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, 55 sivua, 5 liitesivua
Toukokuu 2018
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Kari Systä

Avainsanat: IEC 61850, MMS, AMQP, IED, sähköasema, älykäs elektroniikkalaite, ohjelmistokehitys, hajautettu järjestelmä, kommunikointiparadigmat

Sähkönjakeluverkot ovat tärkeä osa nykyistä yhteiskuntaa ja sen päivittäistä toimintaa. Sähköverkko koostuu sähköntuotantolaitoksista, sähkölinjoista ja sähköasemista. Sähköverkon eri komponenttien avulla sähkö toimitetaan tuotantolaitoksista kuluttajille. Sähköasemat ja niiden automatisointi ovat tärkeässä osassa verkon yleisen toiminnan ja turvallisuuden takaamiseksi. Tässä diplomityössä keskitytään suunnittelemaan ja toteuttamaan hajautetun järjestelmän arkkitehtuuria hyödyntävä ohjelmistokomponentti. Komponentin on tarkoitus olla osa isompaa sähköasemiin liittyvää järjestelmää. Se tilaa tietoa sähköasemalta verkon yli ja jakaa sen järjestelmän muiden komponenttien kanssa. Sähköasemalta tuleva tieto sisältää esimerkiksi mittaustietoa, jota näytetään käyttöliittymässä.

Sähköasemilta tieto tilataan älykkäiltä elektroniikkalaitteilta (Intelligent Electronic Device, IED). IED:t ovat sähköaseman verkkoon kytkettyjä automaatiolaitteita, joista käytetään myös nimitystä suojarele. IED-laitteiden kommunikointiin liittyy maailmanlaajuinen IEC 61850 -standardi (International Electrotechnical Commission). Standardi määrittää kuinka IED-laitteet ja niihin yhteydessä olevat ohjelmat kommunikoivat verkon yli.

Ohjelmasta oli toteutettu demo ennen työn aloitusta, joka todisti osien toimivuuden. Demototeutuksessa oli ongelmia, jotka haittasivat sen jatkokehitystä. Tässä työssä demoa käytetään pohjana uuden version suunnittelulle. Demosta analysoidaan sen ongelmia ja mistä ne johtuivat. Näitä tietoja käytetään uuden komponentin tekniikan suunnitteluun.

Tuloksena on hajautetun järjestelmän arkkitehtuuria hyödyntävä ohjelmistokomponentti, joka kykenee tilaamaan viestejä IED-laitteelta IEC 61850 -standardin mukaisesti. Komponentti kykenee prosessoimaan ja jakamaan tilatut viestit järjestelmän muiden komponenttien kanssa.

ABSTRACT

MAURI MUSTONEN: Substation's intelligent electronic device messages subscription and processing
Tampere University of Technology
Master of Science thesis, 55 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, distributed system, communication paradigms

Electric grids are an important part of society today. It consists of power plants, power lines and substations. These components allow delivering electricity 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 that utilizes distributed system architecture. The component is intended to be a part of a larger system related to the substations. It subscribes information from the substation and shares it with other parts of the system. The information from the substation can include different types of data, such as measurement data which is shown on the user interface.

The information is subscribed from an Intelligent Electronic Device (IED) in substations. An IED is an automation device which controls the other physical devices of the substation. IEDs are connected to the substation's local network. IED can also be referred to as "protection relay". The International Electrotechnical Commission has defined a worldwide standard called IEC 61850 that defines the rules for how IED-devices and software outside of the substation need to communicate with each other over the network.

A proof of concept software component had already been developed before the start of this thesis. However, this component had many problems which did not encourage its further development. Analyzing these problems is a part of this thesis. The new information is then used to desing the new software.

The result of this thesis is a software component that utilizes a distributed system architecture. The component can subscribe to information from the IED according to the IEC 61850 standard and share it with the other parts of the system.

ALKUSANAT

Toteutin tämän diplomityöni yritykselle nimeltä Alsus Oy, jossa työskentelin vuonna 2018. Diplomintyön aihe liittyi sopivasti sen hetkisiin työtehtäviini ja sisälsi todella paljon omia 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.

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 hyvästä ja tarmokkaasta 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 saamastani tuesta ja motivaatiosta. Lopuksi kiitän muita tärkeitä ystäviäni, jotka auttoivat minua diplomityössäni oikolukemalla ja motivoimalla minua.

Diplomityöni on kirjoitettu Latex:illa ja kaikki lähdekoodi on saatavissa GitHub:issa osoitteessa: https://github.com/kazooiebombchu/tut_master_thesis.

Tampereella, 21.11.2018

Mauri Mustonen

SISÄLLYSLUETTELO

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

AMQP	*Advanced Message Queuing Protocol* on avoin standardi viestien välitykseen eri osapuolien kesken
DA	*Data Attribute* on IEC 61850 -standardissa käsite abstrahoimaan jokin sähköaseman laitteen mitattava arvo (esim. jännite)
DO	*Data Object* on IEC 61850 -standardissa käsite abstrahoimaan joukko samaan kuuluvia muuttujia
DSM	*Distributed Shared Memory* on jaettu muisti, joka käyttäjälle näyttää kuin paikallinen fyysinen muisti
FFI	*Foreign Function Interface*, mekanismi, jolla ohjelma voi kutsua toisella kielellä toteutettuja funktiota
GCS	*Group Communication System* tarkoittaa systeemiä, jossa kommunikoidaan joukolle osapuolia
GIL	*Global Interpreter Lock*, Ruby-kielen tulkissa oleva globaali tulkkilukitus, joka rajoittaa yhden säikeen suoritukseen kerrallaan
GVL	*Global Virtual Machine Lock* on sama kuin GIL, mutta eri nimellä
HAL	*Hardware Abstraction Layer* on laitteistoabstraktiotaso abstrahoimaan laitteen toiminnallisuuden lähdekoodista
IEC	*International Electrotechnical Commission*, on sähköalan kansainvälinen standardiorganisaatio
IEC 61850	maailmanlaajuinen sähköasemien IED-laitteiden kommunikoinnin määrittävä standardi
IED	*Intelligent Electronic Device*, sähköaseman älykäs elektroniikkalaite (myös nimellä turvarele), joka toteuttaa aseman automaatiota
IoT	*Internet of Things*, on verkko joka koostu siihen kytketyistä erilaisista laitteista
IP	*Internet Protocol* on protokolla verkkoliikenteessä joka huolehtii pakettien perille toimittamisesta
JRuby	on Ruby-kielen tulkki Ruby-koodin suoritukseen Java-virtuaalikoneella
JSON	*JavaScript Object Notation* on JavaScript-kielessä käytetty notaatio objektista ja sen sisällöstä
JVM	*Java Virtual Machine* on Java-kielen virtuaalikone Java-koodin suoritukseen
LD	*Logical Device* on IEC 61850 -standardissa käsite abstrahoimaan joukko fyysisestä laitteesta joukko loogisesti yhteen kuuluvia laitteita
LN	*Logical Node* on IEC 61850 -standardissa käsite abstrahoimaan fyysinen laite loogisen laitteen ryhmästä
MMS	*Manufacturing Message Specification* on maailmanlaajuinen standardi reaaliaikaiseen kommunikointiin verkon yli eri laitteiden välillä
MQTT	*Message Queuing Telemetry Transport* on julkaisija-tilaaja-pohjainen avoin standardi kommunikointiin hajautetussa järjestelmässä
MRI	*Matz's Ruby Interpreter* on Ruby-kielen tulkki
MV	*Measured Value* on IEC 61850 -standardissa on dataobjektin luokkatyyppi
RCB	*Report Control Block*, viestien konfigurointiin ja tilaukseen tarkoitettu luokkatyyppi IED-laitteelle
RMI	*Remote Method Invocation* on oliopohjainen metodikutsu jossa metodi sijaitsee toisella koneella

RoR	*Ruby on Rails* on kehys web-sovellusten kehittämiseen Ruby-kielellä
RPC	*Remote Procedure Call* on etäproseduurikutsu jossa proseduuri sijaitsee toisella koneella
TCP/IP	*Transmission Control Protocol/Internet Protocol*, on joukko standardeja verkkoliikenteen määrityksiin
UDP	*User Datagram Protocol* on pakettien lähettämisen protokolla Internet protokollan päällä
XML	*Extensible Markup Language* on laajennettava merkintäkieli, joka on ihmis- ja koneluettava
YARV	*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. Nykypäivänä asemien toiminnallisuutta voidaan seurata ja ohjata etäohjauksella verkon yli. Tätä kautta etäohjelman on mahdollista tilata tietoa aseman toiminnasta ja sen tilasta. Sähköaseman yksi tärkeä tehtävä on suojata ja tarkkailla verkon toimivuutta, ja esimerkiksi vikatilanteessa katkaista linjasta virrat pois. Tällainen vikatilanne on esimerkiksi kaapelin poikki meneminen, joka aiheuttaa vaarallisen oikosulkutilanteen.

