tämä aiheuttaa sen, että kaikki datajoukon viitteen MyLD/XCBR1.Pos viitatut attribuutit lisätään viestiin. BRCB-instanssilta myBRCB02 tuleva viestin Value-kenttä sisältää kaikki

viitatut attribuutit ja viestin DatRef-kenttä sisältää datajoukossa käytetyn viitteen. Pos-dataobjektin attribuutit voi tarkistaa taulukosta 3. [11, s. 40–44] [12, s. 108]

2.1.9 Abstraktimallin sovitus MMS-protokollaan

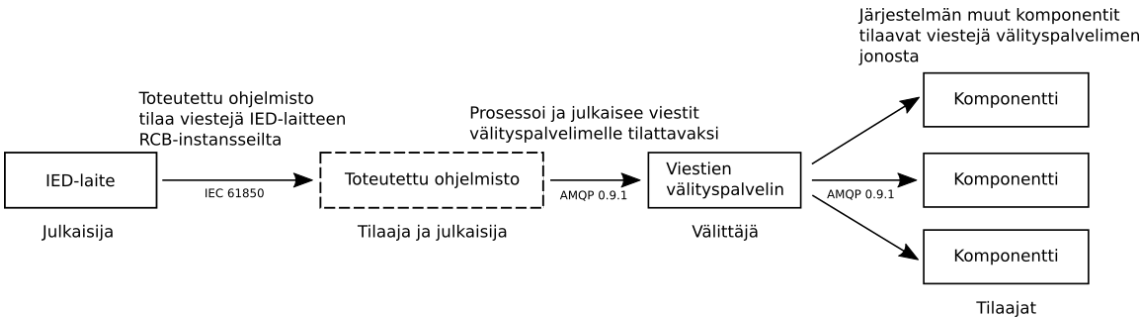
Tähän asti käsitellyt IEC 61850 -standardin mallit ja palvelut ovat olleet abstrahoituja ja tekniikasta riippumattomia. Tässä työssä käytetiin IEC 61850 -standardin MMS-protokollan toteutusta. Tästä toteutuksesta on tarkemmin määritetty IEC 61850 -standardin osassa 8-1. MMS-protokolla on maailmanlaajuinen *ISO 9506* -standardi viestintään, joka on määritetty toimivaksi TCP/IP:n pinon päällä [24]. Tämän työn kannalta lukijan ei tarvitse ymmärtää MMS-protokollaa ja sen toimintaa. Suunnitellussa ohjelmistossa käytettiin apuna kirjastoa, joka hoitaa matalan tason kommunikoinnin IED-laitteen kanssa. Tässä osiossa käsitellään työn kannalta tärkeitä tietoja, mitä toteutuksesta MMS-protokollalle kuitenkin tarvitsee tietää. [36]

IEC 61850 -standardin mallinnuksessa aikaisemmin esitetty instanssien viittaus hierarkiassa muuttuu ja nyt viittaus sisältää myös funktionaalisen rajoitteen. Esimerkkinä kuvassa 4 oleva viite OmaLD/Q0XCBR1.Pos.stVal funktionaalisella rajoitteella ST muuttuu muotoon OmaLD/Q0XCBR1STPos$stVal. Tässä viittauksessa pisteet (.) korvataan dollari-merkillä ($). Ja kaksikirjaiminen funktionaalinen rajoite sijoitetaan loogisen noodin ja ensimmäisen dataobjektin nimien väliin. Muuten viittaus säilyy identtisenä alkuperäiseen, ja samat rajoitteet ja nimeämiskäytännöt ovat voimassa edelleen. [15, s. 34–35, 111]

Tämän uuden viittauksen takia jokaiselle viitattavalle kohteelle täytyy olla funktionaalinen rajoite. Niinpä esimerkiksi RCB-luokkien instansseille täytyy olla myös funktionaalinen rajoite. Puskuroitua RCB-instanssia viitataan funktionaalisella rajoitteella *BR*. Ja puskuroimatonta funktionaalisella rajoitteella *RP*. Esimerkin tästä viittauksesta voi nähdä aikaisemmin mainitusta kuvasta 7. [15, s. 32–34, 75]

2.2 Advanced Message Queuing Protocol (AMQP)

Työssä toteutettu ohjelmisto tilasi viestit IED-laitteen RCB-instansseilta. Viestit tilattiin verkon yli aseman ulkopuolelle. Ohjelma prosessoi IEC 61850 -standardin mukaiset viestit ja lähetti ne eteenpäin välittäjälle (engl. message broker). Välittäjä on verkossa oleva erillinen palvelin, mistä muut ohjelmat pystyvät tilaamaan viestejä tarpeidensa mukaan. Kuvassa 8 on esitetty lopullisen toteutuksen tietoliikenne eri osapuolten välillä. Tässä työssä toteutettu ohjelmisto on merkitty kuvaan katkoviivalla. Toteutuksessa oli kyse tilaaja-julkaisija-arkkitehtuurimallista (engl. publish-subscribe pattern), jossa toteutettu komponentti oli tilaaja IED-laitteelle ja julkaisija välityspalvelimelle. Järjestelmän muut komponentit olivat tilaajia välityspalvelimelle. Tässä osuudessa perehdytään viestien välittäjän teoriaan, ja mitä siitä täytyy tietää ohjelmistokehityksen kannalta.



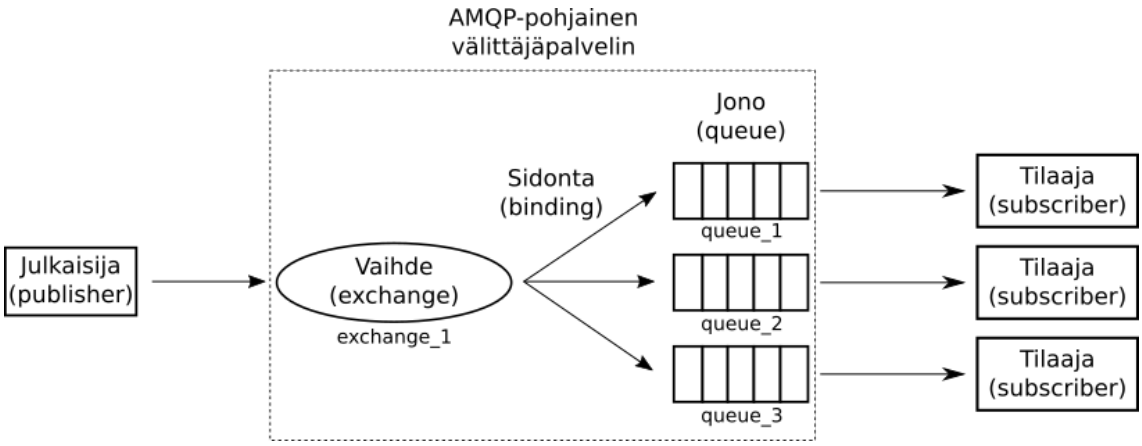
***Kuva 8**. Toteutetun ohjelmiston osuus ja rooli kokonaisuudessa tietoliikenteen kannalta.*

Työssä välittäjänä käytettiin *RabbitMQ*-ohjelmistoa [32], joka on avoimen lähdekoodin välittäjäpalvelin ja perustuu avoimeen *AMQP*-standardiin [3] (engl. *Advanced Message Queuing Protocol*). AMQP määrittää yhteisen protokollan viestintään eri ohjelmistojen välillä verkon yli välityspalvelimen avulla. Verkon ansiosta välityspalvelin voi sijaita eri koneella kuin sitä käyttävät ohjelmistot. Standarista on julkaistu monta eri versiota, ja työn tekohetkellä viimeisin versio oli 1.0. Kuitenkin RabbitMQ-ohjelmisto oli suunniteltu käytettäväksi standardin version 0.9.1 kanssa, ilman asennettuja lisäosia. Versioiden välinen ero oli suuri ja siirto uuteen ei ollut mahdollista, koska standardin versiot eivät olleet keskenään yhteensopivat. RabbitMQ tuki versiota 0.9.1 ja sen kehittäjät mieltävät standardin version 1.0 kokonaan eri protokollaksi [31]. Kuvassa 8 on tietoliikenteen kohtiin merkitty mikä standardi vaikuttaa minkäkin osapuolen kommunikointiin. Tässä työssä välityspalvelin ja siihen yhteydessä olevat ohjelmistot käyttävät AMQP-standardista versiota 0.9.1.

2.2.1 Advanced Message Queuing -malli ja sen osat

AMQP-standardi määrittää komponentteja, joiden läpi viestin täytyy kulkea julkaisijalta tilaajalle. Standardissa nämä komponentit määrittää *AMQ-malli* (engl. *AMQ-model*). Kuvassa 9 on esitetty viestin kulku julkaisijalta tilaajalle mallin eri komponenttien läpi. Mallin komponentit ovat *vaihde* (engl. *exchange*), *jono* (engl. *queue*) ja näiden välinen *sidonta* (engl. *binding*). Välityspalvelimen tehtävän voi tiivistää niin, että se ottaa vastaan viestejä julkaisijoilta vaihteeseen. Vaihde reitittää viestejä tilaajille jonoihin määritettyjen sidosten mukaan. Jos tilaaja ei ehdi prosessoida viestejä tarpeeksi nopeasti, palvelin pitää viestit jonossa tilaajalle. Vaihde voi välittää viestin moneen eri jonoon ja yhtä jonoa voi tilata monta eri asiakasta.

AMQP on ohjelmoitava protokolla siinä mielessä, että julkaisija ja tilaaja voivat määrittää komponentteja ja reitityksiä palvelimelle verkon yli ajon aikana tarpeidensa mukaan. Välittäjäpalvelin ei määritä kuin oletusvaihteet valmiiksi käytettäväksi. Toisin sanoen julkaisuja voi luoda vaihteita ja tilaaja voi luoda jonoja ja sidoksia vaihteiden ja jonojen välille. Voidaan sanoa että julkaisija ja tilaaja tekevät uusia instansseja AMQ-mallin komponenteista palvelimelle. Vaihteiden ja jonojen instansseilla täytyy olla välityspalvelimella yksilöivät nimet, jokainen nimi asetetaan instanssin luonnin yhteydessä. Esimerkkinä kuvassa



Kuva 9. *AMQ-mallin osat ja viestin kulku niiden läpi julkaisijalta tilaajalle (pohjautuu kuvaan [2, s. 11]).*

9 on AMQ-mallin komponenttien alla niille määritetyt nimet. Vaihteella on esimerkiksi nimi exchange_1 ja ylimmällä jonolla queue_1. Tällä ohjelmoitavalla ominaisuudella välityspalvelin voidaan konfiguroida toteuttamaan erilaisia skenaarioita vapaasti ja se antaa kehittäjille vapautta toteutukseen.

2.2.2 Vaihde (exchange) ja reititysavain (routing-key)

Jotta viesti voidaan kuljettaa välittäjäpalvelimen läpi, täytyy julkaisijan aloittaa määrittämällä käytetty vaihde ja sen tyyppi. Julkaisija voi myös käyttää palvelimen oletusvaihdetta. Vaihde on komponentti, joka ottaa vastaan viestejä ja reitittää niitä jonoihin vaihdetyypin (engl. exchange type) ja sidosten mukaan. Vaihteet eivät ikinä tallenna viestejä. Vaihde voi tiputtaa viestin, jos se ei täsmää minkään määritetyn reitityksen kanssa. AMQ-malli määrittää seuraavat käytettävät vaihdetyypit:

- *suoravaihde* (engl. *direct exchange*),
- *hajautusvaihde* (engl. *fanout exchange*), ja
- *aihepiirivaihde* (engl. *topic exchange*).

Näitä tyyppejä ja kuinka ne toimivat käydään tarkemmin läpi tulevissa kappaleissa. Tyypin lisäksi vaihteella on myös attribuutteina *nimi* (engl. *name*), *kestävyys* (engl. *durability*), *automaattinen poisto* (engl. *auto-delete*). Nimi yksilöi vaihteen palvelimella ja tilaaja käyttää tätä nimeä sidoksen tekemiseen jonon ja vaihteen välille. AMPQ-standardissa oletetaan, että nimi on jo tiedossa etukäteen julkaisijalla ja tilaajalla. AMPQ ei tarjoa toiminnallisuutta instanssien nimien noutamiseen. Kestävyys parametrilla julkaisija voi kertoa palvelimelle, että välitäjä säilyttää vaihteen uudelleenkäynnistysten jälkeen. Jos ei, julkaisijan täytyy määrittää vaihde uudelleen käynnistyksen jälkeen. Automaattinen poisto kertoo poistaako välittäjä vaihteen automaattisesti, kun viimeinen siihen sidottu jono on poistettu ja julkaisija ei ole enää yhteydessä.

Kaikki julkaisijan ja tilaajan kutsut välittäjäpalvelimelle, jotka tekevät uuden instanssin komponentista, ovat esitteleviä (engl. declare). Tämä tarkoittaa että palvelin tekee tarvittaessa uuden instanssin komponentista, jos sitä ei ole jo olemassa ja vastaa onnistuneesti molemmissa tapauksissa. Tilanne tulee esimerkiksi silloin kun kaksi julkaisijaa käyttävät samaa vaihdetta keskenään. Toinen ei tiedä onko toinen jo määrittänyt instanssin vaihteesta palvelimelle, esimerkiksi silloin kun ohjelmat käynnistyvät eri aikaan. Jos kummatkin julkaisijat eksplisiittisesti määrittävät saman käytettävän vaihteen. Palvelin vastaa kummallekin onnistuneesti ja tuloksena palvelimella on vain yksi instanssi halutusta vaihteesta. Sama toiminta pätee kaikkiin välittäjäpalvelimen kutsuihin, jotka tekevät uusia instansseja komponenteista.

Vaihde reitittää viestejä jonoihin sen sidosten ja tyypin mukaan. Reititykseen liittyy tärkeä asia, joka on *reititysavain* (engl. *routing-key*). Reititysavain on kuin virtuaalinen osoite viestissä, jonka julkaisija liittää viestiin julkaisun yhteydessä. Tilaaja käyttää myös reititysavainta jonon määrityksen yhteydessä. Vaihde, tyypistä riippuen, voi käyttää tätä avainta reititykseen eri jonoihin. Viestin reititysavainta voi hyvin verrata lähetettävän sähköpostin saaja-kenttään. Saaja kertoo vastaanottajan sähköpostiosoitteen, johon viesti on tarkoitus lähettää. Reititysavain toimii juurikin näin suorassa viestin lähetyksessä, mutta eroaa muissa.

2.2.3 Suoravaihde (direct exchange)

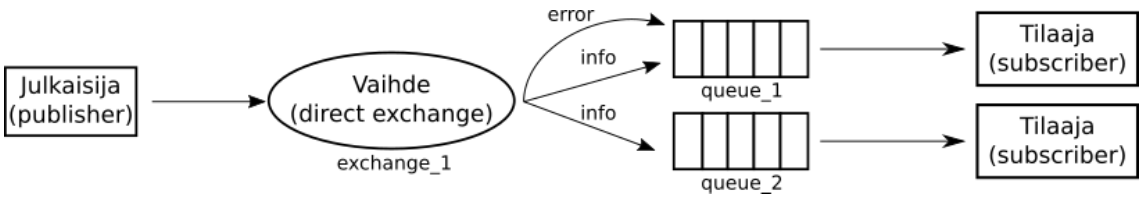
Julkaisija voi määrittää vaihteen instanssin tyypiksi suoravaihteen (engl. direct exchange). Suoravaihde reitittää viestin jonoihin vastaavan reititysavaimen perusteella. Suoravaihde reitittää seuraavasti:

- tilaaja määrittää sidoksen reititysavaimella K,
- julkaisija julkaisee viestin reititysavaimella R,
- vaihde välittää viestin jonoon jos K = R,
- muuten vaihde tiputtaa tai palauttaa viestin lähettäjälle.

Kuvassa 10 on esitetty suoravaihteen toiminta. Vaihteeseen on tehty sidoksia reititysavaimilla error ja info. Yksi tilaaja voi luoda sidoksia samaan vaihteeseen monella eri reititysavaimella. Näin tilaaja voi tilata viestejä mistä on kiinnostunut. Kuvassa 10 julkaisija julkaisee viestin reititysavaimella info. Viesti päätyy molempiin queue_1 ja queue_2 jonoon. Reititysavaimella error, viestit päätyvät vain jonoon queue_1. Välittäjäpalvelin tarjoaa suoravaihteesta oletusvaihteen nimeltä *amq.direct*. [2, s. 27]

2.2.4 Hajautusvaihde (fanout exchange)

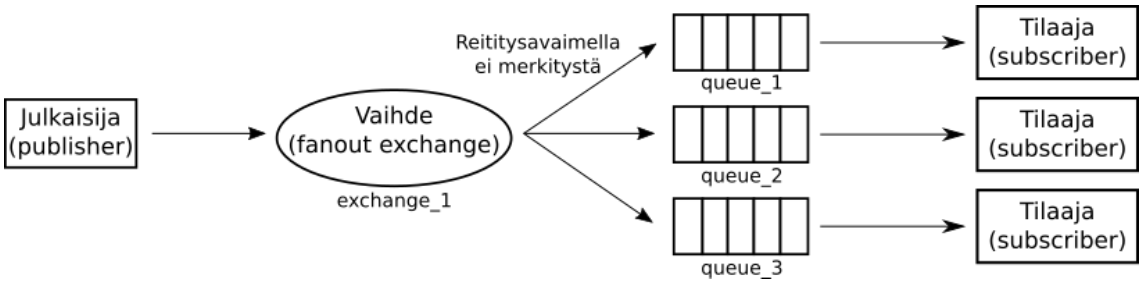
Julkaisija voi määrittää vaihteen instanssiksi hajautusvaihteen (engl. fanout exchange). Hajatusvaihde reitittää viestit kaikkiin sen jonoihin reititysavaimesta välittämättä. Hajautusvaihde toimii seuraavasti:



Kuva 10. *Suoravaihde (engl. direct exchange), reitittää suoraan sidoksen reititysavaimen mukaan (pohjautuu kuvaan [33]).*

- tilaaja määrittää sidoksen vaihteeseen reititysavaimella K,
- julkaisija julkaisee viestin reititysavaimella R,
- vaihde välittää viestin kaikkiin siihen sidottuihin jonoihin, reititysavaimesta riippumatta.