Tässä diplomityössä on tarkoituksena suunnitella hajautetun järjestelmän arkkitehtuuri ja toteuttaa ohjelmisto osaksi isompaa sähköasemiin liittyvää järjestelmää. Tavoitteena on tilata viestejä verkon yli sähköaseman automaatiolaitteelta ja jakaa saadut viestit järjestelmän muiden osien kanssa. Työssä käsitellään hajautetun järjestelmän paradigmoja ja analysoidaan mitkä niistä sopisivat tilanteeseen parhaiten. Analyysin tuloksien ja ohjelmistolle asetettujen vaatimusten perusteella päädytään kokonaisuuden suunnitelmaan. Suunnitelma toteutetaan ohjelmistoksi, joka toimii osana olemassa olevaa isompaa järjestelmää. Toteutus jakaa sähköaseman viestin järjestelmän muille osille, joita ovat esimerkiksi mittaustiedon näyttäminen ja aseman tilan tarkkailu käyttöliittymäkomponenteissa.

Viestit sähköasemilta tilataan sen verkkoon kytketyiltä automaatiolaitteilta. Automaatiolaitteet ohjaavat aseman muuta laitteistoa kuten muuntajia ja katkaisijoita. Nämä automaatiolaitteet noudattavat verkon yli kommunikoinnissa maailmanlaajuisesti määritettyä *IEC 61850 -standardia* (*International Electrotechnical Commission*). Standardin tarkoituksena on määrittää yhteinen kommunikointiprotokolla ja säännöt aseman kaikkien eri laitteiden välille. Standardi määrittää myös asemalta viestien tilaamisen mekanismit, joita aseman ulkopuolisen ohjelman täytyy noudattaa. Nämä määritykset ovat tämän työn kannalta tärkein osa standardia ja vaikuttavat hajautetun järjestelmän suunnitteluun.

Diplomityön tekijä oli jo ennen tämän työn aloitusta Alsus Oy:ssä toteuttanut yksinkertaisen demo-ohjelmiston (proof of concept). Ohjelmisto kykeni tilaamaan viestejä asemalta standardin mukaisesti ja tallentamaan ne tietokantaan. Toteutus oli puutteellinen ja siinä oli toimintaan liittyviä ongelmia, jotka haittasivat sen jatkokehitystä tuotantoon asti. Demon tarkoituksena oli opettaa tekijälle standardia ja sen mekanismeja ennen oikeaa toteutusta. Tässä työssä analysoidaan demon toimintaa ja sen ongelmia. Nämä tulokset yhdistetään aikaisemmin mainitun hajautetun järjestelmän suunnittelun kanssa. Lopputuloksena saadaan toimiva kokonaisuuden suunnitelma uudelle toteutukselle.

Tämän diplomityön rakenne alkaa pohjatietojen käsittelyllä. Ensin käydään läpi järjes-

telmän hajautuksen teoriaa ja siihen liittyviä kommunikointiparadigmoja. Tämän jälkeen käsitellään IEC 61850 -standardia ja sen toimintaa. Edellä mainittujen tietojen avulla analysoidaan erilaisia hajautuksen vaihtoehtoja ja mitkä niistä sopisivat toteutukseen parhaiten. Tuloksena tästä on hajautetun järjestelmän arkkitehtuurisuunnitelma, jota tekninen suunnittelu tulee tarkentamaan. Ennen teknistä suunnittelua työssä analysoidaan demon toimintaa ja sen ongelmia. Kaikkien edellä mainittujen perusteella suunnitellaan ohjelmistoarkkitehtuuri ja valitaan sen käyttämät tekniikat. Suunnitelman ja päätöksien jälkeen työssä käydään läpi itse ohjelman toteutus. Työn lopussa arvioidaan ja pohditaan saatuja tuloksia asetettuihin tutkimuskysymyksiin ja miten tavoitteisiin päästiin. Lisäksi käsitellään myös toteutuksen tulevaisuutta ja mahdollisia vaihtoehtoisia toteutustapoja.

1.1 Vaatimukset

Diplomityön suunnittelulle ja toteutukselle asetettiin vaatimuksia, jotka ohjelmiston pitää pystyä täyttämään. Vaatimuksien tehtävä on asettaa työlle selvät tavoitteet mitä järjestelmään halutaan tuoda lisää. Diplomityön tehtävä on tiedon analyysin ja tutkimustyön kautta löytää sopivat menetelmät suunnitteluun ja sen toteutukseen. Asetettuja vaatimuksia käytetään pohjana työssä tehdyille päätöksille. Pohdintaosiossa vaatimuksilla arvioidaan työn tavoitteisiin pääsyä ja eri ratkaisuvaihtoehtoja. Kaikki vaatimukset oli asetettu jo ennen työn aloittamista tai sen alkuvaiheessa. Jokaiselle vaatimukselle on asetettu lyhenne, jolla siihen viitataan muualla tekstissä.

Ohjelmistolle asetettiin mm. seuraavia vaatimuksia:

- V1: viestien tilaus sähköasemalta IEC 61850 -standardin mukaisesti,
- V2: tilattujen viestien jakaminen järjestelmässä siitä kiinnostuvien komponenttien kanssa,
- V3: tilattuja viestejä haluavien komponenttien määrä pitää pystyä muuttumaan järjestelmän tarpeiden mukaan,
- V4: muu järjestelmä ohjaa milloin viestien tilaus sähköasemilta aloitetaan ja lopetetaan,
- V5: komponenttien täytyy saada ilmoitus uudesta viestistä ilman erillistä kyselyä,
- V6: viestit puskuroidaan myöhempää käsittelyä varten jos komponentti ei ehdi niitä heti käsitellä,
- V7: komponentin pitää pystyä suodattamaan viestit sen lähteen identiteetin perusteella,
- V8: viestien jakamisen muoto pitää olla helposti ymmärrettävä järjestelmän osapuolien kesken,
- V9: sähköasemalta tilattavien viestien määrä pitää pystyä muuttumaan tilauksien välillä,

- V10: viestien välitystekniikka järjestelmässä täytyy tukea verkkopalvelun tapauksessa TCP/IP-proto kollamäärityksiä, ja
- V11: tiedonsiirrossa lähetystakuu ei ole välttämättömyys.

Osa vaatimuksista tulevat muun järjestelmän toimintaperiaatteista ja siitä kuinka sitä käytetään. Käyttäjän on tarkoitus pystyä ohjaamaan viestien tilausta sähköasemille käyttöliittymän kautta. Tästä seurauksena on vaatimus, jossa muun järjestelmän täytyy ohjata viestin hakuun liittyviä osia. Uusien järjestelmän komponenttien kehitys halutaan pitää helppona. Tämän takia viestin saamisen rajapinnat tulee suunnitella komponentissa helposti käytettäväksi. Rajapinnan tulee tarjota viestit helposti ymmärrettävässä muodossa. Lisäksi ilmoitukset uuden viestin saapuessa ja niiden puskurointi, jos niitä ei ehditä heti käsitellä. Komponentille pitää tarjota mahdollisuus erottaa tai saada viestit sähköaseman ja sen automaatiolaitteen mukaan. Tämä sen takia, että järjestelmä käsittelee paljon erilaisia sähköasemia ja niiden automaatiolaitteita. Komponenttien pitää pystyä selvittämään viestien alkuperä. Tästä esimerkkinä on tietyn automaatiolaitteen mittaustiedon näyttäminen. Tiedonsiirtoon ei alustavasti tarvittu lähetystakuita tiedon tyypin takia (mittausdata). Uusi viesti korvaa edellisen lähetyksen tiedot. Olisi tavoiteltavaa, että toteutus kuitenkin mahdollistaisi tämän tulevaisuuden varalta. Muu järjestelmä ja sen osat ovat toteutettu web-sovelluksena. Tästä tulee vaatimus, että viesti täytyy onnistua kuljettamaan osapuolten välillä TCP/IP-protokollaperheen avulla.

1.2 Tutkimuskysymykset

Ohjelmiston vaatimuksien lisäksi diplomityölle asetettiin selviä tutkimuskysymyksiä, joihin työn aikana yritetään etsiä vastausta. Tutkimuskysymykset liittyvät työhön korkealla tasolla ja käsittelevät sen kokonaisuudesta eri kohtia. Tutkimuskysymyksiä käytetään myös työn lopussa tuloksien pohdinnassa.

Työlle asetettiin seuraavat tutkimuskysymykset:

- *Mitkä eri hajautetun järjestelmän kommunikointiparadigmoista sopivat työn vaatimuksien asettaman ongelman ratkaisuun ja mitkä eivät?*
- *Minkälainen on hajautetun järjestelmän ohjelmistoarkkitehtuuri, joka täyttää asetetut vaatimukset?*
- *Järjestelmän hajautuksessa, mikä olisi sopiva tiedon jakamisen muoto eri osapuolten välillä?*
- *Mitkä olivat syyt demoversion ongelmiin ja kuinka nämä estetään uudessa versiossa?*

2. HAJAUTETTU JÄRJESTELMÄ

Työn tarvittavan tiedon käsittely aloitetaan ensin hajautetun järjestelmän teorialla ja käymällä läpi siihen liittyviä kommunikointiparadigmoja. Tässä luvusta käsiteltyjä paradigmoja tullaan myöhemmin vertailemaan arkkitehtuurin suunnittelussa ja päätöksiin tullaan myös apuna käyttämään kommunikointiin liittyvää teoriaa.

2.1 Mikä on hajautettu järjestelmä?

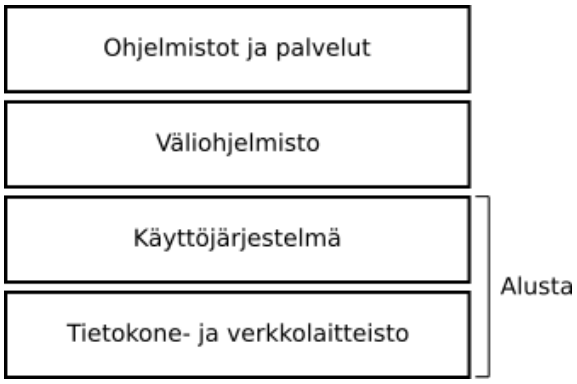
Hajautettu järjestelmä (*distributed system*) on verkko toisiinsa kytkettyjä fyysisiä- tai ohjelmistopohjaisia komponentteja, jotka kommunikoivat toistensa kanssa viestien välityksellä. Hajautetussa järjestelmässä osapuolten etäisyydellä ei ole merkitystä. Hajautetut järjestelmät ovat oma tieteenala, joka lähti liikkeelle käyttöjärjestelmien arkkitehtuurien tutkimuksista 1960-luvulla [3, s. 384]. Ensimmäinen laajalle levinnyt hajautettu järjestelmä oli lähiverkot (Local Area Network tai LAN), joka keksittiin 1970-luvulla [3, s. 32]. Hajautetun järjestelmän määrityksistä ja toteuttamisesta on nykypäivänä olemassa hyvin kirjallisuutta ja tietoa, esimerkiksi [22, 46, 45, 5].

Järjestelmän hajautuksessa ja sen käytössä on pääasiassa kyse resurssien jakamisesta osapuolien kesken. Resurssi on abstrakti käsite ja voi tarkoittaa tekniikasta tai toteutuksesta riippuen montaa eri asiaa. Resurssilla voidaan esimerkiksi kuvata jaettua fyysistä laitetta kuten levyä tai tulostinta, ohjelmiston tapauksessa oliota tai tietokantaa [22, s. 2–3]. Nykyinen internet mahdollistaa monien eri laitteiden kytkemisen verkkoon ja niiden kommunikoinnin keskenään. Ja on nykypäivänä hyvä esimerkki todella laajasta hajautetusta järjestelmästä.

Hajautetussa järjestelmässä voidaan puhua osapuolten *heterogeenisyydestä* (*heterogeneity*), eli osapuolet voivat kommunikoida toistensa kanssa tekniikasta tai toteutuksesta riippumatta. Heterogeenisyys voidaan toteuttaa kerrosarkkitehtuurin avulla. Korkean ja matalan tason ohjelmistojen välissä on ns. *väliohjelmistokerros* (*middleware*). Tämän kerroksen tehtävä on abstrahoida matalan tason ohjelmisto tai alusta ja tehdä siitä heterogeeninen ylemmän tason ohjelmistoille ja palveluille. Kuvassa 1 on esitetty edellä mainittu kerrosarkkitehtuuri. [22, s. 16–17] [46, s. 2–3]

2.1.1 Osapuolien kommunikointi

Hajautetussa järjestelmässä osapuolet kommunikoivat keskenään viestien välityksellä. Jotta viestit voitaisiin vaihtaa tekniikasta riippumattomasti osapuolten välillä, tulee tieto esittää alustariippumattomassa muodossa. Kommunikoivien osapuolten täytyy sopia



Kuva 1. *Väliohjelmistokerros abstrahoimaan alusta heterogeeniseksi ylemmän tason ohjelmistolle (pohjautuu kuvaan [22, s. 52]).*

yhteisestä viestin formaatista. Ohjelmat yleensä käsittelevät tietoa ohjelmointikielen tarjoamilla tietorakenteilla kuten esimerkiksi listoilla, puurakenteilla ja luokilla. Jotta tieto saadaan lähetettyä viestinä, täytyy se ensin muuntaa lähetykseen sopivaan muotoon. Tämä prosessi tunnetaan englannin kielellä nimellä *marshalling*. Jotta vastaanotettua tietoa voidaan käyttää, täytyy viesti purkaa takaisin ohjelmointikielen rakenteiksi. Tämä prosessi englannin kielessä tunnetaan nimellä *unmarshalling*. [22, s. 158]

Kommunikointiin voidaan osapuolten välillä sopia takuita lähetyksistä ja tiedon oikeudesta. Lähettäjä voi esimerkiksi vaatia vastaanottajaa kuittaamaan viestin vastaanottamisen. Jos kuittausta ei tule tietyn ajan sisällä, lähettäjä uudelleenlähettää viestin vastaanottajalle. Esimerkki tästä on TCP-protokollan (Transmission Control Protocol) paketit, jossa lähettäjä odottaa vastaanottajan kuittausta paketista [54, s. 9–10]. On myös tilanteita, jossa lähettäjä ei välitä saako vastaanottaja viestin. Tässä tapauksessa osapuoli lähettää viestejä ilman tietoa siitä ottaako niitä kukaan vastaan. Esimerkki tästä on UDP-protokolla (User Datagram Protocol, jossa lähettäjä ei odota kuittausta paketteihin [53]. Uuden onnistuneesti lähetetyn paketin odotetaan korvaavan edellisen tieto. Minkälainen takuu viestien lähetykseen valitaan, riippuu ohjelmiston vaatimuksista.

2.1.2 Kommunikoinnin luokittelu

Hajautetuissa järjestelmissä osapuolten kommunikoinnin välillä voi olla *suora-* tai *epäsuora liitos*. Suora liitos tarkoittaa tilannetta, jossa osapuolet tietävät toistensa identiteetin. Esimerkiksi lähettäjä lähettää viestin suoraan vastaanottajalle. Epäsuora liitos tarkoittaa tilannetta, jossa osapuolet eivät tiedä toistensa identiteettiä. Epäsuorassa liitoksessa osapuolet yleensä kommunikoivat välittäjän kautta, joka hoitaa viestien lähettämisen toiselle osapuolelle. Suorassa liitoksessa toisen osapuolen vaihtaminen on vaikeampaa kuin epäsuorassa liitoksessa. Esimerkkinä tästä on yksinkertainen asiakas-palvelin-malli. Suoran liitoksen takia palvelin on vaikeampi vaihtaa toiseen samanlaiseen (palvelin tulkkaa HTML-sivuja asiakkaalle). Epäsuorassa liitoksessa palvelimen vaihto samanlaiseen on helpompaa, jos niiden välissä on tarpeeksi epäsuoruutta (geneerinen rajapinta tai välittäjä). [22, s. 230]

Hajautetuissa järjestelmissä voidaan puhua erilaisista heikkojen liitoksien luokittelumalleista. Kirjallisuudessa näitä kutsutaan

- *heikko väliliitos* (*space uncoupling*), tarkoittaa liitosta, jossa osapuolet eivät tiedä toistensa identiteettiä, ja
- *heikko aikaliitos* (*time uncoupling*), tarkoittaa liitosta, jossa osapuolien ei tarvitse olla olemassa samaan aikaan.