Kuvassa 11 on esitetty hajautusvaihteen toiminta. Vaihteeseen exchange_1 on tehty kolme eri sidosta jonoihin queue_1, queue_2 ja queue_3. Julkaisijan lähettämä viesti lähetetään kaikkiin kolmeen sidottuun jonoon, viestin ja jonojen reititysavaimista riippumatta. Välittäjäpalvelin tarjoaa hajautusvaihteesta oletusvaihteen nimeltä *amq.fanout*. [2, s. 27]



Kuva 11. *Hajautusvaihde (engl. fanout exchange) reitittää kaikkiin siihen sidottuihin jonoihin riippumatta reititysavaimesta (pohjautuu kuvaan [1]).*

2.2.5 Aihepiirivaihde (topic exchange)

Aihepiiri-vaihdetyyppi (engl. topic exchange) reitittää viestejä sidottuihin jonoihin reititysavaimen mukaan kuin suoravaihde, mutta tarjoaa lisäksi sääntöjä monen avaimen samanaikaiseen yhteensopivuuteen. Sidoksen reititysavaimen sijaan voidaan puhua reitityskaavasta (engl. routing pattern). Aihepiiri vaihde toimii seuraavasti:

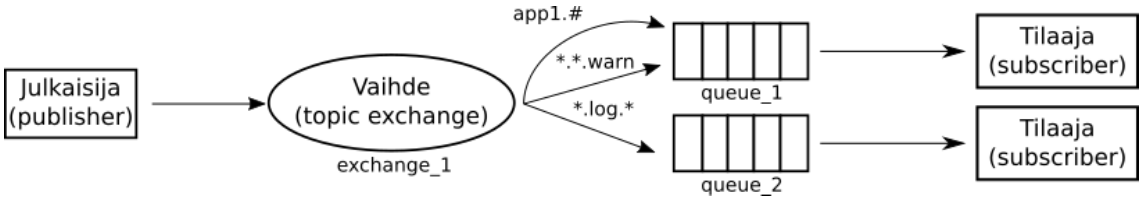
- tilaaja määrittää sidoksen vaihteeseen reitityskaavalla P,
- julkaisija julkaisee viestin reititysavaimella R,
- vaihde välittää viestin jonoon, jos sen reitityskaava P sopii reititysavaimeen R.

Aihepiirivaihteessa reititysavaimen täytyy olla lista sanoja, jotka ovat erotettu pisteillä ja ovat yhdessä maksimissaan 255 merkkiä pitkä [2, s. 35]. Sanat saavat sisältää kirjaimia A-Z ja a-z, ja numeroita 0-9. Yleensä avaimeen sijoitetaan sanoja mitkä liittyvät viestin

sisältöön. Tilaajan määrittämä sidoksen reitityskaava voi olla samaa muotoa kuin reititysavain, mutta sanojen tilalla voidaan käyttää seuraavia erikoismerkkejä:

- ***** (tähti), voi vastata mitä tahansa yhtä sanaa,
- **#** (risuaita), voi vastata nolla tai monta sanaa. [2, s. 27]

Kuvassa 12 on esitetty aihepiirivaihteen toiminta. Vaihteeseen exchange_1 on sidottu jono queue_1 reitityskaavoilla *app1.#* ja **.*.warn*. Ja jono queue_2 reitityskaavalla **.log.**. Oletetaan, että julkaisija lähettää viestejä avaimella muodossa *ohjelma.kanava.taso*, jossa *ohjelma* kuvaa julkaisijan nimeä. *Kanava* kuvaa lokitusväylää ja *taso* kuvaa viestin tasoa (warning, error, info jne.). Voisi sanoa että queue_1 on kiinnostunut kaikista ohjelmalta app1 tulevista viesteistä ja myös kaikista varoitustason (warning) viesteistä. Jono queue_2 on vain kiinnostunut kaikista log-väylän viesteistä.



***Kuva 12**. Aihepiirivaihde (engl. topic exchange), reitittää kaikkiin siihen sidottuihin jonoihin, joiden reitityskaava sopii viestin reititysavaimeen (pohjautuu kuvaan [34]).*

Nyt jos julkaisija lähettää viestin avaimella *app1.debug.warn*, vaihde välittää viestin jonoon queue_1, mutta ei jonoon queue_2. Avaimella *app2.log.info* viesti välitetään vain jonoon queue_2. Avaimella *app1.log.warn* viesti lähetään molempiin jonoihin. Kun taas avaimella *app2.debug.info* viestiä ei lähetetä yhteenkään jonoon.

Aihepiirivaihde on vaihdetyypeistä monimutkaisin, mutta kattaa ison määrän erilaisia käyttötapauksia. Vaihteen avulla tilaajat voivat tilata viestejä, joista ovat esimerkiksi kiinnostuneita. Aihepiirivaihdetta voi käyttää kuin aikaisempia vaihdetyyppejä. Jos jono sidotaan reitityskaavalla #, se vastaanottaa kaikki viestit kyseiseltä vaihteelta ja käyttäytyy kuin hajautusvaihde. Jos jono sidotaan ilman merkkejä * ja #, niin se käyttäytyy samalla tavalla kuin suoravaihde. [34]

2.2.6 Jonon määritys ja viestien kuittaaminen

AMQ-mallissa jono (engl. queue) on vaihteen ja tilaajan välissä oleva puskuri (kuva 9), joka tallentaa tilaajalle tulevia viestejä. Jono pitää viestejä jonossa tilaajalle, kunnes tämä ehtii prosessoida ne. Yksi jono voi puskuroida viestejä monelle eri tilaajalle. Jotta jonoon saapuu viestejä, täytyy tilaajan sitoa (engl. binding) jono palvelimella olemassa olevaan vaihteeseen. Tällä mekanismilla tilaaja voi valita mistä julkaisijasta on kiinnostunut. Tilaaja voi sitoa saman jonon moneen eri julkaisijaan. Tämä mahdollistaa viestien tilaamisen monelta eri vaihteelta. Sidos vaihteeseen tehdään vaihteen nimellä, eli tilaajan täytyy

tietää vaihteen nimi etukäteen. Jonolla tilaaja voi määrittää attribuutteja. Jotkin attribuutit ovat samoja kuin vaihteella. Tilaaja voi määrittää jonolle *nimen* (engl. *name*), *kestävyyden* (engl. *durable*), *poissulkevuuden* (engl. *exclusive*) ja *automaattisen poiston* (engl. *auto-delete*). Nimi yksilöi jonon palvelimella. Tilaaja voi halutessaan pyytää palvelinta generoimaan yksilöivän nimen jonolle automaattisesti. Kestävyys-attribuutti säilyttää jonon palvelimella uudelleenkäynnistyksen jälkeen. Poissulkeva rajoittaa jonon vain yhdelle tilaajalle ja palvelin poistaa jonon kun yhteys tilaajaan katkeaa. Automaattinen poisto poistaa jonon palvelimelta automaattisesti kun yhteys viimeiseen tilaajan on katkennut. [1]

Jos saman nimisen jonoon on liittynyt monta eri tilaajaa. Palvelin lähettää viestin jonosta vain yhdelle tilaajalle kerrallaan kiertovuorottelu (engl. round-robin) periaatteen mukaan. Sama viesti lähetetään ainoastaan toiselle tilaajalle jos se edelleenlähetetään virheen tai peruutuksen seurauksena [2, s. 11–12]. Tilaajan täytyy määrittää jonolle sen käyttämä viestin kuittaamisen (engl. acknowledge) malli ennen kuin jono poistaa viestin puskurista. Malleja on kaksi:

- automaattinen, jolloin palvelin poistaa viestin jonosta heti kun se on lähetetty tilaajalle,
- eksplisiittinen, jolloin palvelin poistaa viestin, kun tilaaja on lähettänyt kuittauksen palvelimelle.

Tilaaja voi lähettää viestistä kuittauksen milloin vain prosessoinnin aikana. Heti kun viesti on vastaanotettu tai silloin kun viesti on prosessoitu. [2, s. 29]

3. PROJEKTIN LÄHTÖKOHDAT

Tarkoituksena tässä diplomityössä oli toteuttaa ohjelmistokomponentti osaksi isompaa järjestelmää. Isompi järjestelmä liittyi sähköasemien toimintaan ja niiden tarkkailuun. Ohjemistokomponentin tarkoituksena oli tilata viestejä sähköaseman IED-laitteen RCB-instansseilta IEC 61850 -standardin mukaisesti. Standardin mukainen viesti prosessoitiin ja jaettiin järjestelmän muiden komponenttien kanssa. Esimerkiksi viesti voi sisältää mittaustietoa, joka halutaan näyttää loppukäyttäjälle käyttöliittymässä. Käyttöliittymän päivittävä komponentti tarvitsee mittaustiedon IED-laitteelta.

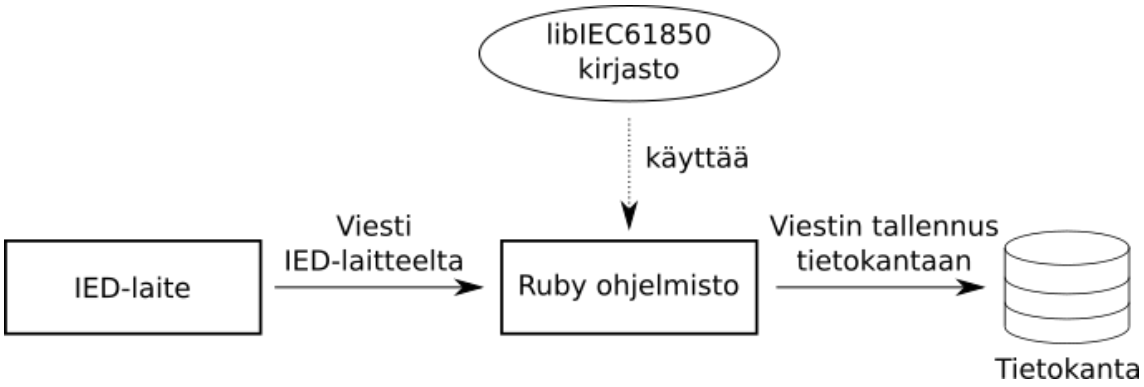
Ennen tämän työn aloittamista yrityksessä oli jo kehitetty ensimmäinen versio ohjelmasta. Ohjelma kykeni tilaamaan viestejä IED-laitteen kaikilta RCB-instansseilta, prosessoimaan viestit ja tallentamaan ne relaatiotietokantaan myöhempää käyttöä varten. Tässä ohjelmistossa oli havaittuja ongelmia ja se ei myöskään tukenut kaikkia IEC 61850 -standardin viesteihin liittyviä ominaisuuksia. Tämän ohjelmiston toimintaperiaate ja siinä olleet ongelmat toimivat pohjana uuden version suunnittelulle ja toteutukselle. Tarkoituksena oli poistaa havaitut ongelmakohdat ja miettiä olisiko jokin muu arkkitehtuuri parempi kyseiseen toteutukseen. Ensimmäistä toteutusta ohjelmasta voisi nimittää ensimmäiseksi protoversioksi tai demovaiheeksi (engl. proof of concept), jonka pohjalta tultiin tekemään toimiva lopullinen versio. Tekstissä eteenpäin sanoilla demo ja demoversio viitataan tähän ohjelmistoon.

Tässä osiossa pohjustetaan työn alkua lukijalle ja mistä lähdettiin liikkeelle. Lisäksi kuvataan mitä ongelmia demovaiheen toteutuksessa ja oli ja analysoida niitä. Demovaiheen ohjelmasta käsitellään sen arkkitehtuuria, mitkä olivat sen komponentit ja niiden toiminnallisuus. Tässä käsitellyt ongelmat toimivat pohjana uuden version suunnittelulle ja auttavat tekemään siihen liittyviä ratkaisuja.

3.1 Demon arkkitehtuuri

Demoversio oli ohjelmoitu Ruby-ohjelmointikielellä. Ohjelman arkkitehtuuri oli yksinkertainen. Kuvassa 13 on esitetty demoversion arkkitehtuuri korkealla tasolla ja kuinka viesti IED-laitteelta kulkee tietokantaa. Yksi ajettu demoversion prosessi pystyi tilaamaan yhden IED-laitteen kaikki RCB-luokkien instanssit. Instanssien tiedot luettiin relaatiotietokannasta. Ohjelmisto prosessoi viestit ja tallensi ne relaatiotietokantaan myöhempää käyttöä varten. Ruby-ohjelmistossa tärkeässä osassa oli *libIEC61850*-kirjasto [22]. libIEC61850-kirjasto on avoimen lähdekoodin C-kielellä toteutettu kirjasto, joka abstrahoi IEC 61850 -standardin matalan tason määrittämiä palvelukutsuja ja datarakenteita helppokäyttöiseksi rajapinnaksi. Kirjasto tarjosi toiminnallisuuden IED-laitteella olevan

palvelinohjelmiston, sekä IED-laitetta käyttävän asiakasohjelmiston toteuttamiseen. IED-laitteen palvelimelle kirjasto tarjosi funktioita ja rakenteita IEC 61850 määrittämien luokkien ja hierarkian rakentamiseen ja käsittelyyn. IED-laitteen asiakasohjelmalle kirjasto tarjosi funktioita ja rakenteita standardin määrittämiin palveluihin, kuten arvojen lukuun ja asettamiseen, datajoukkojen käyttöön ja viestien tilaamiseen. Tätä samaa kirjastoa käytettiin myös tässä työssä toteutetussa ohjelmistokomponentissa. Demossa ja lopullisessa ohjelmistokomponentissa käytettiin vain kirjaston tarjoamia IED-laitteen asiakasohjelmiston ominaisuuksia



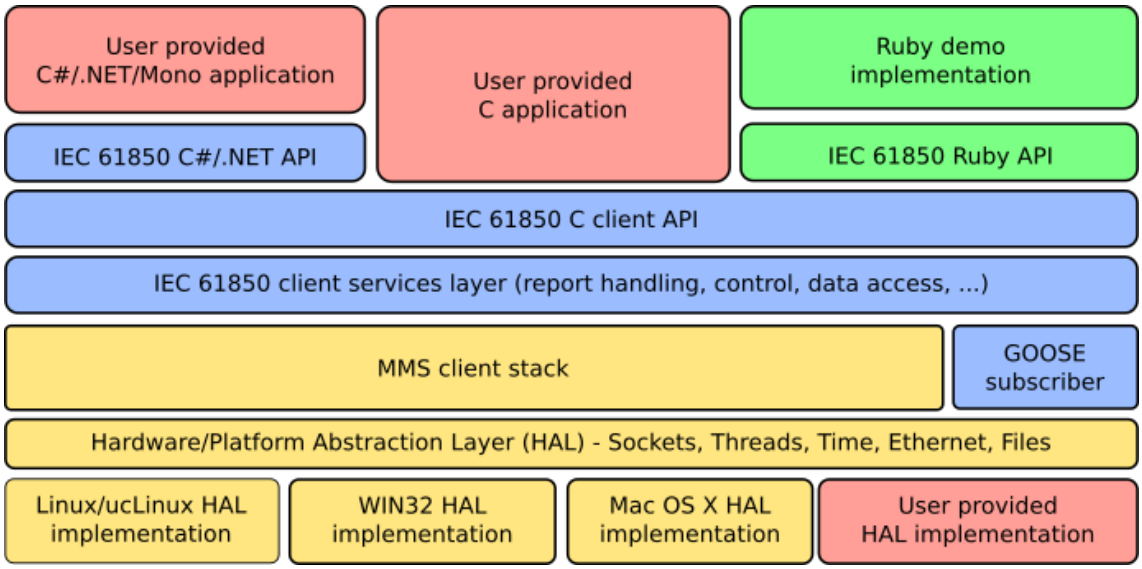
Kuva 13. *Rubylla toteutetun demoversion arkkitehtuuri ja tiedonsiirto.*

LibIEC61850-kirjasto on rakennettu käyttämään MMS-protokollaa tiedonsiirrossa IED-laitteen ja sen asiakasohjelman välillä, kuten IEC 61850 -standardin osassa 8-1 määritetään. Kuvassa 14 on esitetty kirjaston kerrosarkkitehtuuri asiakasohjelmalle. Kirjastoon on toteutettu *laiteabstraktiokerros* (engl. *Hardware Abstraction Layer*, lyhennetään *HAL*). HAL:in avulla kirjasto voi toimia monella eri laitealustalla, ja käyttäjä voi tarvittaessa lisätä oman HAL-implementaation. Demoversiota suoritettiin Linux-käyttöjärjestelmällä, joten kirjastosta käytettiin olemassa olevaa Linux HAL -toteutusta. Kuvassa 14 on punaisella merkitty laatikot, jotka kirjaston käyttäjä voi tarjota itse, keltaisella kirjaston uudelleenkäytettävät MMS-protokollan osuudet ja sinisellä IEC 61850 -standardin toteuttavat osuudet. Kuvaan on merkitty vihreällä demoon toteutetut osuudet, eli Ruby-kielelle liitos C-kieleen ja tämän päälle Rubylla tehty ohjelmisto.

Ruby-koodista C-kielen funktioiden kutsuminen ei ole suoraan mahdollista, vaan kielten väliin täytyy toteuttaa liitos. Demoversiossa liitos oli tehty käyttäen Rubylle saatavaa *ruby-ffi* -kirjastoa [35] (engl. *Foreign Function Interface*, lyhennetään *FFI*). Liitoksen avulla Ruby voi kutsua C-kielen funktioita ja käyttää sen struktuureita ja muuttujia. Demossa kirjasto huolehti matalan tason IEC 61850 asiat, ja Ruby-koodi keskittyi liitoksen avulla korkean tason viestin jäsentämiseen ja tallennukseen tietokantaan.

3.2 Demon toiminta ja sen ongelmat

Demo oli toteutettu käyttäen *Ruby on Rails* -kehystä, lyhennetään *RoR*. RoR on tarkoitettu web-sovellusten toteuttamiseen Ruby-kielellä. Se tarjoaa *Active Record* -nimisen



Kuva 14. *libIEC61850-kirjaston kerrosarkkitehtuurin komponentit, vihreällä Ruby toteutukseen lisätyt osat (pohjautuu kuvaan [20]).*

ORM-kerroksen (engl. *Object-Relational Mapping*) tietokannan käsittelyn helpottamiseen. ORM-kerros abstrahoi relaatiotietokannan käyttämisen oliopohjaiseksi ja kyselyitä tietokantaan voi tehdä suoraan Ruby-kielellä. Demo käytti RoR:in ORM-kerrosta tietokannan käyttämiseen. Demossa päädyttiin käyttämään RoR-kehystä, koska muu järjestelmä oli toteutettu tällä kehyksellä. Ennen suorittamista ohjelmaan täytyi ladata RoR:in ympäristö muistiin, jonka seurauksena yksinkertaisen ohjelman piti varata iso määrä muistia.

Ohjelman toiminta on esitetty sekvenssikaavioissa 15 ja 16. Sekvenssikaavio 15 jatkuu kuvassa 16. Kuvissa ohjelman kaksi eri silmukkaa on esitetty loop-laatikoilla. Sekvenssikaaviossa osallisena ovat tietokanta, Ruby-ohjelma, libIEC61850-kirjasto, libIEC61850-kirjaston natiivisäie ja IED-laitteen palvelinohjelma. Rubyn ja libIEC61850-kirjaston liitos oli tehty ruby-ffi -kirjastolla ja kirjaston natiivisäie oli vastuussa yhteyden ylläpidosta ja datan siirtämisestä. Sekvenssikaavioon on merkitty paksulla suorituksessa olevat palkit, esimerkiksi IED-laitteen palvelinohjelmisto on koko ajan suorituksessa. Tulevassa tekstissä viitataan molempien kuvien kohtiin numeroilla.

Ensimmäisenä ohjelma luki tietokannasta IED-laitteen, sekä sen kaikki RCB-instanssien tiedot. Tämän avulla ohjelma tiesi mikä IED-laitteen IP-osoite on ja mitkä olivat RCB-instanssien viitteet (kohdat 1–2). Ohjelmaan pystyi syöttämään eri tietoja ainoastaan tietokannan kautta ennen suoritusta. Tämän jälkeen ohjelma pystyi muodostamaan yhteyden IED-laitteelle tekemällä instanssin `IedConnection` struktuurista funktiolla `IedConnection_create()` (kohdat 3–4). Tämän jälkeen struktuuri annetaan `IedConnection_connect()` funktiolle, joka avaa yhteyden IED-laitteelle ja palaa vasta kun vastaus saapuu (kohdat 5–11). Tässä vaiheessa libIEC61850-kirjasto käynnistää erillisen natiivisäikeen yhteyden viestien vastaanottoon. Tätä säiettä kirjasto käyttää tulevien viestien vastaanottoon ja lähettämiseen. Yhteyden avauksen jälkeen jokainen RCB-instanssi tilataan lukemalla ensin sen arvot IED-laitteelta funktiolla `IedConnection_getRCBVa-`

`lues()` (kohta 12). Funktiokutsu nukkuu ja palaa vasta kunnes erillinen säie ilmoittaa, että vastaus on saapunut tai yhteyden aika ylittyy. Kirjaston funktio `sendRequestAndWaitForResponse()` nukkuu ja odottaa vastausta (kohdat 13–16). RCB-arvot luettuaan, kirjasto palauttaa struktuurin `ClientReportControlBlock`, joka sisältää luetut tiedot RCB-instanssista (kohta 17). Samaa struktuuria käytetään arvojen muuttamiseen ja niiden takaisin kirjoittamiseen IED-laitteelle. Ennen muunneltujen RCB-arvojen takaisin kirjoittamista ja viestien tilaamista, täytyy kirjastolle asettaa takaisinkutsufunktio, jota kirjasto kutsuu aina kun tilattu viesti saapuu IED-laitteelta. Takaisinkutsufunktioksi asetetaan funktiolla `IedConnection_installReportHandler()` (kohdat 19–20). Asetuksen ajaksi kirjasto lukitsee `reportHandlerMutex`:in. Tätä samaa lukitusta käytetään kun viesti takaisinkutsufunktiota kutsutaan viestin saapuessa (kohdat 33–36). Tilanteessa jossa takaisinkutsufunktiota asetetaan samalla kun viesti on saapunut, joutuu toinen osapuoli odottamaan lukituksen vapautumista. Tämä lukitus on tärkeä huomio myöhemmin kun käydään läpi ohjelman huonoa suorituskykyä. Tämän jälkeen arvot kirjoitetaan takaisin IED-laitteelle funktiolla `IedConnection_setRCBValues()` (kohdat 21–26). Tämä funktio palaa vasta kun IED vastaa tai yhteyden aika ylittyy. Heti arvojen kirjoitusten jälkeen IED aloittaa lähettämään viestejä tilaajalle. Eli samalla kun muita RCB-instansseja tilataan, tilatut RCB-instanssit lähettävät jo viestejä ja aiheuttavat takaisinkutsufunktion suorittamisen. Kun kaikki RCB-instanssit on tilattu, ohjelma jää viimeiseen silmukkaan odottamaan ja prosessoimaan viestejä (kohdat 27–36). Kun viesti saapuu, säie kutsuu ensin sisäisesti `mmsIsoCallback()` funktiota, joka kutsuu muita kirjaston sisäisiä funktioita ja lopuksi asetettua takaisinkutsufunktiota (kohdat 28–33). Takaisinkutsufunktio on liitetty Ruby funktioon ja funktio tallentaa raportin tiedot tietokantaan (kohdat 33–36). Ruby-funktion suorituksen ajaksi kirjasto lukitsee `reportHandlerMutex`:in, ja vapautetaan kunnes Ruby-funktion suoritus palaa. Tätä jatkuu niin kauan kunnes ohjelmalle lähetetään jokin signaali joka lopettaa sen suorituksen. [19, 37, 27]

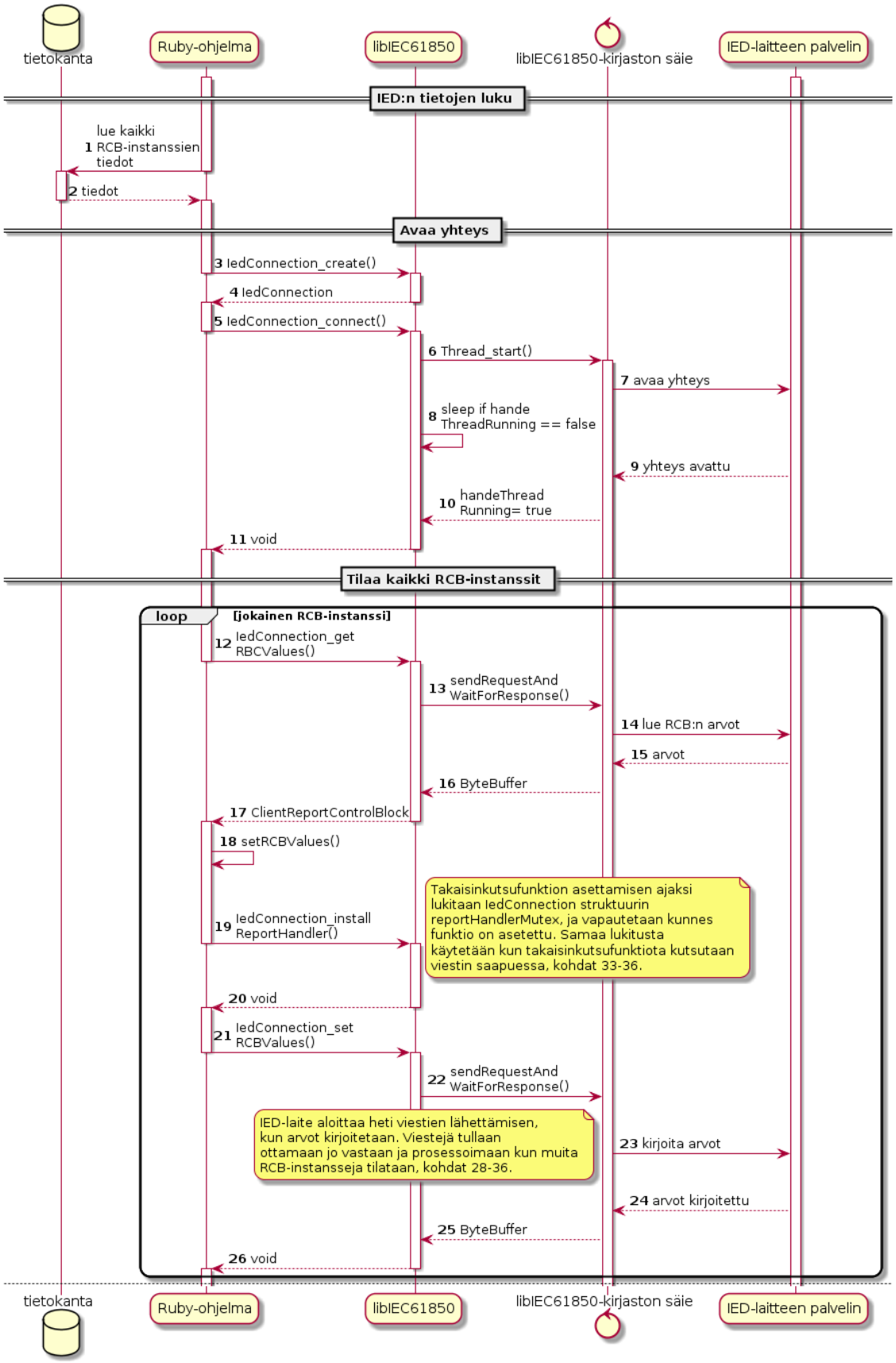
Demossa isoimpana ongelmana oli sen huono suorituskyky ja toiminnan epävarmuus RCB-instanssien määrän ollessa enemmän kuin muutama. RCB-instanssien määrän ollessa liian suuri ohjelma saattoi epäonnistua osan tilaamisessa, koska yhteys aikakatkaistiin arvojen kirjoituksessa tai luvussa. Lisäksi ongelmaksi muodostui usean RCB instanssin tilaamisen kulunut aika. Yhteensä aikaa saattoi kulua 30 sekuntia kaikkien instanssien tilaamiseen.

Huonoon suorituskykyyn oli syynä muutama asia. Yksi niistä oli Ruby-kielen huonompi suorituskyky verrattuna natiivisti käännettyyn C-kieleen. Ruby on tulkattava kieli kuten esimerkiksi Python, joka tulkataan rivi kerrallaan ja suoritetaan. Lähdekoodia ei käännetä kokonaan ensin konekäskyiksi erillisellä kääntäjällä, kuten C-kielessä. Valmiiksi käännetty lähdekoodi tarvitsee vain ajaa, kun taas tulkattavassa kielessä rivi täytyy ensin tulkata ja sitten ajaa. Rubyssa käytettiin sen oletustulkkia *MRI/YARV* (engl. *Matz's Ruby Interpreter*, lyhennetään *MRI* tai *Yet another Ruby VM*, lyhennetään *YARV*). Ruby versiosta 1.9 eteenpäin käyttää YARV-tulkkia. Toinen syy oli Ruby-kielen oletustulkissa oleva *globaali*

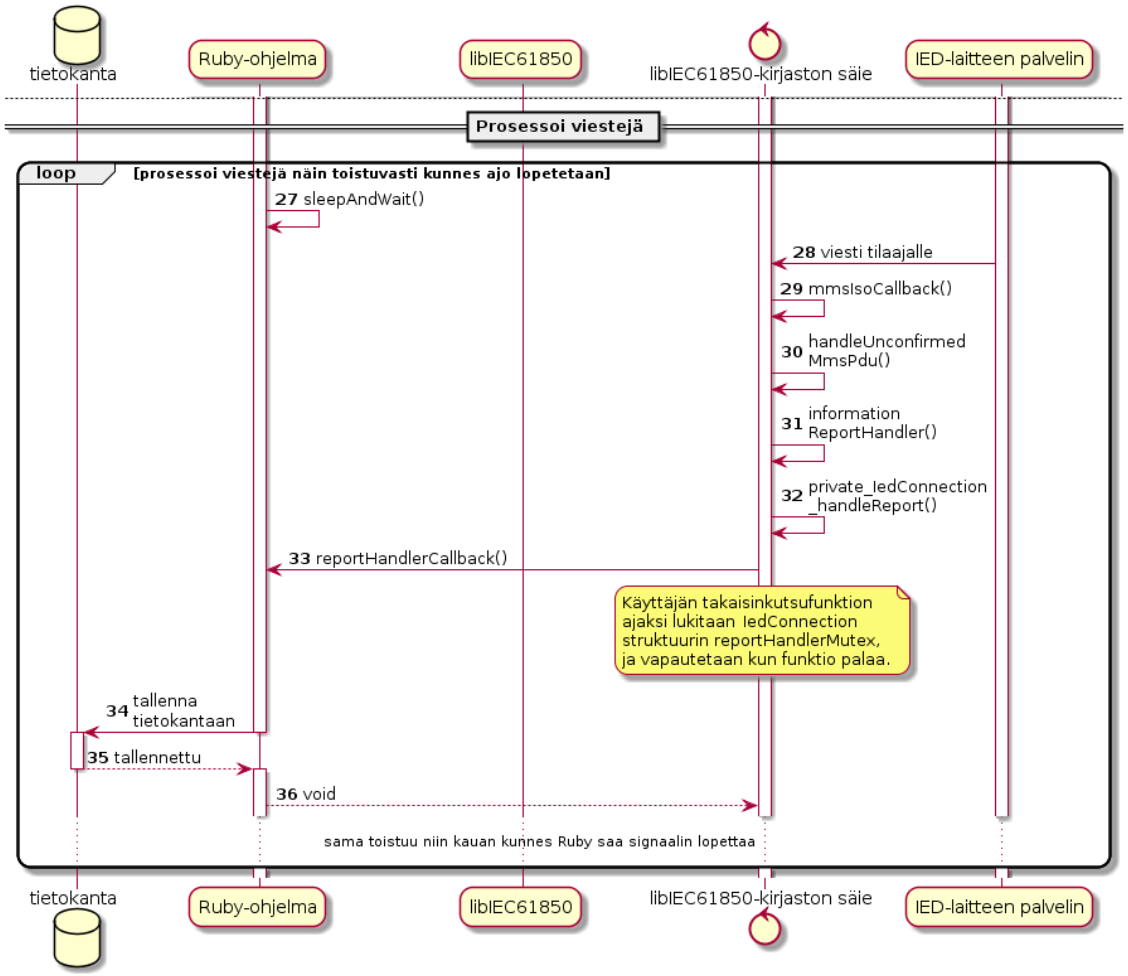
tulkkilukitus (engl. *Global Interpreter Lock*, lyhennetään *GIL*, tai *Global Virtual Machine Lock*, lyhennetään *GVL*). GIL pakottaa Ruby-ohjelman ajoon vain yhdellä ytimellä ja vain yksi säie vuorossa kerrallaan ja on täysin riippumaton käyttöjärjestelmän vuorottajasta [26, s. 131–133]. Kuvassa 17 on esitetty kuinka Ruby-tulkki vuorottaa kahta ajossa olevaa säiettä. Kuvassa demon Ruby-koodi kutsuu `IedConnection_setRCBValues()` funktiota, ajo jää kesken ja tapahtuu vaihto, koska viesti saapui. Takaisinkutsufunktio suoritetaan ja suoritus palaa takaisin aikaisempaan funktion suoritukseen. Tässä vaiheessa jos vaihto on huonolla hetkellä ja kesti liian kauan, tulee yhteyden aikakatkaisu ja RCB-instanssi jää tilaamatta. Huonoon suorituskykyyn mahdollisesti vaikutti myös lukitus `reportHandlerMutex`, jota kirjastossa käytetään kun takaisinkutsufunktio asetetaan tai sitä suoritetaan. Lukitus aiheuttaa säikeen nukkumisen niin kauan kunnes lukitus vapautuu. Tässä tapauksessa, jos viestin prosessointi kestää liian kauan (kuvassa 16 kohdat 33–36) ja samalla tilataan muita RCB-instansseja (kuvassa 15 kohdat 12–26). Säie joutuu odottamaan lukituksen vapautusta takaisinkutsufunktion asetettamisen ajan (kohdat 19–20). Ratkaisuna tähän olisi pitää takaisinkutsufunktio mahdollisimman lyhyenä suoritusajan suhteen.

Tämän lisäksi demototeutuksessa oli muistivuoto. Muistivuoto on tilanne, missä ohjelma varaa kokoajan lisää muistia eikä vapauta sitä takaisin käyttöjärjestelmälle uudelleen käyttöön. Muistivuoto johtui todennäköisesti jostakin ohjelmointivirheestä ruby-ffi -kirjaston liitoksen kanssa. Kun liitos Rubysta tehdään C-kieleen, täytyy ohjelmoijan miettiä roskien keruuta tarkasti. Tätä ei normaalisti tarvitse miettiä Rubyssä, koska tulkki implementoi automaattisen roskien keruun. Muistivuoto havaittiin kun ohjelma jätettiin suoritukseen pitemmäksi aikaa ja ohjelma oli varannut melkein kaiken käyttöjärjestelmän muistista itselleen. Lisäksi jos ohjelmaa ajaa ja tarkkailee Linuxin *htop*-ohjelmalla, voi *MEM%*-sarakkeesta huomata prosentuaalisen osuuden kasvavan koko käyttöjärjestelmän muistista.