Näiden mukaan voidaan esittää malli, jolla luokitellaan osapuolten välistä liitosta. Tämä malli on esitetty taulukossa 1. [22, s. 230] [17, s. 116]

Taulukko 1. *Hajautetussa järjestelmässä osapuolten kommunikoinnin luokittelun malli (pohjautuu taulukoihin [22, s. 231] [13, s. 84]).*

	Vahva aikaliitos	**Heikko aikaliitos**
Vahva väliliitos	Kommunikointi tarkoitettu suoraan toiselle osapuolelle. Osapuolten täytyy olla olemassa samaan aikaan, esimerkiksi suora viestintä tai etäfunktiokutsu.	Kommunikointi tarkoitettu suoraan toiselle osapuolelle. Osapuolet voivat olla olemassa eri aikaan.
Heikko väliliitos	Osapuolten ei tarvitse tietää toistensa identiteettiä. Osapuolten täytyy olla olemassa samaan aikaan, esimerkiksi IP-ryhmälähetys.	Osapuolten ei tarvitse tietää toistensa identiteettiä. Osapuolet voivat olla olemassa eri aikaan, esimerkiksi julkaisija-tilaaja ja viestijono

Taulukossa 1 vasemmassa yläkulman luokittelu tarkoittaa kommunikointia, missä osapuolet tietävät toistensa identiteetin ja niiden täytyy olla olemassa samaan aikaan. Tämä on siis kaikista vahvin liitos mitä osapuolten välillä voi olla ja jossa on vähiten epäsuoruutta. Esimerkkinä tästä kommunikoinnista on etäfunktiokutsu tai suora viestintä prosessien välillä. Oikeassa yläkulmassa on tilanne, jossa osapuolet edelleen tietävät toistensa identiteetin, mutta niiden ei tarvitse olla olemassa samaan aikaan. Esimerkkinä osapuoli lähettää viestin tietylle identiteetille, mutta vastaanottaja ottaa viestin vastaan vasta myöhemmin. Esimerkkinä tästä on telekommunikaatiossa käytetty tallenna-ja-välitä-tekniikka (store-and-forward) [11]. Taulukon vasemmassa alakulmassa on tilanne, jossa osapuolet eivät tiedä toistensa identiteettiä, mutta niiden täytyy olla olemassa samaan aikaan. Esimerkkinä tästä on IP-ryhmälähetys, jossa viesti lähetetään kaikille verkon osapuolille ilman tietoa niiden identiteetistä. Vastaanottaja on olemassa samaan aikaan ja vastaa lähettäjälle ilman identiteettiä. Taulukon oikeassa alakulmassa on tilanne missä osapuolten välillä on kaikista eniten epäsuoruutta muista luokitteluista. Osapuolet eivät tiedä toistensa identiteettiä ja niiden ei tarvitse olla olemassa samaan aikaan. Esimerkkinä julkaisija-tilaaja- ja viestijonoparadigmat. [22, s. 230–232] [13]

Epäsuoruus ohjelmoinnissa tuo hyötyjä ja joustavuutta ohjelmaan ja sen toimintaan kuten aikaisemmin tuotiin esille. Epäsuoruuden mukana tulee myös haittoja, kuten kommunikointi voi olla vaikeampaa osapuolten välillä ja epäsuoruus yleensä heikentää ohjelman

suorituskykyä. Suorituskyky ja kommunikoinnin vaikeudet vaihtelevat tekniikasta ja toteutuksesta riippuen.

2.2 Hajautuksen paradigmoja

Hajautetussa järjestelmässä kommunikoinnin ja sen toteuttamisen mahdollisuudet ovat laajat ja vaihtoehtoja on paljon. Siksipä on tärkeää ymmärtää mitä ovat järjestelmän kommunikoivat osapuolet ja miten ne kommunikoivat. Näiden ymmärtäminen auttaa kehittäjää paremmin hahmottamaan vaihtoehtoja ja kokonaisuuden toimintaa.

Hajautetuissa järjestelmissä kommunikoivien osapuolien voidaan ajatella olevan *olioita*, *komponentteja* tai *web-palveluita* (*World Wide Web*). Oliot tulevat suoraan olio-ohjelmoinnista ja on tarkoitettu jäsentämään asiaa tai toiminnallisuutta pienempiin omiin kokonaisuuksiinsa, joilla on oma sovittu rajapinta. Rajapinnan tarkoituksena on kapseloida toiminnallisuutta helppokäyttöiseksi rajapinnaksi sen käyttäjälle. Komponentti on kokoelma olioita, jotka muodostavat loogisen käytettävän kokonaisuuden. Komponentit myös kommunikoivat rajapinnan läpi samoin kuin oliot. Ero näiden välillä on, että komponentti oman rajapintansa lisäksi olettaa tiettyjä rajapintamäärityksiä muilta komponenteilta. Komponentti käyttää muiden komponenttien rajapintoja toteuttaakseen sille määritettyä toiminnallisuutta. Web-palvelut noudattavat samaa periaatetta rajapintojen kanssa kuin olio ja komponentti, mutta toimivat web-teknologioiden päällä. Olioita ja komponentteja käytetään toteuttamaan tiukemmin sidottuja ohjelmistoja. Web-palveluita käytetään toteuttamaan omia palvelukokonaisuuksia, joita voidaan liittää yhteen toteuttamaan kokonaisia ohjelmistoja. [22, s. 42–43]

Hajautetussa järjestelmässä kommunikoinnin paradigmat voidaan jakaa kolmeen kategoriaan, jotka on esitetty taulukossa 2. Taulukossa tyyppien alla on esitetty sille tyypillisiä kommunikoinnin paradigmoja. Näitä paradigmoja käsitellään tarkemmin kappaleissa 2.2.1, 2.2.2 ja 2.2.3.

***Taulukko 2**. Hajautetun järjestelmän kommunikointiparadigmat kolmella päätasolla (pohjautuu tauluun [22, s. 46]).*

Prosessien välinen kommunikaatio	**Etäkutsu**	**Epäsuora kommunikaatio**
Viestien välitys (message passing)	Pyyntö-vastaus (request-reply)	Joukkokommunikointi (group communication)
Pistoke (socket)	Etäproseduurikutsu (RPC)	Julkaisija-tilaaja (publish-subscribe)
Ryhmäkutsu (multicast)	Etämetodikutsu (RMI)	Viestijono (message queue)
		Jonotaulu (tuple space)
		Hajautetusti jaettu muisti (DSM)

Prosessien välinen kommunikointi (*interprocess communication*) on matalan tason kommunikointia osapuolten välillä yleensä suoralla yhteydellä ohjelmointirajapintaan. *Etäkutsussa* (*remote invocation*) lähettäjä lähettää pyynnön vastaanottajalle, johon vastaanottaja lähettää takaisin vastauksen. Tyypillinen esimerkki on asiakkaan ja palvelimen välinen kommunikointi. Kahdessa aikaisemmin mainitussa kommunikoinnissa yhteistä on, että osapuolet tietävät eksplisiittisesti toistensa identiteetin ja niiden välillä on yleensä vahva aikaliitos. *Epäsuora kommunikaatio* (*indirect communication*) tapahtuu käyttäen kolmatta osapuolta viestien välitykseen. Tämä mahdollistaa heikon aika- ja väliliitoksen osapuolten välille. Lähettäjä ei tiedä vastaanottajan identiteettiä ja toisin päin. Kolmas osapuoli vastaa viestin ohjaamisesta vastaanottajalle. [22, s. 43–45]

2.2.1 Prosessien välinen kommunikaatio

Prosessien välinen kommunikointi on matalan tason kommunikointia suoralla rajapinnan käytöllä. Esimerkkinä matalan tason kommunikoinnista on internet-protokollien, kuten UDP ja TCP, käyttö. Käyttöjärjestelmät tarjoavat matalan tason rajapinnan kommunikointiin, esimerkiksi pistokerajapinta. Pistokerajapinnalla voidaan käyttää sekä UDP:tä että TCP:tä [36, s. 1152]. Tyypillistä tämän tason kommunikoinnille se, että osapuolet tietävät toistensa identiteetin.

Käyttöjärjestelmän rajapintojen avulla prosessit voivat välittää viestejä suoraan toisilleen. Prosessit siis lähettävät ja vastaanottavat viestejä keskenään. Vastaanottaja yleensä toteuttaa jonon viestien vastaanottamiseen ennen käsittelyä. Jos kommunikointi on molemminsuuntaista, kummatkin osapuolet toteuttavat puskurin viestien vastaanottamiseen. Vastaanottaja voi kysellä tietoa jonosta toistuvasti saadakseen tiedon uudesta viestistä tai siitä voidaan ilmoittaa keskeytyksellä. Viestien lähetys voi olla synkronista tai asynkronista. Synkronisessa viestien vaihdossa lähettäjä ja vastaanottaja synkronoidaan jokaisella viestillä. Eli viestin lähetys ja vastaanotto ovat pysäyttäviä (blocking) operaatioita. Lähettäjän suoritus pysähtyy niin kauan, kunnes vastaanottaja vastaa ja suoritus jatkuu. Asynkronisessa kommunikoinnissa operaatiot eivät ole pysäyttäviä (non-blocking). Lähettäjä voi jatkaa muuta prosessointia heti kun viesti on kopioitu lähetyspuskuriin ja viestin lähetys jatkuu rinnakkain muun prosessoinnin kanssa. [22, s. 147–148]

Hyviä puolia matalan tason kommunikoinnissa on, että sillä saavutetaan saumaton kommunikointi prosessien välille ja, että sitä tukee moni nykyaikainen käyttöjärjestelmä. Matalan tason kommunikoinnin toteuttaminen vaatii paljon enemmän aikaa ja vaivaa ohjelmoijalta kuin korkeamman tason rajapinnan käyttö. Matalalla tasolla osapuolien välillä on vahva aika- ja väliliitos. Osapuolet tietävät toistensa identiteetit ja niiden täytyy olla olemassa samaan aikaan. Vastaanottajan täytyy olla vastaanottamassa viesti sen lähetyshetkellä. Lisäksi matalan tason ohjelma ei välttämättä ole modulaarinen ja on sidoksissa ympäristöönsä. Esimerkkinä Linux-käyttöjärjestelmälle toteutettu ohjelma ei toimi Windows-käyttöjärjestelmällä. Tähän kuitenkin voidaan vaikuttaa ohjelman sisäisillä abstrahointitasoilla.