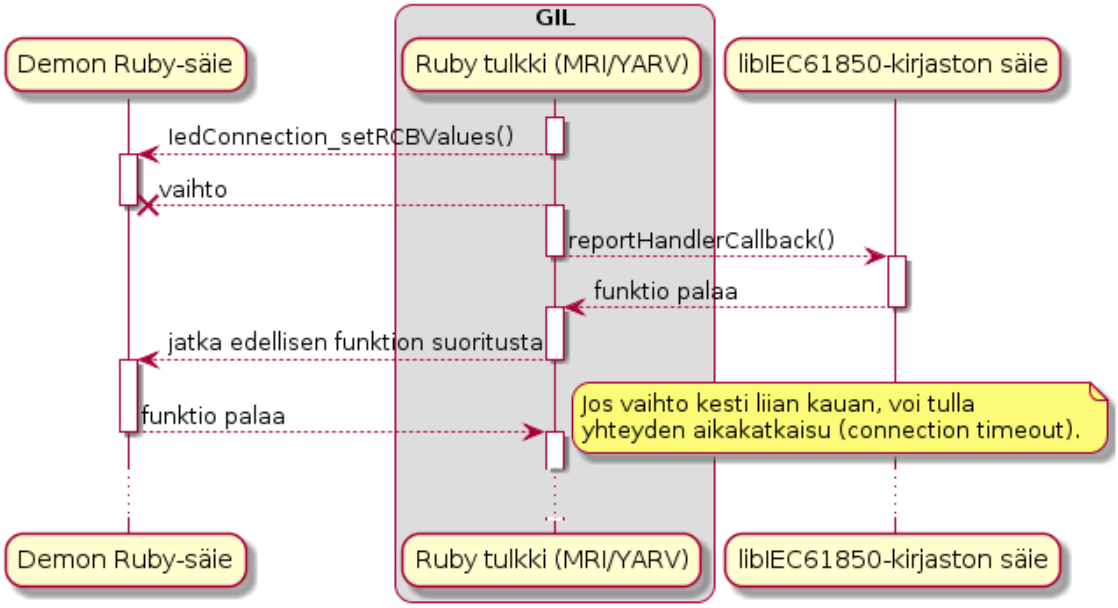
Näiden lisäksi tiedon jako järjestelmän muiden komponenttien kanssa oli huono. Ohjelma prosessoi viestit relaatietokantaan. Muiden komponenttien täytyi kysellä tietoa tietokannasta tasaisin väliajoin, ilman tietoa milloin uusi tieto on saatavissa. Komponenttien määrästä riippuen tietokanta on turhan rasituksen alaisena jatkuvasti. Tulevaisuutta ajatellen lopullinen tiedon tallennuspaikka ei ole muiden tietoa tarvitsevien ohjelmien kannalta järkevä. Tarvittaisiin keino, jolla komponentit voisivat saada tiedon uudesta viestistä erikseen, ilman että sitä täytyisi kysellä väliajoin.



Kuva 15. *Sekvenssikaavio kuinka Ruby-ohjelma avaa yhteydet ja tilaa kaikki IED-laitteen RCB-instanssit (jatkuu kuvassa 16).*



Kuva 16. *Sekvenssikaavio kuinka Ruby-ohjelma prosessoi ja tallentaa viestejä libIEC61850-kirjastoa käyttäen (jatkuu kuvasta 15).*



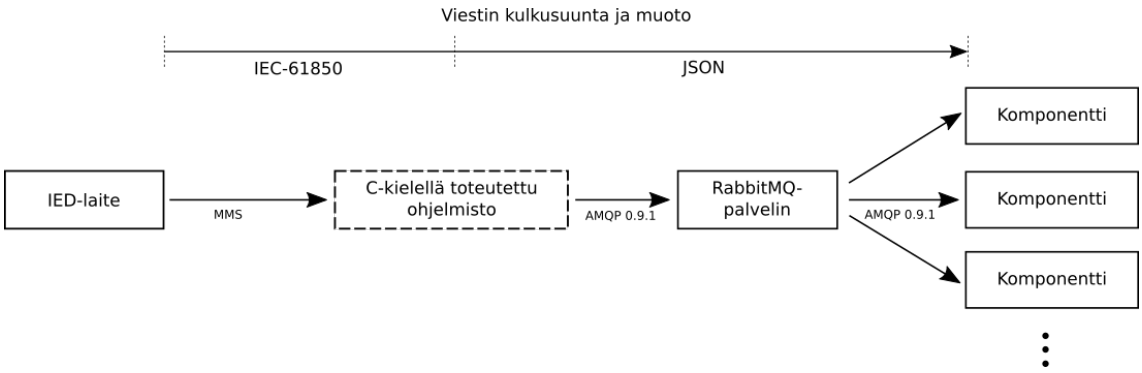
Kuva 17. *Ruby-tulkin globaalin lukituksen toiminta, joka vuorottaa ajossa olevia säikeitä riippumatta käyttöjärjestelmän vuorottajasta.*

4. SUUNNITTELU

Tässä osuudessa käydään toteutetun ohjelman suunnittelu läpi ja kerrotaan miten ja miksi ratkaisuihin päädyttiin. Kappaleissa vertaillaan eri vaihtoehtoja ja peilataan kappaleessa 3.2 käytyjä demoversion ongelmia ja niiden perusteella yritetään löytää toimiva ratkaisu. Ensin suunnitellusta ohjelmasta annetaan kattava kokonaiskuva lukijalle ja tämän jälkeen tulevissa kappaleissa mennään tarkemmin yksityiskohtiin.

4.1 Kokonaiskuva

Aikaisemmin kappaleessa 3.1 kuvassa 13 esiteltiin demoversion arkkitehtuuri ja sen toimintaa. Esimerkiksi kuinka viestit IED-laitteelta kulkevat ohjelman läpi ja päätyvät tietokantaan. Kuinka järjestelmän muut komponentit lukevat tietoa tietokannasta kyselemällä sitä väliajoin ilman tietoa milloin uusi tieto on saapunut. Suunnittelun jälkeen demosta päädyttiin kuvassa 18 olevaan arkkitehtuuriin. Kuvassa katkoviivalla on merkitty tässä kappaleessa suunniteltu ohjelmisto. Ja kuvan yläreunassa oleva viiva kuvaa viestin kulkua järjestelmän eri osapuolten läpi ja missä muodossa viesti on missäkin kohtaa. Kuvassa vasemmalla on IED-laite, josta komponentti viestit tilaa. Komponentti prosessoi ja julkaisee viestit RabbitMQ-välityspalvelimelle. Kuvassa oikealla järjestelmän muut komponenttit tilaavat viestejä välityspalvelimelta.



***Kuva 18**. Suunnitellun komponentin toiminta ja viestin kulkeminen ja muoto osapuolten välillä.*

Komponentti päädyttiin toteuttamaan C-kielellä ja on komentorivipohjainen ohjelmisto. Komponentti ei käyttänyt tietokantaa vaan kaikki ohjelman ajoon tarvittavat parametrit annettiin komentoriviparametreille ennen ohjelman käynnistämistä. C-ohjelma pystyi tilaamaan yhden IED-laitteen RCB-instansseja. Tilattuaan RCB-instanssit, ohjelma odotti viestejä IED-laitteelta IEC 61850 -standardin määrittämässä muodossa. Viestin saapuessa ohjelma prosessoi sen ja julkaisi AMPQ-standardin pohjaiselle välittäjäpalvelimelle *JSON*-muodossa (engl. *JavaScrip Object Notation*). Lopullisessa toteutuksessa

välittäjäpalvelimena käytettiin RabbitMQ-nimistä ohjelmistoa, joka pohjautuu AMPQ-standardin versioon 0.9.1. Välittäjäpalvelimelta muut järjestelmän komponentit pystyivät tilaamaan viestejä, ja viestin saapuessa palvelin ilmoitti siitä tilaajalle. Toteutettu C-ohjelmisto käytti edelleen demoversiosta tuttua libIEC61850-kirjastoa hoitamaan matalan tason IEC 61850 -standardin toiminnallisuuden.

4.2 Järjestelmän hajautus ja arkkitehtuuri

Komponentille vaatimuksena oli järjestelmän hajauttamisen mahdollistaminen. Komponentti pitäisi olla oma kokonaisuutensa ja sen ei tarvitse tietää mitään muista järjestelmän komponenteista. Järjestelmän hajautus tarkoittaa järjestelmän osittamista pieniin omiin kokonaisuuksiinsa, jotka kommunikoivat keskenään esimerkiksi viestien välityksellä. Demossa erilliset ohjelmat joutuivat toistuvasti lukemaan viestejä tietokannasta ilman tietoa siitä, milloin uusi viesti oli saatavilla. Tällainen ratkaisu ei ollut hyvä järjestelmän hajautuksen näkökulmasta. Tilanne pahenisi jos tietoa tarvitsevia järjestelmän komponentteja olisi enemmän. Lisäksi tässä toteutuksessa tietokanta on jatkuvan lukemisen alaisena. Tilanteeseen tarvittiin ratkaisu, jossa järjestelmän osa voisi tilata tietoa ja saada ilmoituksen kun tieto on saatavilla. Toisin sanoen tilaaja-julkaisija-arkkitehtuurimalli.

Yhtenä ratkaisuna tilanteeseen jossa moni komponentti tarvitsee samaa tietoa, olisivat ne voineet suoraan tilata viestit IED-laitteelta. Näin kaikki tilaajat saisivat saman viestin. Kuitenkin tässä esteenä on, että IEC 61850 -standardin määrityksen mukaan yksi RCB-instanssi voi olla tilattuna vain yhdellä asiakkaalle kerrallaan. Tämä käsiteltiin kappaleessa 2.1.6. Ja IED-laitteiden RCB-instanssit ovat rajalliset ja päätetty laitteen konfiguroinnin yhteydessä. Lisäksi IED-laitteet pystyvät rajoittamaan päällä olevien yhteyksien määrää johonkin lukuun, joka voi olla pieni. Tavoitteena siis olisi minimoida avoimet yhteydet IED-laitteelle, ja samalla tarjota saapunut viesti mahdollisimman monelle siitä kiinnostuneelle osapuolelle. Näistä vaatimuksista päästään ratkaisuun, missä yksi ohjelma tilaa kaikki halutut RCB-instanssit yhdeltä IED-laitteelta. Se odottaa viestien saapumista ja lähettää ne edelleen muille niitä tarvitseville ohjelmille. Viestejä tarvitsevien ohjelmien määrä voi vaihdella tarpeen mukaan. Tästä päästään vaatimukseen, että IED-laitteelta viestejä tilaavan ohjelmiston ei tarvitsisi tietää muista tilaavista ohjelmista mitään. Ohjelman pitäisi pystyä julkaisemaan viestit eteenpäin, välittämättä siitä kuka viestejä vastaanottaa.

Ratkaisuna yllä mainittuihin vaatimuksiin oli sijoittaa IED-laitteen ja muiden järjestelmän osien väliin *väliohjelmisto* (engl. *middleware*), kuten kuvassa 18 C-ohjelma on sijoitettu. Tällä pystyttiin minimoimaan yhteyksien määrä IED-laitteelle yhteen. Lisäksi sijoittamalla C-ohjelman ja muiden järjestelmän komponenttien väliin välittäjäpalvelin, saatiin aikaan vaatimuksista haluttu joustavuus. C-ohjelman ei tarvitse välittää siitä kuka viestejä vastaanottaa, ja välittäjäpalvelimen avulla yhden julkaisijan viestit voi tilata monta erillistä tilaaja. Välittäjäpalvelimelta jokainen tilaaja saa saman alkuperäisen viestin, mutta kopiona. IEC 61850 -standardi määritti vain tilaaja-julkaisija-arkkitehtuurimallin

ainoaksi tavaksi saada viestejä. Niinpä väliohjelmiston toteuttaminen ja saman arkkitehtuurin jatkaminen järjestelmän muille osille on tilanteeseen sopiva arkkitehtuurimalli. Tämä suunnitelma arkkitehtuurista täytti kaikki sille asetetut vaatimukset ja todettiin toimivaksi. Lisäksi nämä päätökset vastaavat kappaleessa 1 asetettuun tutkimuskysymykseen, mikä arkkitehtuurimalli sopisi parhaiten tähän tilanteeseen sopivaksi. Myös kysymykseen kuinka järjestelmä hajautetaan niin että tiedon siirto eri osapuolten välillä olisi mahdollista ja joustavaa saatiin vastaus. Vastaus tähän kysymykseen on käyttää jotakin viestintäprotokollaa, jonka kaikki osapuolet voivat ymmärtää. Arkkitehtuurissa käytettiin AMQP-viestintäprotokollaa, joka mahdollista yhteiset säännöt ja kommunikoinnin eri osapuolten välillä. Viestintä on yleinen tapa kommunikoida hajautetussa järjestelmässä [8, s. 2]. Ratkaisu todettiin hyväksi ja toimivaksi.

Demoversiossa ohjelma luki IED-laitteen tiedot kuten IP-osoitteen ja RCB-instanssien viitteet tietokannasta ja tallensi saapuneet viestit myös tietokantaan. Uudessa arkkitehtuurissa viestit julkaistiin erilliselle välittäjäpalvelimelle, jonka ansiosta tietokanta ei ollut enää tarpeellinen. C-ohjelman tarkoitus oli olla väliohjelma viestien välittämiseen eteenpäin, joten siihen ei tarvittu käyttöliitymää. Ohjelmasta päätettiin toteuttaa komentorivipohjainen, jolle kaikki tiedot voitaisiin syöttää komentoriviparametereillä käynnistyksen yhteydessä. Näillä muutoksilla tietokanta voitiin jättää kokonaan pois riippuvuuksista.

4.3 Suorituskyky ja kielen valinta

Demoversio oli ohjelmoitu Ruby-kielellä ja siinä oli paikoin suoritukseen liittyviä ongelmia ja epävarmuutta, etenkin viestien ja RCB-instassien määrän ollessa suurempi. Demon toimintaa ja ongelmia käytiin läpi aikaisemmin kappaleessa 3.2. Suorituskyvyn ja toiminnan epävarmuuden takia demo ei ollut tuotantovalmis, vaan vaati parannuksia. Ennen kuin päädyttiin kirjoittamaan koko ohjelma uudestaan eri tekniikalla, kokeiltiin demoa korjata vaihtamalla Ruby-tulkkia. Rubyn MRI-oletustulkki yritettiin vaihtaa *JRuby*-tulkkiin [17]. Tavoitteena vaihdossa oli saada Ruby-ohjelma toimimaan ilman globaalia tulkkilukitusta, eli GIL:ilä. GIL:iä käsiteltiin aikaisemmin kappaleessa 3.2. JRuby on Ruby-tulkki, joka suorittaa Ruby-koodia *Java-virtuaalikoneen* (engl. *Java Virtual Machine*, lyhennetään *JVM*) avulla. JRuby mahdollistaa säikeiden suorituksen rinnakkain JVM:n omilla säikeillä ja näin ollen suorituksen pitäisi olla nopeampaa [40]. Aidolla rinnakkaisuudella ohjelman suoritus ei olisi pysähtynyt viestin saapuessa takaisinkutsufunktion suorituksen ajaksi. LibEIC61850-kirjasto kutsuu asetettua takaisinkutsufunktioa viestin saapuessa, joka prosessoi viestin ja tallentaa sen tietokantaan. Tämä ei olisi kuitenkaan ratkaissut kaikkia ohjelmassa olevia ongelmia, kuten muistivuotoa ja hitaampaa suoritusta verrattuna käännettävään kieleen. Tämä toteutus ei kuitenkaan toiminut, ja nopean yrityksen jälkeen päätettiin palata suunnitelmaan kirjoittaa koko ohjelma uudestaan. Lisäksi demon arkkitehtuuri olisi silti pitänyt viedä saman suuntaan kuin kappaleessa 4.2 kuvattiin. Demo oli toteutettu osaksi isompaa RoR-projektia, joka toimi Rubyn-oletustulkin päällä. JRuby ei tukenut kaikkia projektin käyttämiä kirjastoja. Tämä oli pääsyynä miksi JRuby:n ei päädytty. Rubyssä kirjastoja kutsutaan *jalokiviksi* (engl. *gem*). Seurauksena olisi ollut saman

projektin ylläpitäminen kahdelle eri tulkille tai asennettavien kirjastojen erottaminen. Lopulta tähän vaihtoehtoon ei päädytty ja lisäksi täytyi ottaa huomioon implementaatioon käytetty aika ja useat mahdolliset viat jotka olisi silti pitänyt demosta korjata. Kaikkiaan oli helpompaa kirjoittaa ohjelma alusta kokonaan uudella tekniikalla. Samalla uudelle toteutukselle asetetut tavoitteet tiedettiin ja kuinka demossa olevat ongelmat pystyttiin välttämään. Demon uudelleenkirjoittamiseen ja korjaamiseen käytetty aika voitiin käyttää kokonaan uuden toteutuksen kirjoittamiseen.

Uuden toteutuksen kieleksi valittiin C-kieli. Isona syynä kielen valintaan oli tekijän iso mieltymys matalan tason ohjelmointiin ja C-kieleen. Lisäksi C-kieli käännetään alustalle suoraan konekäskyiksi, joiden suoritus on nopeampaa kuin tulkattavan kielen, kuten Ruby ja Python. Kielen valinnan yhteydessä kuitenkin oli hyvä varmistaa kaikkien suunniteltujen liitosten mahdollisuus. C-kielelle löytyi kirjastoja RabbitMQ-välittäjäpalvelimen käyttämiseen ja lisäksi JSON-rakenteen muodostamiseen. Hyötynä vielä C-kielen valinnasta oli, että demossa käytettyä libIEC61850-kirjastoa pystyttiin käyttämään suoraan ilman erillistä liitosta, koska kirjasto oli myös tehty C-kielellä. Tarkemmin käytettyihin kirjastoihin ja toteutukseen mennään kappaleessa 5.

Tutkimuskysymykseen jossa etsittiin vastausta siihen, mikä aiheutti demon ongelmia ja kuinka niitä voisi estää saatiin vastaus. Vastauksia syihin saatiin aikaisemmin kappaleessa 3.2. Valitsemalla eri tekniikat kuten C-kieli, vastataan osittain kysymykseen kuinka suorityskykyongelma ja toiminnan epävarmuus vältetään. Muistivuotoa vältetään huolellisella ohjelmoinnilla ja testaamisella. Tiedon jakamisen ongelma liittyy osin järjestelmän hajauttamisen kysymykseen. Molempiin kysymyksiin vastaus on ottaa käyttään erillinen välittäjäpalvelin viestien välitykseen järjestelmässä eri osapuolten kesken.

4.4 Prosessoidun viestin muoto ja rakenne

Saapuva viesti esitetään libIEC61850-kirjastossa *ClientReport*-struktuurin instanssina. Se sisältää viestin datan ja sen voi lukea käyttämällä kirjaston tarjoamia funktioita [21]. Saapunut viesti haluttiin jakaa välittäjäpalvelimen läpi muille osapuolille, joten viestin täytyi olla helposti luettavassa muodossa. Viesti päädyttiin muuttamaan JSON-rakenteeksi. JSON-rakenteen voi helposti ihminen lukea ja se on nykypäivänä paljon käytetty tiedonsiirtomuoto erilaisissa web-palveluissa ja rajapinnoissa. Myöskin JSON-rakenteiden lukemiseen on monelle eri kielellä olemassa valmiita kirjastoja sen monikäyttöisyyden takia [29].

Liitteessä A on esitetty C-ohjelman viestistä muodostettu JSON-rakente. Saman JSON:in C-ohjelmaa julkaisi RabbitMQ-välittäjäpalvelimelle. Standardin määrittämää viestin rakennetta ja sisältöä käytiin läpi kappaleessa 2.1.8. JSON:in rakenne noudattaa pääosin standardin määrittämää viestin rakennetta. C-ohjelma lisäsi viestiin attribuuteihin sen viitteen, tyypin ja koon. Nämä tiedot eivät ole standardin määrittämässä viestissä ja ne todettiin tarpeelliseksi tiedoksi muille järjestelmän osille jotka viestejä lukevat. Tiedot luetaan

ja selvitetään IED-laitteelta erillisillä palvelukutsuilla ennen tilauksen aloittamista. Nämä tiedot yhdistetään viestin saapuessa JSON-rakenteeseen. Tätä käsitellään tarkemmin kappaleessa 5.

Standardin viestin kenttien määrää pystyi säätämään RCB-instanssin *OptFlds*-attribuutilla. JSON:iin kuitenkin haluttiin lisätä kaikki mahdolliset kentät selkeyden vuoksi. Jos kenttä puuttui viestistä, asetettiin sen arvoksi JSON:issa *null*-arvo. Esimerkiksi liitteessä A kentän confRevision arvo on null. Tällöin RCB-instanssissa OptFlds-attribuutin *conf-revision*-bitti on ollut epätosi. Sama toistettiin kaikille muillekin vaihtoehtoisille kentille. Tällä periaatteella viestin OptFlds-kenttä voitiin jättää pois JSON:ista. JSON:iin päädyttiin lisäämään FCD- ja FCDA-viitteisiin sisällytetyt oikeat attribuutien viitteet, tyyppit ja koot. Näin voidaan päätellä mitkä arvot oikeasti kuuluvat viestiin ja mitkä ovat niiden viitteet. Standardissa viesti sisälsi vain datajoukon FCD- tai FCDA-viitteen ja taulukon arvoja mihin attribuutteihin sen alla viitattiin. JSON:iin tämä taulukon arvot on avattu ja jokaiselle taulukon arvolle on lisätty sen viite, tyyppi ja koko. Liitteessä A oleva JSON-rakenne koostuu kahdesta sisäkkäisestä values-taulukosta (rivit 7 ja 13). Rivillä 7 oleva values-taulukko sisältää viestissä olevat datajoukon FCD- tai FCDA-viitteet ja niihin liittyvät kentät. Samalla periaatteella, kuin standardin määrittämässä viestin rakenteessa taulukon arvot 1–n:ään (kuva 6). Eli viestin *Reason Code* on laitettu *reasonForInclusion* attribuuttiin. Viestin *DataRef*-kenttä on pilkottu kolmeen eri kenttään *mmsReference*, *reference* ja *functionalConstraint*. Viestien viitteet tulevat MMS-protokollamäärityksen muodossa, eli pisteet (.) on korvattu dollari-merkillä ($) ja viite sisältää funktionaalisen rajoitteen. Nyt mmsReference sisältää viestin alkuperäisen MMS-viitteen, reference sisältää standardin abstraktin viitteen ja functionalConstraint sisältää funktionaalisen rajoitteen. Nämä on erotettu selkeyden takia, koska mahdollisesti jotkin komponentit saattavat tarvita standardin käyttämää abstraktia viitettä ja näin välttää teksimuunnokset. JSON:in sisempi values-attribuutti (liitteessä A ensimmäinen rivillä 13) sisältää viestistä avatun arvotaulukon. Jokaisen taulukon arvoon on lisätty sen koko viite, tyyppi ja koko. Poikkeuksena ovat *boolean* ja *utc-time* tyypit, joilla ei ole kokoa. Koko kertoo monellako bitillä kyseinen attribuutti esitetään, ja se voi vaihdella saman tyypin välillä, esimerkiksi *bit-string*. Myöskin bit-string-tyypille päädyttiin lisäämään kaksi eri arvoa *valueLittleEndian* ja *valueBigEndian*. Attribuutista riippuen, sen tavujärjestys voi olla eri. Tämän takia päätettiin tarjota tilaajalle molemmat vaihtoehdot. Ajat päätettiin antaa suoraan samassa formaatissa kuin viestissä. Viestin päätason aikaleima on millisekunteja *UNIX*-ajanlaskun alusta 1. tammikuuta 1970 klo 00:00:00 UTC tähän hetkeen. Attribuuteissa tyypiltään utc-time, luku on sekunteja samasta UNIX-ajanlaskusta tähän hetkeen [12, s. 26–27].

Tutkimuskysymykseen jossa etsittiin vastausta kysymykseen mikä olisi sopiva tiedon jakamisen muoto hajautetussa järjestelmässä. Vastauksena kysymykseen tässä työssä valittiin JSON. JSON on nykypäivänä web-ohjelmoinnissa paljon käytetty tiedon jakamisen muoto rajapinnoissa. Toinen vaihtoehto olisi ollut XML-muoto, mutta se on raskaampi kuin JSON ja ei niin helposti ihmisen luettavissa. JSON-muodon on ihmiselle helposti luettavissa ja ymmärrettävissä. Lisäksi JSON:in lukuun monelle eri kielelle on olemassa

valmiita kirjastoja sen yleisyyden takia. JSON:in valinta todettiin hyväksi ja toimivaksi ratkaisuksi. [38, 39]

5. TOTEUTUS

Tässä osiossa käydään läpi kappaleessa 4 suunniteltun ohjelman toteuttaminen. Toteutus alkaa yleiskuvalla sen komponenteista ja niiden toiminnasta. Yleiskuvan jälkeen mennään tarkemmin ohjelman yksityiskohtiin kuten kirjastoihin ja niiden toimintaan. Lopuksi mietitään jatkokehitysideoita, eli mitä olisi voinut lisätä, tehdä toisin ja mahdollisia puutteita.

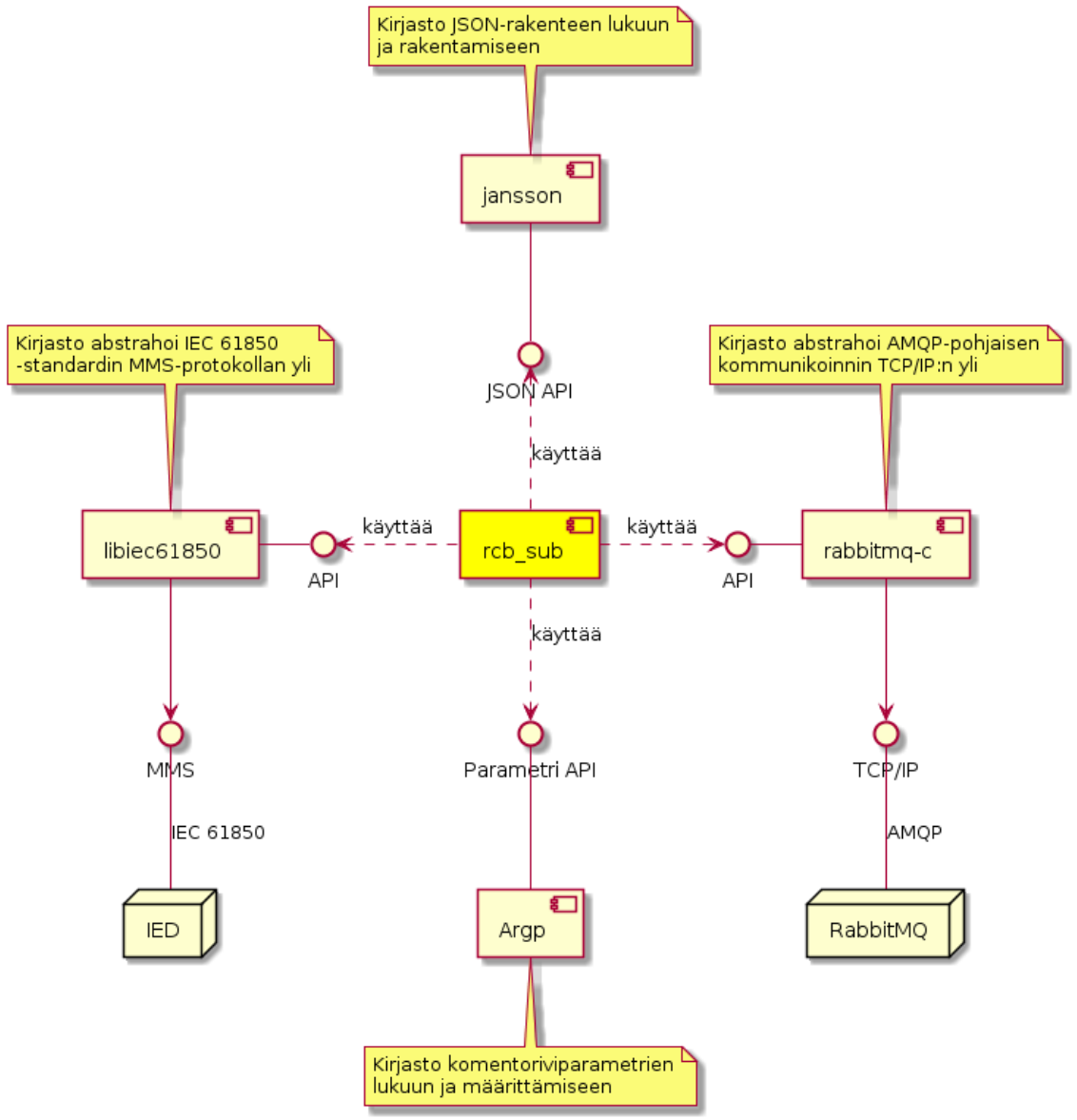
5.1 Yleiskuva

Työssä toteutetiin komentorivipohjainen ohjelma C-kielellä. Ohjelman tarkoitus oli tilata IED-laitteen viestit ja prosessoida ne JSON-muotoon RabbitMQ-palvelimelle. RabbitMQ:lta muut ohjelmat pystyivät tilaamaan JSON-viestejä. Kuvassa 19 on esitetty komponenttikaavio toteutetusta ohjelmasta ja siihen käytetyistä kirjastoista. Toteutettu komponentti on kuvassa keskellä keltaisella ja nimeltään *rcb_sub*. Kuvasta voi nähdä miten eri komponentit ovat relaatiossa keskenään rcb_sub-ohjelman kanssa. Kuvassa on myös esitetty IED-laite ja RabbitMQ-palvelin.

Toteutuksessa käytettiin seuraavia kirjastoja:

- *libIEC61850*,
- *rabbitmq-c*,
- *jansson*, ja
- *Argp*.

Kaikki käytetyt kirjastot on toteutettu C-kielellä, kuten rcb_sub. Kirjastojen tarkoitus on abstrahoida jonkin asian käyttö, ja tarjota käyttäjälle siitä helppokäyttöinen ja ymmärrettävä rajapinta. Rajapintaa käyttämällä kirjasto hoitaa matalan tason toiminnan ilman, että sen käyttäjän tarvitsee siitä välittää. LibIEC61850-kirjasto abstrahoi IEC 61850 -standardin käyttöä ja hoitaa matalan tason MMS-protokollan kommunikoinnin [27]. Samaa kirjastoa käytettiin demoversiossa (kappale 3.1) ja kirjaston kerrosarkkitehtuuri esitettiin aikaisemmin kuvassa 14. Kuvassa 19 libIEC61850 kommunikoi suoraan IED-laitteen kanssa MMS-protokollaa käyttäen. Rabbitmq-c-kirjasto abstrahoi RabbitMQ-palvelimen käyttöä ja hoitaa matalan tason AMQP-pohjaisen kommunikoinnin [30]. Toteutuksessa rabbitmq-c kommunikoi suoraan RabbitMQ-palvelimen kanssa. Jansson-kirjasto abstrahoi JSON-rakenteiden lukua ja käsittelyä C-kielelle [6]. Kirjastoa käytettiin rakentamaan IED-laitteelta saapuneesta viestistä JSON-muotoinen viesti. JSON-rakenne on nähtävissä liitteessä A. Argp-kirjasto auttaa ohjelman komentoriviparametrien määrittämisessä ja käsittelyssä [28]. Kirjasto auttaa toteuttamaan ohjelmalle *UNIX*-tyyliset parametrit. Eli vaaditut parametrit ja vaihtoehtoiset lyhyet ja pitkä parametrit. Vaadituista parametreista



Kuva 19. *Toteutuksen komponenttikaavio sen osista ja relaatioista toisiinsa.*

esimerkiksi Linuxin komento `mv foo.txt bar.txt`, jossa *foo.txt* ja *bar.txt* ovat vaadittuja parametreja. Vaihtoehtoisista parametreista esimerkkinä pitkä muoto `--bytes` ja lyhyt muoto `-b`. Lisäksi kirjasto lisää ohjelmaan automaattisesti Linuxista käyttäjille tutut `--help` ja `--version` vaihtoehtoiset parametrit. Komennolla `--help` kirjasto tulostaa Linuxilta tutun ohjelman aputekstin käyttäjälle, jossa on esitetty ohjelman kaikki parametrit ja niiden selitteet [4].

Kuvassa 20 on esitetty rcb_sub-ohjelman sekvenssikaavio pääpiirteisestä toiminnasta. Toteutus noudattaa suurinpiirtein samoja periaatteita kuin demo (kuva 15). Tässä kohtaa käydään läpi ohjelman pääpiirteinen toiminta ja myöhemmin tarkemmin läpi kappaleessa 5.2. Ensin ohjelman suoritus alkaa lukemalla annetut parametrit Argp-kirjastolla (kohdat 1–2). Parametreissa tulee tiedot yhteyden muodostamiseen IED-laitteelle ja RabbitMQ-palvelimelle (kohdat 3–6). Parametreissa on myös tiedot RCB-instansseista jotka halu-

taan IED:ltä tilata. Yhteyksien muodostamisen jälkeen jokainen parametrina annettu RCB käydään läpi silmukassa ja sen arvot ja datajoukon viitteet luetaan IED:ltä (kohdat 7–12). Tämän jälkeen sisäkkäisessä silmukassa luetaan datajoukon viitteiden muuttujien *spesifikaatiot* (kohdat 11–12). Spesifikaatio antaa tiedot muuttujien pituudesta ja tyypistä. Näitä tietoja käytettiin JSON-rakenteessa täydentämään viestiä (esimerkkinä liiteessä A rivit 21–22). Tämän jälkeen tehdään toinen silmukka, jossa jokainen RCB-instanssi tilataan ja niille asetetaan takaisinkutsufunktio (kohdat 13–16). Arvojen kirjoitushetkellä (kohta 15) RCB varataan ja se aloittaa viestien lähettämisen rcb_sub-ohjelmalle. Jokaisen RCB:n kirjoituksen jälkeen ohjelma jää loputtomaan silmukkaan ottamaan viestejä vastaan (kohdat 17–22). Viestin saapuessa kutsutaan asetettua takaisinkutsufunktiota, jonka parametrina on saapunut viesti (kohta 17). Viesti muutetaan JSON-muotoon jansson-kirjastolla ja julkaistaan RabbitMQ-palvelimelle rabbitmq-c-kirjastolla (kohdat 18–21).