2.2.2 Etäkutsu

Etäkutsu paradigmoja ovat *pyyntö-vastaus* (*request-reply*), *etäproseduurikutsu* (*remote procedure call*, lyhennetään *RPC*) ja *etämetodikutsu* (*remote method invocation*, lyhennetään *RMI*). Etäkutsuja voivat suorittaa prosessit tai korkeamman tason objektit ja palvelut. Primitiivisin etäkutsu paradigmoista on pyyntö-vastaus, joka edustaa mallia aikaisemmin mainitun viestien välityksen päällä ja lisää siihen kaksisuuntaisen viestien vaihdon. Paradigmaa käytetään asiakas-palvelin kommunikoinnissa. [22, s. 185–186]

Todella tärkeä tekniikka nykypäivän hajautettujen järjestelmien kannalta oli etäproseduurikutsut (RPC). Tämän määrittivät Birrell ja Nelson vuonna 1984 [8]. Se mahdollistaa ohjelman kutsua etänä toisella koneella sijaitsevaa proseduuria samoin kuin paikallista proseduuria. RPC-systeemi piilottaa taustalle kaiken kutsuun tarvittavan teknisen toteutuksen, kuten tiedon siirron, pakkaamisen ja purkamisen [22, s. 195–196]. Etämetodikutsun (RMI) toiminta on sama kuin RPC:n, mutta se on laajennettu toimivaksi olioilla. Olioiden metodeita voidaan kutsua niin kuin ne olisivat sen paikallisia metodeita [22, s. 204]. Sekä RPC ja RMI tarjoavat saman tason abstraktion ohjelmoijalle ja piilottavat teknisen toteutuksen alleen.

Etäkutsuparadigmat ovat korkeamman tason paradigmoja kuin aikaisemmin mainittu prosessien välinen kommunikointi. Hyvänä puolena paradigmoissa on, että ohjelmoijan ei välttämättä tarvitse kirjoittaa matalan tason rajapinnan ohjelmaa. Rajapinta on abstrahoitu helppokäyttöisemmäksi ja sen toteutus ei vie niin paljon aikaa ja resursseja kuin matalan tason implementointi. Etäkutsut kuitenkin kärsivät osapuolten välisistä vahvoista liitoksista kuten aikaisemmin mainittut prosessien väliset kutsut.

2.2.3 Epäsuora kommunikaatio

Epäsuora kommunikaatio tarkoittaa osapuolten välistä kommunikaatiota välittäjän avulla. Osapuolet eivät vaihda tietoa suoraan keskenään ja eivät tiedä toistensa identiteettiä. Välittäjä huolehtii viestien reitittämisestä ja lähettämisestä vastaanottajalle. Välittäjän tarkempi toiminta ja tarkoitus vaihtelee toteutuksesta riippuen. Aikaisemmin mainituissa kahdessa paradigmakategoriassa väli- ja aikaliitokset ovat pitkälti vahvoja. Poikkeuksena tähän on esimerkiksi IP-ryhmälähetys, jossa osapuolilla on heikko väliliitos, mutta vahva aikaliitos. Epäsuora kommunikointi vähentää liitoksien vahvuutta osapuolten välillä. Väli- ja aikaliitoksien heikkous vaihtelee toteutuksesta riippuen. Epäsuora kommunikointi mahdollistaa toisen osapuolen helpomman vaihdon, siirron ja päivittämisen kuin suora kommunikointi. Useasti epäsuorassa kommunikoinnissa vastaanottajia voi olla monta yhden sijaan. Tällöin voidaan puhua yksi-moneen-suhteesta (one-to-many) osapuolten välillä. Epäsuoruus tuo hyötyjä, mutta se tuo myös väistämätöntä suorituskykykuormaa epäsuoruuden vaativan toiminnan takia. [22, s. 230–231]

Epäsuoran kommunikoinnin paradigmoja ovat *joukkokommunikointi* (*group communication*), *julkaisija-tilaaja* (*publish-subscribe*), *viestijono* (*message queue*), *tuplesäilö* (*tuple*

space) ja *hajautetusti jaettu muisti* (*distributed shared memory*, lyhennetään *DSM*). Nämä esitettiin aikaisemmin taulukossa 2. Jokaista paradigmaa käsitellään erikseen tulevissa kappaleissa.

2.2.4 Joukkokommunikointi

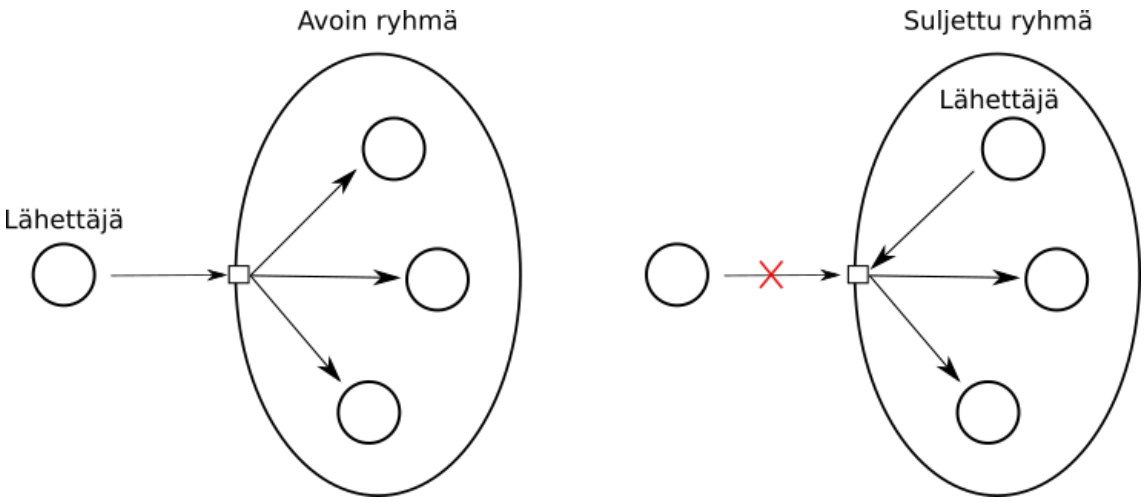
Joukkokommunikointi on paradigma, jossa osapuoli lähettää viestin ryhmälle, jonka jälkeen viesti uudelleenlähetetään kaikille ryhmän osapuolille. Ryhmä vastaanottajia tunnistetaan yksilöivällä ryhmätunnisteella. Lähettäjä käyttää tunnistetta kohdentamaan viestin haluamalleen ryhmälle. Tässä tapauksessa lähettäjä ei ole tietoinen ryhmässä olevien vastaanottajien identiteetistä [22, s. 232–233]. Joukkokommunikointi tunnetaan myös *joukkokommunikointisysteeminä* (*Group Communication System*, lyhennetään *GCS*). Briman keksi ja kuvasi ensimmäisen tällaisen systeemin 1986-luvulla [6]. Joukkokommunikointi on tärkeä paradigma hajautettujen järjestelmien kannalta ja sillä on olemassa monta eri käyttökohdetta. Applikaatioina esimerkiksi voi olla kollaboraatio- ja monitorointiohjelmistot, jossa kaikkien osapuolien täytyy tietää uusimmasta muutoksesta.