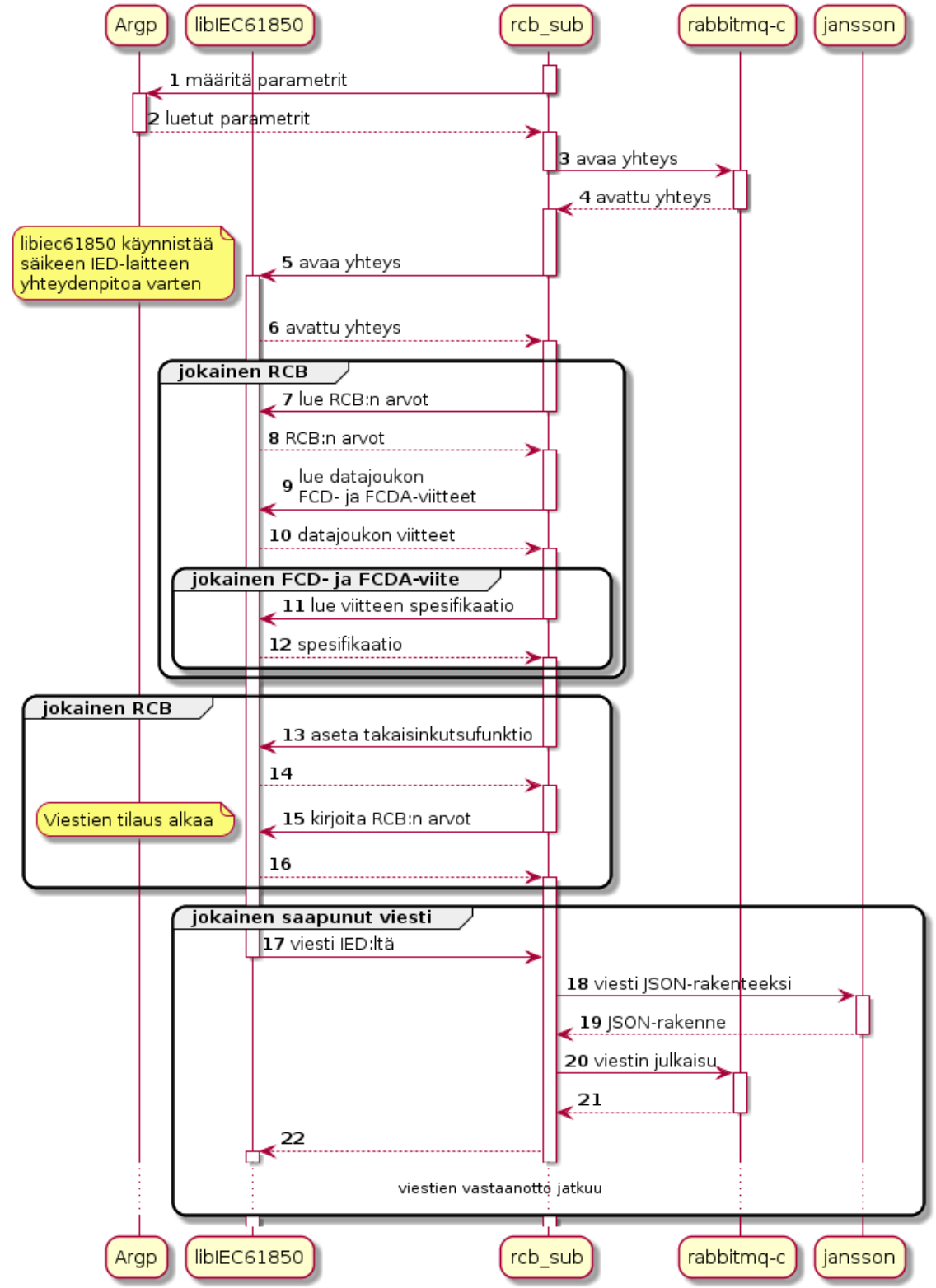
5.2 Ohjelman toiminta

Tulevissa kappaleissa käydään läpi yksityiskohtaisemmin rcb_sub-ohjelman toimintaa, joka esiteltiin pääpiirteittäin kappaleessa 5.1. Kappaleiden järjestys noudattaa kuvassa 20 olevan sekvenssikaavion järjestystä. Toisin sanoen ohjelmaa käydään tarkemmin läpi sen suorituksen järjestyksessä.

5.2.1 Parametrisointi

Ohjelma parametrisoitiin Argp-kirjastolla. Kirjasto tarjoaa rajapinnan komentoriviparametrien käsittelyyn ja määrittämiseen. Parametrien muodot ovat tutut muista Linux-käyttöjärjestelmän parametreista ja samaa periaatetta käytettiin tässäkin ohjelmassa. Kirjasto myös lisäsi ohjelmaan automaattisesti aputekstin käyttäjää varten. Aputeksti sisältää tietoa ohjelman parametreista ja niiden käytöstä. Aputekstin pystyi tulostamaan parametrilla `--help`. Liitteessä B on esitetty miltä ohjelman aputeksti näyttää. Liitteestä voi myös nähdä kaikki ohjelman parametrit ja lyhyen selityksen mihin kutakin käytetään.

Ohjelmiston parametrien voidaan ajatella koostuvan kolmesta eri ryhmästä. Ensin päätason vaihtoehtoiset parametrit `OPTIONS`. Pakolliset parametrit `EXCHANGE` ja `ROUTING_KEY`. Viimeisenä n-kappaletta `RCB_REF` ja `RCB_OPTIONS` parametreja ryhmissä. Suurin osa `OPTION` parametreista on itsestäänselviä. Esimerkkinä `--amqp-host`, joka kertoo AMQP-palvelimen IP-osoitteen, ja `--ied-host`, joka kertoo IED-laitteen IP-osoitteen. Parametrit `EXCHANGE` ja `ROUTING_KEY` määrittävät nimet RabbitMQ-palvelimen vaihteelle ja reititysavaimelle. Ryhmässä ensimmäinen `RCB_REF` määrittää viitteen tilattavaan RCB-instanssiin IED-laitteella. Tätä seuraa vaihtoehtoinen `RCB_OPTIONS` parametri. Se määrittää arvot, jotka kirjoitetaan edeltävä RCB-instanssille ennen tilausta. RCB-instanssin parametri `RCB_OPTIONS` määrittää käytetyt vaihtoehtoiset kentät (`--opt-fields`), käytetyt liipaisimet (`--trigger`) ja pyydetäänkö yleistä kyselyä ennen muita viestejä (`--gi`). Liipaisimet ja vaihtoehtoiset kentät asetetaan numeerisella arvoilla, jotka löytyvät myös aputekstistä (liite B). Numeerisia arvoja voidaan summata yhteen, jotta voidaan

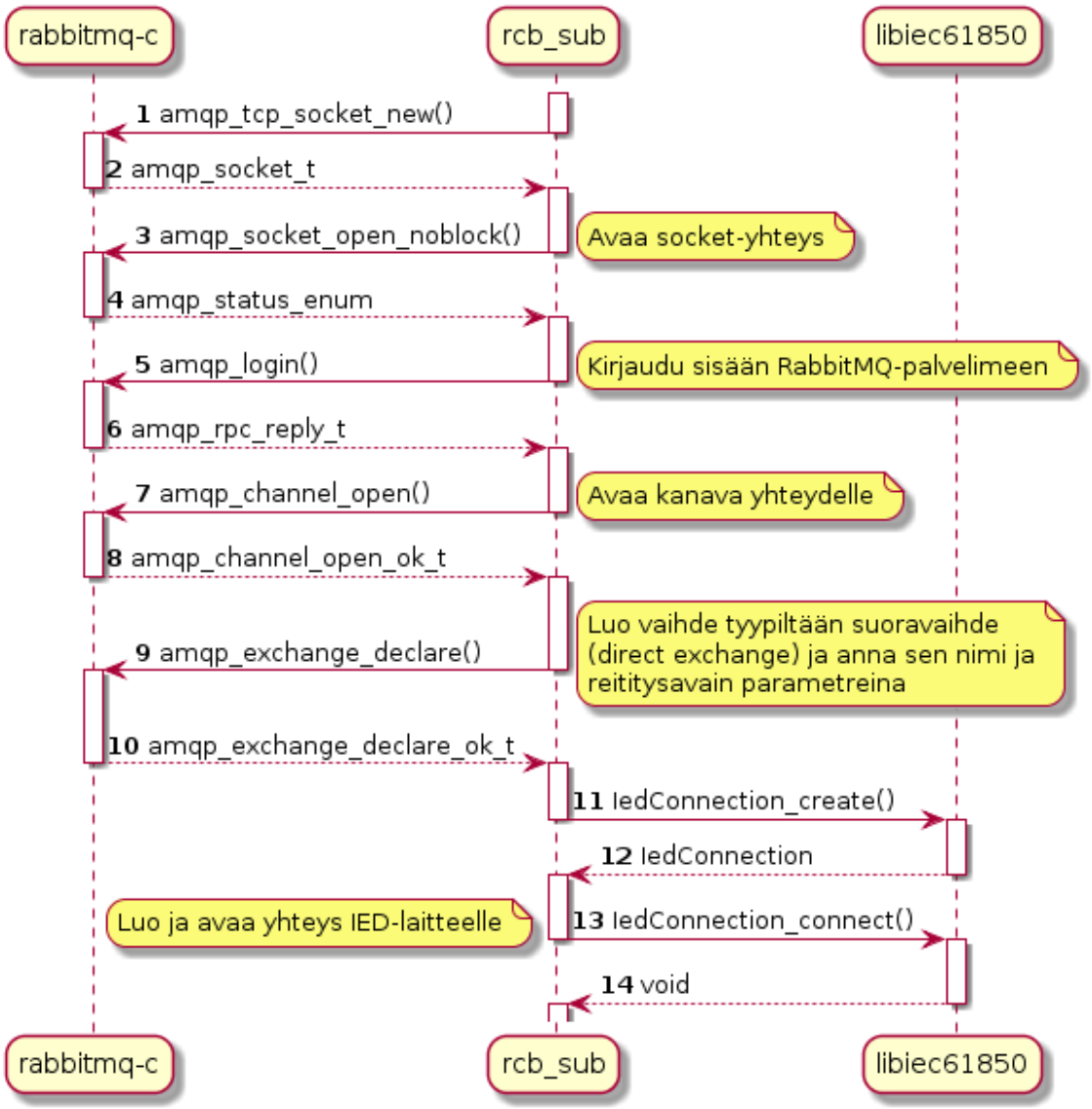


Kuva 20. *Sekvenssikaavio rcb_sub-ohjelman kokonaistoiminnasta.*

asettaa monta arvoa yhtä aikaa. Liipaisimien nimet vastaavat aikaisemmin kappaleessa 2.1.7 esitettyjä arvoja ja numeeriset arvot tulevat libIEC61850 -kirjastosta. Vaihtoehtoisten kenttien nimet vastaavat aikaisemmin taulukossa 8 esitettyjä arvoja ja numeeriset arvot tulevat myös libIEC61850-kirjastosta.

5.2.2 Yhteyksien muodostus

Parametrien luvun jälkeen ohjelma muodosti yhteydet ensin RabbitMQ-palvelimelle ja sen jälkeen IED-laitteelle. Kuvassa 21 on esitetty sekvenssikaavio, joka näyttä mitä kirjaston funktioita ohjelma kutsuu missäkin järjestyksessä. Funktiot ja niiden parametrit voi tarkemmin tarkistaa kirjaston omasta dokumentaatiosta. Tämä tarkentaa yleiskuvasta 20 kohdat 3–6. Kaaviossa ohjelma muodostaa yhteydet vain kerran. Ohjelma on kuitenkin toteutettu niin, että se yrittää muodostaa yhteydet uudestaan vikatilanteissa. Jos muodostus ei onnistu, ohjelma kirjoittaa lokin tapahtuneesta ja odottaa hetken ennen uudelleen yritystä.



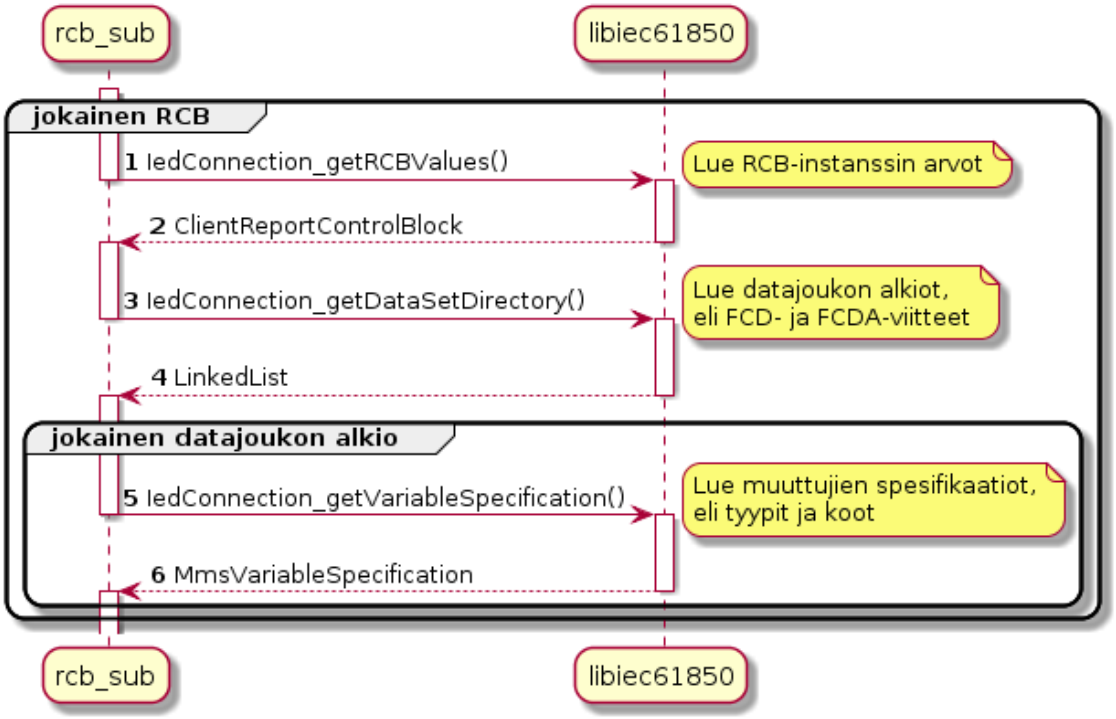
Kuva 21. *Sekvenssikaavio kuinka rcb_sub avaa yhteydet RabbitMQ-palvelimelle ja IED-laitteelle.*

Yhteyden avauksen ja sisäänkirjautumisen jälkeen ohjelma avaa kanavan kohdassa 7–8. Kanava on yhteyden päälle avattu oma erillinen kommunikointiväylä, joka ei sotkeudu muihin kanaviin. Yhteen avattuun yhteyteen voi olla avattuna monta eri kanavaa. Kanavat

mahdollistavat monen eri säikeen jakaa sama yhteys, ilman että tieto voi vuotaa toiseen säikeeseen. Kohdassa 9 kutsutaan funktiota `amqp_exchange_declare()`. Funktio määrittää vaihteen tyyppiä suoravaihde RabbitMQ-palvelimelle. Suoravaihde käsiteltiin kappaleessa 2.2.3. Ohjelmaan ei toteutettu parametria vaihdetyypin määrittämiseen, koska katsottiin että suoravaihde on riittävä nykyisten vaatimusten täyttämiseksi. Tulevaisuudessa voidaan tarvittaessa lisätä parametrit vaihdetyypin vaihtamiseen.

5.2.3 IED:n attribuuttien tyyppin ja koon luku

Yhteyksien muodostamisen jälkeen ohjelma käy läpi silmukassa jokaisen parametrina annetun RCB:n viitteen. Lukee RCB:n datajoukon viitteet ja selvittää jokaisen viitatun attribuutin spesifikaatiot, eli sen oikean viitteen, tyypin ja koon. Kuvassa 22 on esitetty sekvenssikaavio kuinka rcb_sub tämän tekee libIEC61850-kirjaston avulla. Kuva tarkentaa yleiskuvassa 20 kohtia 7–12.



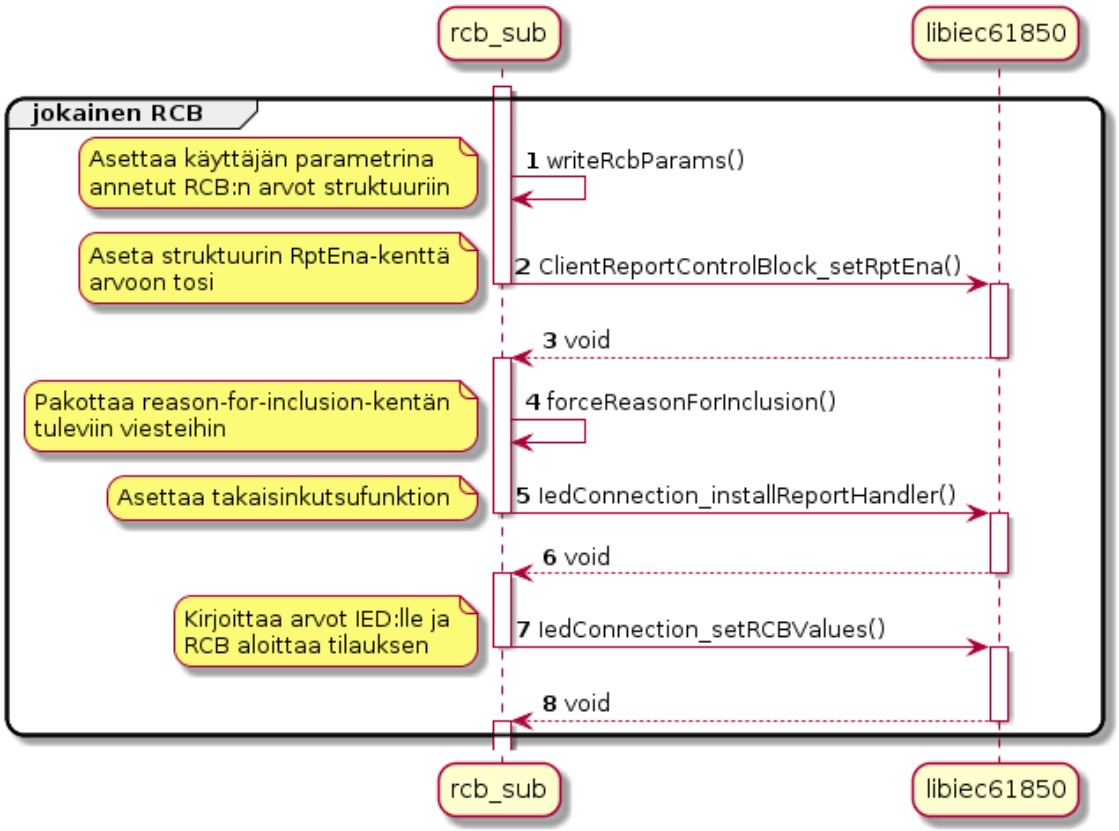
Kuva 22. *Sekvenssikaavio kuinka rcb_sub lukee RCB-instanssin arvot ja muuttujien spesifikaatiot.*

Ensin RCB:sta luetaan sen tiedot IED-laitteelta (kohdat 1–2). RCB:ltä saadaan tieto mihin datajoukkoon se on liitetty. Tätä käsiteltiin kappaleessa 2.1.7 ja taulukossa 7 kenttä *DatSet*, joka kertoo käytetyn datajoukon viitteen. Tällä tiedolla ohjelma voi lukea datajoukon FCD- ja FCDA-viitteet (kohdat 3–4). Tästä saadaan jokainen viite listassa, joka käydään läpi silmukassa kohdissa 5–6. Jokaiselle viitteelle luetaan sen spesifikaatio. Spesifikaatiorakenne sisältää sisäkkäisiä spesifikaatioita, jos viite viittaa moneen muuttujaan IED-laitteen hierarkiassa. Tämä tapahtuu samalla periaatteella, jolla FCD- ja FCDA-viitteet

viittaavaat moneen muuttujaan hierarkiassa alaspäin. Kuinka FCD- ja FCDA-viitteet toimivat käsiteltiin kappaleessa 2.1.5. Jokainen luettu viite tallennetaan ja niitä käytetään myöhemmin viestin kanssa JSON-rakenteessa. Esimerkkinä liitteessä A riveillä 21–22 tyyppi ja koko -tiedot.

5.2.4 Viestien tilaus

Ohjelman luettua kaikki muuttujien spesifikaatiot. Ohjelma tilaa silmukassa kaikki parametrina annetut RCB-instanssit. Kuvassa 23 on esitetty sekvenssikaavio, kuinka rcb_sub tilaa RCB-instanssit libIEC61850-kirjaston avulla. Kuvan tarkentaa yleiskuvassa 20 kohtia 13–16.



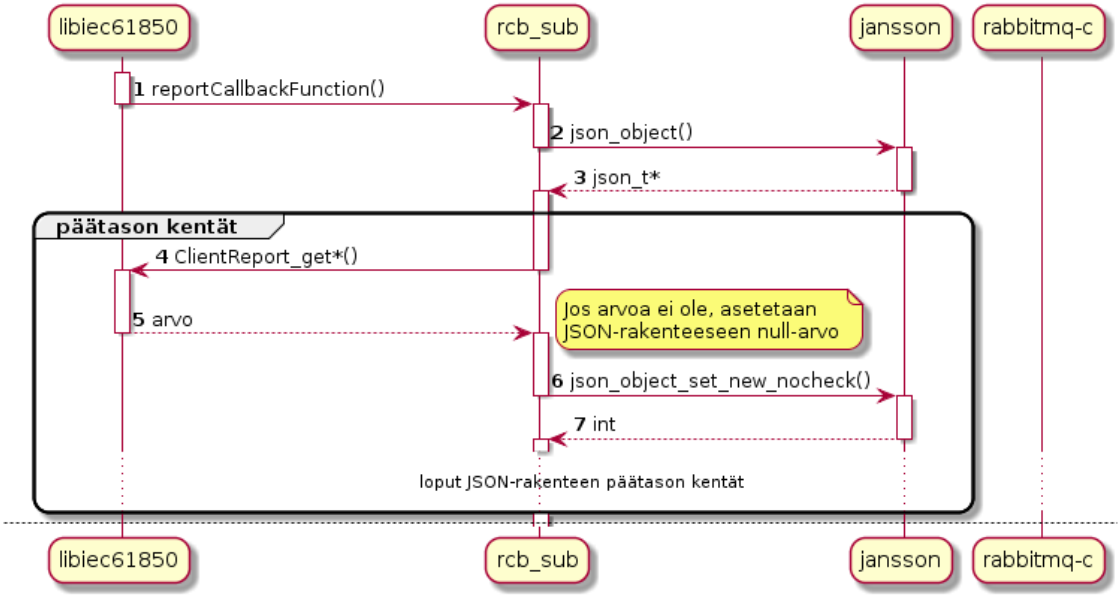
Kuva 23. *Sekvenssikaavio kuinka rcb_sub tilaa RCB-instanssit.*

Ohjelma käsittelee libIEC61850-kirjaston tarjoamaa *ClientReportControlBlock* struktuurin instanssia. Kirjasto palauttaa struktuurin instanssin, kun RCB:n arvot luetaan IED-laitteelta. Kaikki RCB:lle kirjoitettavat arvot asetetaan instanssiin ennen IED-laitteelle kirjoitusta. Näitä arvoja ovat ohjelmalle parametreina annetut arvot, kuten liipaisimet ja vaihtoehtoiset kentät. Tämän ohjelma tekee kutsumalla omaa funktiota `writeRcbParams()` (kohta 1). Tämän jälkeen ohjelma asettaa RCB:n *RptEna*-kentän arvoksi tosi (kohdat 2–3). Tämä kenttä kontrolloi RCB-instanssin varausta ja onko tilaus päällä. Seuraavaksi ohjelma pakottaa viestiin vaihtoehtoisen kentän *reason-for-inclusion* (kohta 4). Tätä kenttää tarvitaan, jotta aikaisemmin luetut spesifikaatiotiedot saadaan yhdistettyä

saapuneeseen viestiin. Tämän jälkeen asetetaan takaisinkutsufunktio, jota kirjasto kutsuu kun viesti saapuu (kohdat 5–6). Viimeisenä struktuurin arvot kirjoitetaan IED:llä olevalle RCB:lle (kohdat 7–8). Tämä varaa RCB-instanssin kirjoittavalle asiakkaalle, ja aloittaa tilauksen jos RptEna-kentän arvo oli tosi. RCB tulee lähettämään viestejä ohjelmalle samalla kun silmukan muilla kierroksilla käsitellään tilaamattomia RCB-instansseja.

5.2.5 JSON:nin muodostaminen ja julkaisu

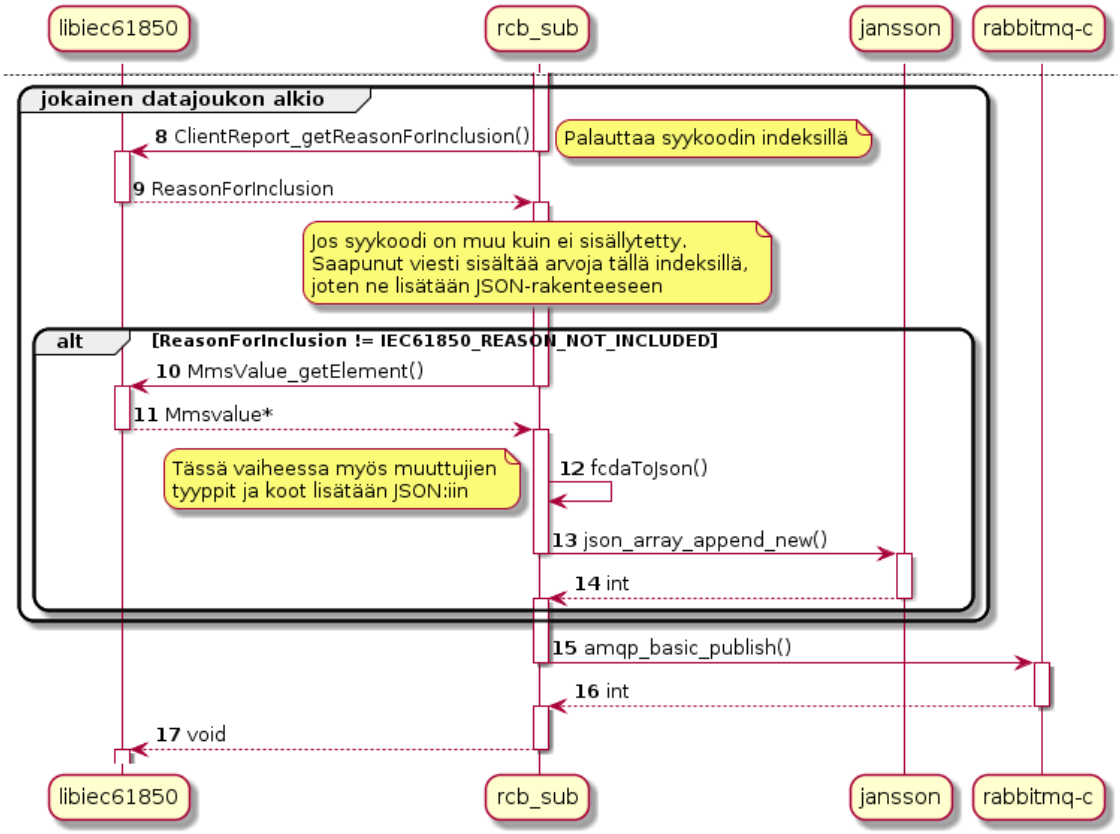
Viestin saapuessa libIEC61580-kirjasto kutsuu asetettua takaisinkutsufunktiota. Takaisinkutsufunktio muuttaa viestin JSON-muotoon ja lisäsi siihen aikaisemmin luetut muuttujien oikeat viittet, tyypit ja koot. Tämän jälkeen JSON-julkaistiin RabbitMQ-palvelimelle. Kuvissa 24 ja 25 on esitetty sekvenssikaaviolla kuinka ohjelma muuttaa viestin JSON:iksi ja julkaisee RabbitMQ:lle. Kuva 24 jatkuu kuvassa 25. Kuva 24 tarkentaa yleiskuvan 20 kohtia 17–19 ja kuva 25 kohtia 20–22.



Kuva 24. *Sekvenssikaavio kuinka rcb_sub muodostaa JSON:nin päätason kentät.*

Kuvassa 24 suoritus alkaa kun libIEC61850-kirjasto kutsuu takaisinkutsufunktiota. Funktiolle annetaan parametrina saapunut viesti *ClientReport*-struktuurin instanssina (kohta 1). Tämän jälkeen ohjelma käy läpi viestin jokaisen päätason kentän ja lisää ne JSON-rakenteeseen. Osa viestin kentistä on vaihtoehtoisia riippuen siitä, mitä käyttäjä asetti `--opt-fields` parametrilla. Jos arvoa viestissä ei ole, korvataan se null-arvolla JSON:iin. Esimerkkinä liiteessä A rivillä 4 oleva `confRevision` muuttuja, jonka arvo on null. Tämän jälkeen suoritus jatkuu kuvasta 24 kuvaan 25.

Päätason viestin kenttien jälkeen ohjelma käy läpi silmukassa viestin datajoukon indeksit (kuvassa 25 kohdat 8–14). Viesti oikeasti sisältää vain ne datajoukon alkiot, jotka sisältyivät viestiin. Ongelmana tässä on se, että viesti ei sisällä indeksiä tai tietoa siitä mikä datajoukon alkio on kyseessä. Jotta tästä saadaan tieto, ohjelma pakottaa syy-



Kuva 25. *Sekvenssikaavio kuinka rcb_sub lisää JSON:iin muuttujat viestistä.*

koodin päälle viestiin. Tämän avulla kun silmukassa käydään kaikki datajoukon indeksit läpi, voidaan jokaiselle indeksille ensin kysyä syykoodi viestistä (kohdat 8–9). Jos datajoukon alkio ei ole viestissä, palauttaa kirjaston funktio `ClientReport_getReasonForInclusion()` arvon `IEC61850_REASON_NOT_INCLUDED`. Tätä tietoa voidaan käyttää löytämään oikea datajoukon indeksi. Jos datajoukon indeksi on viestissä, suoritetaan kohdat 10–14, muuten mennään seuraavaan indeksiin ja toistetaan kohdat 8–9. Datajoukon indeksi tarvitaan, jotta aiemmin luetut spesifikaatiot saadaan yhdistettyä attribuutteihin arvojen kanssa. Datajoukon indeksillä, viestin arvoilla ja muuttujien tyypeillä ja koolla saadaan rakennettua loppuosa JSON-rakenteesta. Kuvassa 25 oleva silmukka rakentaa liitteessä A olevan values-taulun alkaen riviltä 7. JSON:in sisempi values-taulu (rivi 13) on lista FCD- tai FCDA-viitteen muuttujia, mitä se viittaa arvoineen. Tämä taulukko muodostetaan kuvan 25 kohdassa 12 funktiolla `fcdaToJson()` ja lisätään JSON:iin kohdassa 13. Lopuksi viesti lähetetään RabbitMQ-palvelimelle funktiolla `amqp_basic_publish()` ja takaisinkutsufunktio palaa (kohdat 15–17).

5.3 Jatkokehitys

Ohjelma jätettiin työssä pisteeseen, missä se saavutti kaikki sille asetetut vaatimukset. Kuitenkin tulevaisuudessa ohjelmaa voidaan lisätä ominaisuuksia tarpeen vaatiessa. Isoin puute ohjelmassa oli testiympäristö ja sen yksikkötestit. C:ssä ei ole suoraan tukea yksik-

kötestien kirjoittamiseen. Ympäristön pystytys vaatii erillisen kirjaston projektin yhteyteen, millä yksikkötestit kirjoitetaan. Tämä jäi tulevaisuuden kehitystyöksi ja ei sisältynyt tähän työhön. Yksikkötestit ovat kuitenkin tärkeä osa ohjelman ylläpitoa ja toiminnan varmistamista muutosten jälkeen. Testit tullaan tarvitsemaan ennemmin tai myöhemmin.

Ohjelma toteutettiin nyt niin, että se aina käyttää suoraa vaihdetyyppiä RabbitMQ-palvelimella. Tämä täytti työlle asetetut vaatimukset. Jos tulevaisuudessa tarvitaan joustavuutta, voidaan ohjelmaan tehdä muutoksia ja parametreja lisätä helposti lisämään toiminnallisuutta. Esimerkkinä käyttäjä voisi valita käytettävän vaihteen tyypin parametrilla.

6. YHTEENVETO

Tässä diplomityössä lähdettiin etsimään ratkaisua kuinka suunnitellaan ja toteutetaan yksittäinen komponentti osaksi isompaa järjestelmää, joka pystyisi jakaamaan tietoa sen muiden komponenttien kanssa. Lähtökohtana oli, että komponentti pystyisi tilaamaan tietoa sähköseman IED-laitteelta IEC 61850 -standardin mukaisesti. Lisäksi tieto täytyi saada jaettua järkevästi järjestelmän muille komponenteille, ilman että päädyttäisiin huonoihin teknisiin ratkaisuihin. Ennen diplomityön aloitusta tekijä oli yrityksessä toteuttanut demon komponentin toimivuudesta. Demo oli samalla tie oppia IEC 61850 -standardin toiminta ja perehtyä aiheeseen tarkemmin ennen oikeaa toteutusta. Tämä diplomityö keskittyi komponentin uuden version suunnitteluun ja toteutukseen käyttäen hyväksi demototeutuksen analyysia toiminnallisuudesta ja ongelmista. Lisäksi työn alussa asetettiin tutkimuskysymyksiä aiheeseen liittyen, joihin pyrittiin etsimään vastausta. Tutkimuskysymykset esiteltiin työn alussa kappaleessa 1.

Työssä etsittiin ratkaisua siihen, mitkä olisivat sopivat ohjelmiston arkkitehtuurimallit tämän kaltaisen ongelman ratkaisuun. Toteutuksessa päädyttiin käyttämään tilaaja-julkaisija-arkkitehtuurimallia, missä toteutettu komponentti on julkaisija ja muut järjestelmän komponentit ovat tilaajia. Sähköaseman IED-laitteen ja komponentin välisen tiedonjaon täytyy automaattisesti noudattaa tilaaja-julkaisija-arkkitehtuuria, koska IEC 61850 -standardissa määritetään niin. Tähän toteutusmalliin päädyttiin, koska järjestelmän muut komponentit tarvitsevat tietoa samalla periaatteella kuin ulkopuolinen ohjelma tarvitsee tietoa IED-laitteelta. Periaatteessa komponentti jatkaa IED-laitteen ja komponentin välistä tilaaja-julkaisija-mallia ohjelmiston muille komponenteille välittäjäpalvelimen avulla. Lisäksi arkkitehtuurimalli on tarkoitettu ratkomaan tämän kaltaisia tilanteita, jossa toinen osapuoli tarvitsee tietoa halutessaan ja saa siitä tiedon kun uusi tieto on saatavilla. Näin saatiin aikaiseksi tiedon kulku järjestelmässä yhteen suuntaan ilman kyseenalaisia teknisiä toteutuksia. Demossa viestien jako oli toteutettu käyttäen tietokantaa tiedon välittäjänä, mikä ei ollut hyvä ratkaisu. Työn toteutuksessa viestien välitykseen valittiin RabbitMQ-välittäjäpalvelin. Tämän avulla järjestelmän muut komponentit pystyivät tilaamaan viestejä tarpeidensa mukaan ja saivat siitä huomautuksen kun uusi viesti saapui. Lisäksi palvelin tarjosi sisäisesti jonot tilaajille, jos ne eivät ehtineet prosessoida viestejä tarpeeksi nopeasti. RabbitMQ-välittäjäpalvelimena viestien välitykseen järjestelmän muille komponteille osoittautui järkeväksi ja toimivaksi vaihtoehdoksi.