Joukkokommunikoinnissa ryhmät voivat olla ns. avoimia tai suljettuja. Kuvassa 2 on esitetty avoimen ja suljetun ryhmän toiminta. Avoin ryhmä mahdollistaa ulkopuolisen osapuolen lähettää viesti kaikille sen jäsenille. Suljettu ryhmä on tarkoitettu ainoastaan sen osapuolten väliseen kommunikointiin. Suljetussa ryhmässä yksi osapuoli voi ryhmälähettää viestin kaikille sen osapuolille ryhmän tunnisteella, samoin kuin ulkopuolinen avoimen ryhmän tapauksessa. Avoimen ja suljetun ryhmän lisäksi ryhmillä voi olla muitakin tärkeitä piirteitä. Ryhmät voivat olla päällekkäisiä tai erillisiä. Päällekkäiset ryhmät mahdollistavat yhden osapuolen kuulumisen yhtä aikaa moneen eri ryhmään. Ei päällekkäisessä osapuoli voi kuulua enimmillään yhteen ryhmään. Joissakin tapauksissa ryhmän osapuolten voidaan sallia liittyä ja lähteä ryhmästä. Tällöin erillisen osapuolen täytyy pitää kirjaa ryhmän osapuolista ja tarjota toiminnallisuudet liittyä ja poistua ryhmästä. [22, s. 233–235] [7, s. 48]

Ryhmät mahdollistavat yhden-moneen-suhteen kommunikoinnissa. Yksi viesti saadaan lähetettyä monelle osapuolelle yhdellä kertaa. Tämä säästää kaistaa verrattuna tilanteeseen, jossa viesti lähetettäisiin jokaiselle osapuolelle erikseen. Joukkokommunikoinnissa osapuolten välillä on heikko väliliitos, mutta vahva aikaliitos. Joukkokommunikointisysteemit ovat monimutkaisia ja niiden lähetystakuiden vaatimukset vaativat paljon vaivannäköä systeemiltä. Lähetystakuihin yleensä kuuluu, että jokainen ryhmän jäsen saa sille lähetetyn viestin ainakin kerran jossakin vaiheessa. Dynaamisen ryhmän tapauksessa, jossa osapuolia voi tulla ja lähteä ryhmästä, lähetystakuiden kiinni pitäminen monimutkaistuu entisestään. [15] [22, s. 236]

2.2.5 Julkaisija-tilaaja

Julkaisija-tilaaja on kommunikointiparadigma, jossa *julkaisijat* julkaisevat tapahtumia/-viestejä ja *tilaajat* tilaavat haluamiaan tapahtumia/viestejä *tilauksilla*. Osapuolten välissä

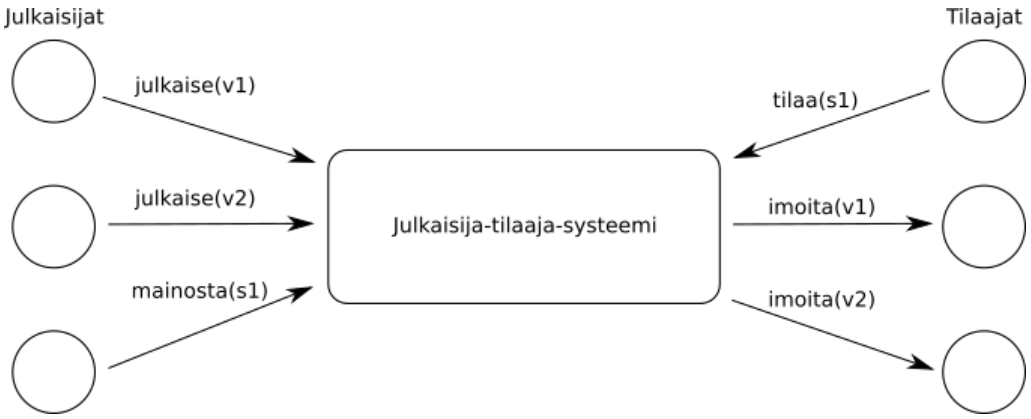


Kuva 2. *Osapuolten kommunikointi avoimessa ja suljetussa ryhmässä (pohjautuu kuvaan [22, s. 235]).*

on julkaisija-tilaaja-systeemi, joka vastaa viestien reitittämisestä tilaajille tehtyjen tilauksien perusteella. Järjestelmä mahdollistaa monen tilaajan tilata viestejä samalta julkaisijalta. Näin ollen kommunikointisuhde osapuolten välissä on yksi-moneen, kuten joukkokommunikoinnissa. Systeemiin tehdyllä tilauksella tilaaja voi tilata samalla kertaa monta eri julkaisijaa. Tehty tilaus on ns. *suodatin/malli* (*pattern*), joka systeemin sääntöjen mukaan voi täsmätä enemmän kuin yhteen julkaistuun viestiin. Kirjallisuudessa paradigma tunnetaan nimellä *julkaisija-tilaaja-systeemi* (*publish-subscribe system*) [5], sekä *hajautettu tapahtumapohjainen systeemi* (*distributed event-based system*) [46]. Julkaisija-tilaaja-paradigmalle on olemassa monia erilaisia käyttökohteita hajautetuissa järjestelmissä. Varsinkin systeemeissä, jossa erilaisten tapahtumien ja viestien jakaminen eri osapuolille on tarpeen.

Osapuolien kommunikointi julkaisija-tilaaja-systeemin kanssa tapahtuu funktioilla. Funktiot tarjoavat toiminnot, joilla osapuolet tekevät systeemin julkaisuja ja tilauksia. Kuvassa 3 on esitetty systeemin toiminta funktioita käyttäen. Julkaisija julkaisee viestin systeemiin funktiolla *julkaise(v)*, jossa parametri v kuvaa julkaistavaa viestiä. Tilaaja tilaa viestejä funktiolla *tilaa(s)*, jossa s-parametri tarkoittaa suodatinta viesteihin joihin tilaaja osoittaa kiinnostusta. Viestin saapuessa systeemiin ja tilauksen suodattimen täsmätessä, viesti lähetetään tilaajalle *ilmoita(v)* funktiolla, jossa v on julkaistu viesti. Tilaajien on myös mahdollista peruuttaa tilaus *peruuta_tilaus()* funktiolla. Joissakin systeemeissä julkaisijoiden on myös mahdollista mainostaa viestejä. Julkaisija kutsuu *mainosta(s)* funktiota, jossa s-parametri kertoo viestien tyypistä ja noudattaa samaa formaattia kuin tilaajien suodattimet. Mainostus voidaan perua *peruuta_mainos()* funktiolla. [5, s. 2–3] [46, s. 26–28]

Osapuolten kommunikointi on epäsuoraa ja niiden välillä on heikko väliliitos ja vahva aikaliitos. Systeemi osapuolten välissä mahdollistaa heikon väliliitoksen, jolloin ne eivät tiedä toistensta identiteettiä. Jotta tilaaja saa viestin, täytyy sen olla olemassa samaan aikaan kuin julkaisijan. Seurauksena tästä vahva aikaliitos osapuolien välille.



Kuva 3. *Julkaisija-tilaaja-systeemi välittäjänä viestien välittämisessä julkaisijoiden ja tilaajien välissä (pohjautuu kuvaan [22, s. 246]).*

Yksinkertaisin julkaisija-tilaaja-systeemi on helppo implementoida yhdellä välittäjällä. Tämä voi kuitenkin tulla ongelmaksi tai pullonkaulaksi järjestelmässä, missä tiedon vaihto on nopeaa ja osapuolia on paljon tai välittäjä kaatuu kesken kaiken. Tällaisen systeemin skaalaaminen ylöspäin on vaikeampaa ja vaatii hajautettua välittäjäverkkoa [22, s. 248–249]. Lisäksi systeemin epäsuoruus tuo mukanaan kommunikointiin vaikeutta ja suoritukseen kuormaa verrattuna suoraan kommunikointiin.

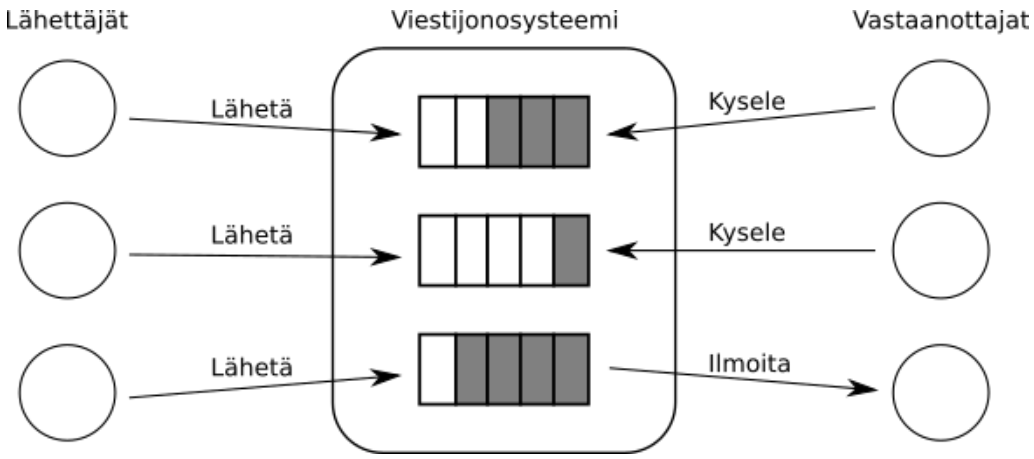
2.2.6 Viestijono

Viestijono paradigmassa osapuolet kommunikoivat epäsuorasti jonon välityksellä. Joukkokommunikointi ja julkaisija-tilaaja paradigmat ovat yksi-moneen-suhde osapuolten välillä, kun taas viestijono on yksi-yhteen-suhde (point-to-point). Lähettäjä lähettää viestin jonoon, josta vastaanottaja poistaa sen käsittelyyn. Viestijonon tarkoitus on puskuroida viestejä vastaanottajalle ja näin taata niiden saatavuus ja järjestys millä ajan hetkellä hyvänsä. Viestijono mahdollistaa siis osapuolten välillä heikon tila- ja aikaliitoksen. Jonon ansiosta lähettäjä ja vastaanottaja voivat olla olemassa eri aikaan [22, s. 254]. Kuvassa 4 on esitetty viestijonosysteemin toiminta osapuolten välillä.

Vastaanottaja voi poistaa viestejä jonosta kolmella eri periaatteella:

- *pysäyttävä vastaanotto* (*blocking receive*), joka pysäyttää suorituksen, kunnes viesti on saatavissa,
- *ei pysäyttävä vastaanotto* (*non-blocking receive*), joka tarkistaa jonon tilan ja palauttaa viestin jos saatavilla, ja
- *ilmoita* (*notify*), joka lähettää tiedon uudesta viestistä, kun sellainen on saatavilla [22, s. 254].

Ei pysäyttävä vastaanotto on sama kuin vastaanottaja kyselisi viestejä jonosta tietyin väliajoin (poll). Viestijonosysteemissä jonoa ei ole rajoitettu yhteen lähettäjään ja vastaanottajaan. Systeemissä moni lähettäjä voi lähettää viestejä samaan jonoon ja moni vastaa-



Kuva 4. *Viestijonosysteemi puskuroi viestejä lähettäjiltä vastaanottajille (pohjautuu kuvaan [22, s. 255]).*

nottaja voi vastaanottaa niitä samasta jonosta. Jonojen jonotuspolitiikka on yleensä FIFO-periaate (First-In-First-Out). Moni systeemi kuitenkin tukee myös muitakin periaatteita, kuten viestien priorisointia, jossa ylemmän prioriteetin viestit käsitellään ennen alemman prioriteetin viestejä [17, s. 120].

Viestijonon hyvänä puolena on osapuolten heikon aikaliitoksen mahdollistaminen. Tämä mahdollistetaan sillä, että viestit jonossa ovat pysyviä (persistent). Systeemi tallentaa viestit levylle, jossa ne säilyvät niin kauan, kunnes vastaanottaja poistaa ne jonosta. Systeemi takaa viestien lähettämisen vastaanottajalle. Viesti tullaan lähettämään jossakin vaiheessa ja enintään kerran. Lisäksi lähetetty viesti vastaa alkuperäistä lähettäjän viestiä. [22, s. 255]

Aikaisemmin käsiteltiin viestien välitystä prosessien välisessä kommunikoinnissa ja mainittiin että prosessit voivat implementoida jonon viestien vastaanottoon. Jonosysteemillä on paljon samanlaisia piirteitä tämän kanssa. Kuitenkin viestien välityksessä jonot ovat liitoksissa sen osapuoleen ja ovat implisiittisiä. Viestijonosysteemissä jonot ovat kolmannen osapuolen tarjoamia eksplisiittisiä jonoja, jotka erottavat lähettäjät ja vastaanottajat toisistaan. Tämä on tärkeä ero, joka tekee viestijonosta oman epäsuoran kommunikointiparadigman. [22, s. 256]

3. IEC 61850 -STANDARDI

IEC 61850 -standardi määrittää säännöt, kuinka sähköaseman laitteet kommunikoivat keskenään verkon yli [35, s. 1]. Standardi koskee myös aseman kanssa kommunikoivia ohjelmistoja [27, s. 10]. Standardi on laaja monesta dokumentista koostuva kokonaisuus. Tämän takia tässä luvussa standardista käsitellään vain diplomityön kannalta tärkeitä asioita. Käsittely aloitetaan yleiskuvalla ja edetään tärkeään asiaan. Saatuja tietoja tullaan käyttämään myöhemmin pohjana järjestelmän arkkitehtuurin ja ohjelmiston suunnittelussa.

3.1 Yleiskuva

Sähköaseman tehtäviin kuuluu mm. jännitteen muuntaminen, verkon jakaminen ja sen toiminnan tarkkailu. Virhetilanteissa aseman laitteisto toimii estääkseen suuremman vahingon. Näitä laitteita sähköasemalla ohjaa ja tarkkailee sen automaatiolaitteet, joita kutsutaan *älykkäiksi elektroniikkalaitteiksi* (*Intelligent Electronic Device*, *IED*). IED:t ovat kytketty sähköverkon laitteisiin joita ne ohjaavat [27, s. 63–64]. Näitä laitteita on mm. muuntajat ja katkaisijat. IED:t myös kytketään verkkoon, jotta ne voivat kommunikoida keskenään ja asemalta ulospäin [27, s. 31]. Asemalta ulospäin kommunikointi mahdollistaa sen etäohjauksen ja tarkkailun ohjelmiston avulla [9, s. 1].

IEC 61850 -standardi määrittää kuinka IED-laitteet ja etäohjelmistot kommunikoivat keskenään. Standardin tarkoitus on asettaa yhteiset säännöt, jotta eri valmistajien laitteet ja ohjelmistot olisivat yhteensopivia keskenään. Standardi on määritetty tekniikasta riippumattomaan muotoon, joka erikseen mallinnetaan halutulle tekniikalle [25]. Mallinnuksia on olemassa eri tekniikoille, mutta tässä työssä kommunikointi tapahtuu pelkästään *MMS*-protokollan (*Manufacturing Message Specification*) päällä. MMS-protokolla on TCP/IP-protokollaperheen päällä toimiva kommunikointiprotokolla [43].

Standardi määrittää säännöt kuinka asiakasohjelmat voivat tilata viestejä IED-laitteelta verkon yli. Viestien tilaus noudattaa julkaisija-tilaaja-paradigmaa, jossa IED-laite on julkaisija ja asiakasohjelmat tilaajia. Julkaisija-tilaaja-paradigma käsiteltiin aikaisemmin kappaleessa 2.2.5. Viestien sisältö on IED-laitteessa olevaa tietoa, kuten mittausdataa tai siihen kytketyn laitteen tilatietoa. Viestit tilataan IED-laitteelta olevalta oliolta, joka konfiguroidaan ennen tilausta. Standardi asettaa rajoituksia tilaajien määriin, jotka täytyy ottaa huomioon ohjelmiston suunnittelussa. Jotta näitä rajoituksia ja tilauksen toimintaa voidaan ymmärtää, täytyy ensin tarkastella, kuinka standardi määrittää muuttujia IED-laitteeseen. [28, s. 91–97]

3.2 Sähköaseman laiteiden mallinnus

Muuttujilla IED-laite mallintaa siihen kytkettyä sähköaseman laitetta kuten katkaisijaa tai muuntajaa. Muuttujat esittävät mm. linjasta mitattua jännitteen arvoa tai katkaisijan tilaa, joka voi olla auki tai kiinni. IED-laitteessa olevien muuttujien määrä vaihtelee sen mukaan mitä laitetta sen on tarkoitus ohjata. [27, s. 28]

Mitä muuttujia IED-laite sisältää määrittyy standardin mukaan. Standardi esittää luokkamäärityksiä kaikille sähköaseman laitteille mm. katkaisijoille ja muuntajille. Luokka määrittää mitä muuttujia sen instanssi tulee sisältämään jotka kuvaavat kyseistä laitetta. Luokasta muodostetaan instanssi IED-laitteelle sen mukaan, mitä laitetta se ohjaa. Esimerkiksi jos IED-laite ohjaa linjan katkaisijaa, määritetään IED-laitteeseen standardin määrittämä katkaisijaluokan instanssi. Tällöin IED-laite sisältää muuttujia katkaisimen tilasta ja sen ohjauksesta. [29, 30]

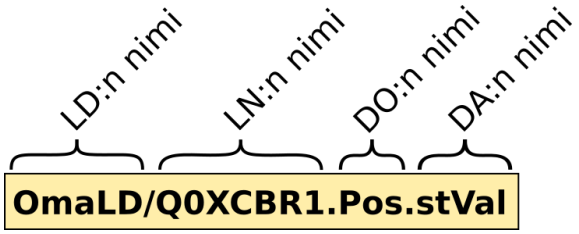
Standardin luokkamääritykset sisältävät aliluokkia, jotka vuorostaan sisältävät muuttujia tai muita aliluokkia. Standardissa luokkamääritykset toimivat samalla periaatteella kuin oliohjelmoinnin luokat. Luokat sisältävät samaan asiaan liittyviä muuttujia ja toiminnallisuutta. Näistä luokista ja muuttujista standardissa muodostuu oliopohjainen hierarkkinen rakenne. Esimerkiksi muuntajaa mallintava luokka sisältää aliluokkia mittaustiedon esittämisestä. Samaa aliluokkaa käytetään jonkin muun mittaustietoa sisältävän laitteen mallintamiseen. Voidaan sanoa, että standardin mukainen laitteen luokka koostuu muista aliluokista. Kun laitteen mallintavasta luokasta tehdään instanssi IED-laitteelle, samalla myös tehdään instanssi kaikista sen aliluokista. IED-laite siis sisältää myös kaikki aliluokkien muuttujat. [27, s. 108]