Viestin muoto IED-laitteelta noudattaa IEC 61850 -standardin määrittämää muotoa. Toteutuksessa tämän viestin lukeminen hoidettiin libIEC61850-kirjastolla. Jotta tieto saataisiin järkevästi järjestelmän muille komponenteille, täytyi se muuttaa johonkin muuhun helppokäyttöisempään muotoon. Jos järjestelmän osia toteutetaan eri tekniikoilla, vies-

tin luku olisi mahdollista tekniikasta riippumatta. Toteutuksessa viesti päädyttiin muuttamaan JSON-muotoon. JSON on nykypäivänä paljon käytetty tiedonvälityksen muoto rajapinnoissa verkko-ohjelmointiin liittyen. Se on helppo luettava ihmiselle ja JSON-rakenteen lukemiselle löytyy toteutus monelle eri tekniikalle valmiiksi. Toteutuksessa valinta osoittautui hyväksi ja toimivaksi. Ohjelman tekemä JSON-muoto on nähtävissä liitteessä A.

Osa työtä oli demototeutuksen ongelmien analysointi ja sen selvittäminen kuinka ne vältettäisiin uudessa versiossa. Ongelmia oli mm. huono suorituskyky, muistivuoto ja toiminnan epävarmuus. Näitä ongelmia analysoitiin työssä syvällisesti kappaleessa 3.2. Suorituskykyä saatiin toteutuksessa paremmaksi valitsemalla suorituskykyisempi C-kieli. C on käännettävä kieli verrattuna Rubyn tulkattavaan kieleen. Lisäksi C voi hyödyntää käyttöjärjestelmän säikeitä ilman rajoituksia verrattuna Ruby-tulkin globaaliin lukitukseen (GIL). Muistivuoto saatiin korjattua huolellisella ohjelmoinnilla ja varmistamalla, että muisti varmasti vapautettiin kun sitä ei enää tarvittu. Toiminnan epävarmuus liittyi yhteyden aikakatkaisuihin, joka johtui taas huonosta suorituskyvystä. Tästä päästiin eroon kun kieli vaihdettiin nopeampaan. Työn toteutuksessa ei ollut demossa havaittavia ongelmia ja on osoittanut tuotannossa toimivaksi muun järjestelmän kanssa.

Tärkeimpänä puutteena toteutuksessa oli testiympäristön ja yksikkötestien puuttuminen. Testit olisivat tärkeät ohjelman jatkoa ja ylläpidettävyyttä ajatellen. Etenkin tulevaisuudessa kun siihen tullaan tekemään muutoksia, voidaan yksikkötesteillä varmistaa että ohjelma toimii ainakin niiltä osin halutulla tavalla. Tällä hetkellä muutoksien jälkeen ohjelma testata käsin. Testejä ei kirjoitettu tämän työn puitteissa, koska aika ei siihen riittänyt. Testit kuitenkin tullaan lisämään ohjelmaan myöhemmin osana muun järjestelmän testiajoa.

Diplomityössä tuloksena oli järjestelmään erillinen komponentti, joka kykeni tilaamaan viestit yhdeltä IED-laitteelta ja jakamaan tiedon järjestelmän muiden komponenttien kanssa. Kaikki ongelmat mitä demoversiossa oli, saatiin ratkaistua onnistuneesti. Ohjelmisto otettiin käyttöön muun järjestelmän kanssa tuotantoon. Voidaan sanoa, että työ pääsi asetettuihin tavoitteisiin ja onnistui niiltä osin hyvin. Lisäksi asetettuihin tutkimuskysymyksiin löydettiin vastaukset, jotka todettiin toimiviksi.

LÄHTEET

[1] AMQP 0-9-1 Model Explained, RabbitMQ verkkosivu. Saatavissa (viitattu 31.7.2018): https://www.rabbitmq.com/tutorials/amqp-concepts.html

[2] AMQP Advanced Message Queuing Protocol v0-9-1, Protocol Specification, mar. 2008, 39 s. Saatavissa (viitattu 10.7.2018): http://www.amqp.org/specification/0-9-1/amqp-org-download

[3] AMQP kotisivu, AMQP verkkosivu. Saatavissa (viitattu 23.9.2018): http://www.amqp.org/

[4] B. Asselstine, Step-by-Step into Argp, Askapache verkkosivu, 2010, 75 s. Saatavissa (viitattu 1.9.2018): http://nongnu.askapache.com/argpbook/step-by-step-into-argp.pdf

[5] C. Brunner, IEC 61850 for power system communication, teoksessa: 2008 IEEE/-PES Transmission and Distribution Conference and Exposition, April, 2008, s. 1–6.

[6] C library for encoding, decoding and manipulating JSON data, GitHub verkkosivu. Saatavissa (viitattu 1.9.2018): https://github.com/akheron/jansson

[7] B. E. M. Camachi, O. Chenaru, L. Ichim, D. Popescu, A practical approach to IEC 61850 standard for automation, protection and control of substations, teoksessa: 2017 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), June, 2017, s. 1–6.

[8] C. George, J. Dollimore, T. Kindberg, G. Blair, Distributed Systems: Concepts and Design, Addison-Wesley, 2012, 1047 s.

[9] IEC 61850-1 Communication networks and systems for power utility automation – Part 1: Introduction and overview, International Electrotechnical Commission, International Standard, maa. 2013, 73 s. Saatavissa (viitattu 15.6.2018): https://webstore.iec.ch/publication/6007

[10] IEC 61850-6 Communication networks and systems for power utility automation – Part 6: Configuration description language for communication in electrical substations related to IEDs, International Electrotechnical Commission, International Standard, jou. 2009, 215 s. Saatavissa: https://webstore.iec.ch/publication/6013

[11] IEC 61850-7-1 Communication networks and systems in substations - Part 7-1: Basic communication structure for substation and feeder equipment - Principles and models, International Electrotechnical Commission, International Standard, hei. 2003, 110 s. Saatavissa (viitattu 16.5.2018): https://webstore.iec.ch/publication/20077

[12] IEC 61850-7-2 Communication networks and systems for power utility automation - Part 7-2: Basic information and communication structure - Abstract communication service interface (ACSI), International Electrotechnical Commission, International Standard, elo. 2010, 213 s. Saatavissa (viitattu 16.5.2018): https://webstore.iec.ch/publication/6015

[13] IEC 61850-7-3 Communication networks and systems for power utility automation - Part 7-3: Basic communication structure - Common data classes, International Standard, jou. 2010, 182 s. Saatavissa (viitattu 16.5.2018): https://webstore.iec.ch/publication/6016

[14] IEC 61850-7-4 Communication networks and systems for power utility automation - Part 7-4: Basic communication structure - Compatible logical node classes and data object classes, International Standard, maa. 2010, 179 s. Saatavissa (viitattu 16.5.2018): https://webstore.iec.ch/publication/6017

[15] IEC 61850-8-1 Communication networks and systems for power utility automation - Part 8-1: Specific communication service mapping (SCSM) - Mappings to MMS (ISO 9506-1 and ISO 9506-2) and to ISO/IEC 8802-3, International Standard, kes. 2011, 386 s. Saatavissa (viitattu 16.5.2018): https://webstore.iec.ch/publication/6021

[16] IEC 61850:2018 SER Series, International Electrotechnical Commission, verkkosivu. Saatavissa (viitattu 9.6.2018): https://webstore.iec.ch/publication/6028

[17] JRuby kotisivu, JRuby verkkosivu. Saatavissa (viitattu 29.9.2018): http://jruby.org/

[18] K. Kaneda, S. Tamura, N. Fujiyama, Y. Arata, H. Ito, IEC61850 based Substation Automation System, teoksessa: 2008 Joint International Conference on Power System Technology and IEEE Power India Conference, Oct, 2008, s. 1–8.

[19] S. Kozlovski, Ruby's GIL in a nutshell, syy. 2017. Saatavissa (viitattu 13.8.2018): https://dev.to/enether/rubys-gil-in-a-nutshell

[20] libIEC61850 API overview, libIEC61850 verkkosivu. Saatavissa (viitattu 3.8.2018): http://libiec61850.com/libiec61850/documentation/

[21] libIEC61850 documentation, libiec61850 verkkosivu. Saatavissa (viitattu 18.8.2018): https://support.mz-automation.de/doc/libiec61850/c/latest/index.html

[22] libIEC61850 kotisivu, libIEC61850 verkkosivu. Saatavissa (viitattu 24.9.2018): http://libiec61850.com

[23] R. E. Mackiewicz, Overview of IEC 61850 and Benefits, teoksessa: 2006 IEEE PES Power Systems Conference and Exposition, Oct, 2006, s. 623–630.

[24] MMS Protocol Stack and API, Xelas Energy verkkosivu. Saatavissa (viitattu 9.7.2018): http://www.xelasenergy.com/products/en_mms.php

[25] New documents by IEC TC 57. Saatavissa (viitattu 9.6.2018): http://digitalsubstation.com/en/2016/12/24/new-documents-by-iec-tc-57/

[26] R. Odaira, J. G. Castanos, H. Tomari, Eliminating Global Interpreter Locks in Ruby Through Hardware Transactional Memory, SIGPLAN Not., vsk. 49, nro 8, hel. 2014, s. 131–142. Saatavissa (viitattu 16.5.2018): http://doi.acm.org/10.1145/2692916.2555247

[27] Official repository for libIEC61850, the open-source library for the IEC 61850 protocols http://libiec61850.com/libiec61850, GitHub verkkosivu. Saatavissa (viitattu 17.5.2018): https://github.com/mz-automation/libiec61850

[28] Parsing Program Options with Argp, The GNU C Library. Saatavissa (viitattu 1.9.2018): https://www.gnu.org/software/libc/manual/html_node/Argp.html

[29] A. Patrizio, XML is toast, long live JSON, kes. 2016. Saatavissa (viitattu 18.8.2018): https://www.cio.com/article/3082084/web-development/xml-is-toast-long-live-json.html

[30] RabbitMQ C client, Github verkkosivu. Saatavissa (viitattu 1.9.2018): https://github.com/alanxz/rabbitmq-c

[31] RabbitMQ Compatibility and Conformance, RabbitMQ verkkosivu. Saatavissa (viitattu 11.7.2018): https://www.rabbitmq.com/specification.html

[32] RabbitMQ kotisivu, RabbitMQ verkkosivu. Saatavissa (viitattu 23.9.2018): https://www.rabbitmq.com/

[33] RabbitMQ Tutorial Routing, RabbitMQ verkkosivu. Saatavissa (viitattu 31.7.2018): https://www.rabbitmq.com/tutorials/tutorial-four-python.html

[34] RabbitMQ Tutorial Topics, RabbitMQ verkkosivu. Saatavissa (viitattu 31.7.2018): https://www.rabbitmq.com/tutorials/tutorial-five-python.html

[35] Ruby FFI, GitHub verkkosivu. Saatavissa (viitattu 24.9.2018): https://github.com/ffi/ffi

[36] K. Schwarz, Introduction to the Manufacturing Message Specification (MMS, ISO/IEC 9506), NettedAutomation verkkosivu, 2000. Saatavissa (viitattu 9.7.2018): https://www.nettedautomation.com/standardization/ISO/TC184/SC5/WG2/mms_intro/index.html

[37] J. Storimer, Nobody understands the GIL, kes. 2013. Saatavissa (viitattu 16.5.2018): https://www.jstorimer.com/blogs/workingwithcode/8085491-nobody-understands-the-gil

[38] TwoBitHistory, The Rise and Rise of JSON, syy. 2017. Saatavissa (viitattu 29.9.2018): https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html

[39] J. Wyse, Why JSON is better than XML, elo. 2014. Saatavissa (viitattu 29.9.2018): https://blog.cloud-elements.com/json-better-xml

[40] P. Youssef, Multi-threading in JRuby, hel. 2013. Saatavissa (viitattu 18.8.2018): http://www.restlessprogrammer.com/2013/02/multi-threading-in-jruby.html

LIITE A: VIESTISTÄ PROSESSOITU JSON-RAKENNE


```
{
  "dataSetName": "LD0_CTRL/LLN0$StatUrg",
  "sequenceNumber": 0,
  "confRevision": null,
  "timestamp": 1534993167923,
  "bufferOverflow": false,
  "values": [
    {
      "reasonForInclusion": "GI",
      "mmsReference": "LD0_CTRL/CBCILO1$ST$EnaCls",
      "reference": "LD0_CTRL/CBCILO1.EnaCls",
      "functionalConstraint": "ST",
      "values": [
        {
          "reference": "LD0_CTRL/CBCILO1.EnaCls.stVal",
          "type": "boolean",
          "value": false
        },
        {
          "reference": "LD0_CTRL/CBCILO1.EnaCls.q",
          "type": "bit-string",
          "size": 13,
          "valueLittleEndian": 0,
          "valueBigEndian": 0
        },
        {
          "reference": "LD0_CTRL/CBCILO1.EnaCls.t",
          "type": "utc-time",
          "value": 1534845456
        }
      ]
    },
    {
      "reasonForInclusion": "GI",
      "mmsReference": "LD0_CTRL/CBCSWI1$ST$Loc",
      "reference": "LD0_CTRL/CBCSWI1.Loc",
      "functionalConstraint": "ST",
```

```
38          "values": [
39            {
40              "reference": "LD0_CTRL/CBCSWI1.Loc.stVal",
41              "type": "boolean",
42              "value": true
43            },
44            {
45              "reference": "LD0_CTRL/CBCSWI1.Loc.q",
46              "type": "bit-string",
47              "size": 13,
48              "valueLittleEndian": 0,
49              "valueBigEndian": 0
50            },
51            {
52              "reference": "LD0_CTRL/CBCSWI1.Loc.t",
53              "type": "utc-time",
54              "value": 1534845456
55            }
56          ]
57        },
58        {
59          "reasonForInclusion": "GI",
60          "mmsReference": "LD0_CTRL/CBCSWI1$ST$Pos",
61          "reference": "LD0_CTRL/CBCSWI1.Pos",
62          "functionalConstraint": "ST",
63          "values": [
64            {
65              "reference": "LD0_CTRL/CBCSWI1.Pos.stVal",
66              "type": "bit-string",
67              "size": 2,
68              "valueLittleEndian": 0,
69              "valueBigEndian": 0
70            },
71            {
72              "reference": "LD0_CTRL/CBCSWI1.Pos.q",
73              "type": "bit-string",
74              "size": 13,
75              "valueLittleEndian": 2,
76              "valueBigEndian": 2048
77            },
78            {
79              "reference": "LD0_CTRL/CBCSWI1.Pos.t",
```

```
80              "type": "utc-time",
81              "value": 1534845480
82            },
83            {
84              "reference": "LD0_CTRL/CBCSWI1.Pos.stSeld",
85              "type": "boolean",
86              "value": false
87            }
88          ]
89        }
90    ]
91  }
```

Ohjelma 1. *Viestin prosessoitu JSON-rakenne.*

LIITE B: C-OHJELMAN TULOSTAMA APUTEKSTI

```
Usage: rcb_sub [OPTION...]
        EXCHANGE ROUTING_KEY
        RCB_REF [RCB_OPTIONS...]
        [RCB_REF [RCB_OPTIONS...]...]
Configure and subscribe IED report control blocks.
Received reports are combined with variable specification
data and formatted to JSON. Formatted messages are
forwarded to AMQP broker using direct exchange.

 OPTION options:
  -a, --amqp-host=HOST       Host address of the AMQP
                             broker, defaults to localhost
  -e, --ied-port=PORT        Port for MMS communication,
                             defaults to 102
  -h, --ampq-vh=VH           Virtual host for the AMQP
                             broker, defaults to '/'
  -i, --ied-host=HOST        Host address of the IED,
                             defaults to localhost
  -m, --ampq-port=PORT       Port for AMQP communication,
                             defaults to 5672
  -p, --ampq-pwd=PWD         User password for the AMQP
                             broker, defaults to 'quest'
  -u, --ampq-user=USER       User for AMQP broker,
                             defaults to 'quest'
  -v, --verbose              Explain what is being done

 RCB_OPTIONS options:
  -g, --gi=VALUE             Set general interrogation
                             bit (1/0)
  -o, --opt-fields=MASK      Report optional fields int
                             bit mask (0 <= MASK <= 255)
  -t, --trigger=MASK         Report triggering int bit
                             mask (0 <= MASK <= 31)

   -?, --help                Give this help list
       --usage               Give a short usage message
  -V, --version              Print program version
```

```
38
39 Mandatory or optional arguments to long options are also
40 mandatory or optional for any corresponding short options.
41
42 EXCHANGE is name of the exchange used with the AMQP
43 broker. ROUTING_KEY is routing key used for the
44 published AMPQ broker messages. RCB_REF is reference
45 to report control block as specified in IEC 61850 standard.
46 For example MY_LD0/LLN0.BR.rcbMeas01
47
48 Reason for inclusion optional field is set automatically
49 in order for the program to combine read specification
50 data and only to include needed data set values which
51 actually triggered the report.
52
53 Trigger MASK values:
54   1  : data changed
55   2  : quality changed
56   4  : data update
57   8  : integrity
58   16 : general interrogation
59
60 Optional field MASK values:
61   1   : sequence number
62   2   : timestamp
63   4   : reason for inclusion (automatically set, see above)
64   8   : data set
65   16  : data reference
66   32  : buffer overflow
67   64  : entry id
68   128 : configure revision
69
70 Example usage:
71 $ rcb_sub -v -i192.168.2.220 testexchange testkey \
72 MY_LD0/LLN0.BR.rcbMeas01 -g1 -t27 -o16
73
74 Report bugs to mauri.mustonen@alsus.fi.
```

Ohjelma 2. *rcb_sub-ohjelman aputeksti.*