3.3 Muuttujien yksilöinti IED-laitteessa

IED-laite siis sisältää muuttujia hierarkiassa sen mukaan mitä laitetta se ohjaa. Näitä muuttujia voidaan lukea ja kirjoittaa standardin määrittämillä palvelukutsuilla. Jotta voidaan yksilöidä mitä muuttujia halutaan kutsulla lukea tai kirjoittaa, pitää niitä pystyä yksilöimään hierarkiassa. Tätä varten standardi määrittää viittaustekniikan, jota käytetään kutsun yhteydessä yksilöimään mitä muuttujia se koskee. Tämä viittausformaatti on määritetty kuvassa 5. Viittaus muodostuu hierarkiassa olevien luokkien instanssien ja muuttujien nimistä. Nimiä yhdistetään peräkkäin, jolloin viittausta voidaan seurata lukemalla minkä luokan instanssin mitä muuttujaa halutaan lukea tai kirjoittaa. Ensimmäisen ja toisen nimen väliin tulee kauttaviiva (/) ja loput yhdistetään pisteellä (.). [41, s. 625–626] [27, s. 93–95]

Viite muodostuu käsitteistä:

- *looginen laite* (*Logical Device*, *LD*),
- *looginen noodi* (*Logical Node*, *LN*),



Kuva 5. *IEC 61850 -standardin määrittämä viitteen rakenne (pohjautuu kuvaan [27, s. 93]).*

- *dataobjekti* (*Data Object*, *DO*) ja
- *data-attribuutti* (*Data Attribute*, *DA*).

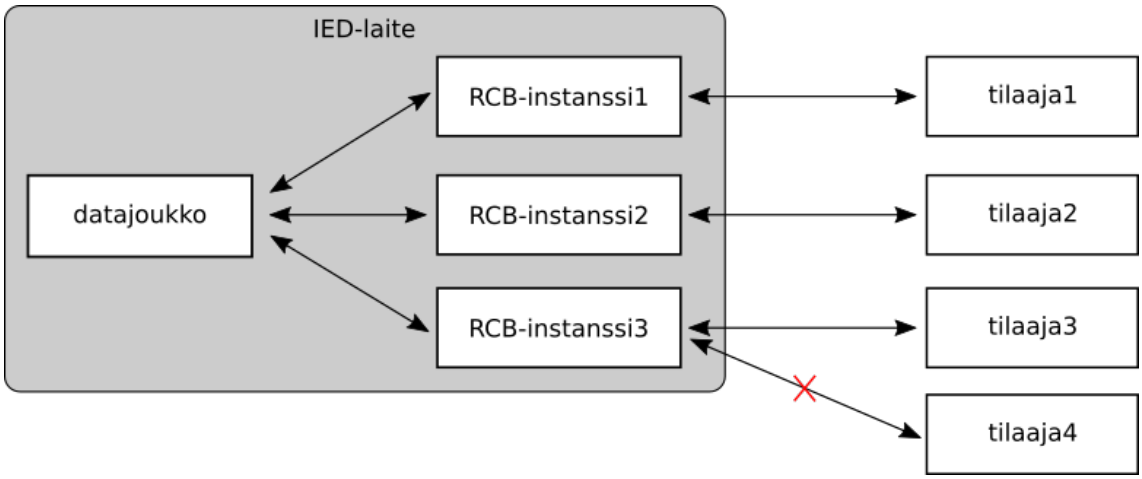
viitteessä vasemmalta oikealle järjestyksessä. Käsitteet ovat standardissa tapa esittää minkä tason luokka tai muuttuja hierarkiassa on kyseessä. Tämä tieto on diplomityön ulkopuolella ja lukija voi ne tarkistaa tarvittaessa standardista. Kuitenkin lyhyesti *looginen laite* esittää jotakin aseman fyysistä kokonaisuutta, jota IED-laite ohjaa. *Looginen noodi* on luokka laitteesta kuten muuntajasta tai katkaisijasta. *Dataobjekti* on luokka joka sisältää muuttujia samaan asiaan liittyen, esimerkiksi mittaukseen. *Data-attribuutti* on muuttuja, joka sisältää arvon mitatusta jännitteestä tai katkaisijan tilasta. [14, s. 2] [25, s. 24]

3.4 Viestien tilaus

Standardi määrittää luokkia eri asioiden mallintamiseen, joista tehdään instansseja IED-laitteelle tarpeen mukaan. Samaa periaatetta noudattaen standardi määrittää myös luokan, joka vastaa viestien tilauksesta. Standardissa luokkaa kutsutaan nimellä *RCB* (*Report Control Block*). Luokasta on tehty instanssi muuttujien hierarkiaan ja sillä on nimi. Luokan instanssi sisältää sisältää muuttujia tilauksen konfigurointiin, aloittamiseen ja lopettamiseen. Näitä muuttujia tilajaa voi kirjoittaa ja lukea palvelukutsuilla jotka viittaavaat RCB-luokan instanssiin. Esimerkki viitteestä RCB-instanssin arvojen kirjoittamiseen voi olla *OmaLD/LLN0.BR.OmaRCB*. [28, s. 95–97]

RCB-instanssi vastaa tilaajan tilauksen konfiguroinnista ja viestien lähettämisestä. Sisältö viestiin tulee standardin määrittämistä *datajoukoista* (*data set*). IED-laitteelle on mahdollista määrittää standardin mukaisia datajoukkoja. Datajoukko on kokoelma IED-laitteella olevista muuttujista, jotka on kerätty yhteen niiden tärkeyden takia. Esimerkiksi kaikki mittauspisteet voivat kuulua yhteen datajoukkoon. RCB-instanssi konfiguroidaan tarkkailemaan yhtä tällaista datajoukkoa, josta tilaaja on kiinnostunut. Datajoukossa tapahtuneen muutoksen myötä RCB-instanssi luo viestin, johon se sisällyttää muuttuneet arvot ja lähettää sen tilaajalle. Esimerkki tästä on jännitteen mitatun arvon muuttuminen uuteen, jonka tarkkaileva RCB-instanssi huomaa. Seurauksena instanssi luo uuden viestin, sisällyttää muuttuneet arvot viestiin ja lähettää sen tilaajalle. Sama toistuu minkä tahansa datajoukon muuttujan muuttuessa. [28, s. 93]

Standardi asettaa rajoitteita viestien tilaukseen. Yksi RCB-instanssi voi palvella vain yhtä tilaajaa kerrallaan. Instanssi niin sanotusti varataan tilauksen aloitushetkellä, jolloin muiden tilaajien kirjoitus luokkaan estetään, niin kauan kunnes nykyinen tilaaja lopettaa tilauksen tai yhteys katkeaa. Monen tilaajan halutessa saman datajoukon tiedot muutoksista, täytyy IED-laitteeseen määrittää RCB-instansseja tarpeen mukaan ja konfiguroida ne tarkkailemaan samaa datajoukkoa [28, s. 93]. Kuvassa 6 on esitetty tilanne, jossa kolme tilaaja tilaavat saman datajoukon käyttäen kolmea eri RCB-instanssia. Kuvassa neljäs tilaaja ei voi kirjoittaa jo varattua isntanssia, joten sen ei ole mahdollista tilata datajoukosta tulevia viestejä.



Kuva 6. *Kolme RCB-instanssia IED-laitteessa palvelee kolmea tilaajaa samasta datajoukosta.*

3.5 Viestien sisältö

Tilaaja siis tilaa viestejä datajoukon muuttujien tapahtumista. RCB-instanssi puskuroi viestejä sille asetetun ajan ja kaikki tämän ajan sisällä tapahtuneet muuttujien tapahtumat sisällytetään samaan viestiin [28, s. 98]. Seurauksena on, että tilaajille lähetetyt viestit sisältävät taulukon arvoja, joka vaihtelee viestien välillä. Viestin siis koostuu yleisistä tiedoista ja taulukosta muuttujien arvoja [28, s. 104]. Viestin yleiset tiedot sisältävät kopioita RCB-instanssin muuttujista. Tämä toimii tietona tilaajalle millä arvoilla instanssi on konfiguroitu tilausta varten.

RCB-instanssi sallii tilaajan konfiguroida viestin kenttien määrää. Tilaaja voi poistaa viestistä tarpeetonta tietoa. Joitakin vaihtoehtoisia kenttiä ovat mm. syykoodit muuttujan sisältämiseen viestiin ja muuttujan viite arvon lisäksi. Tilaaja voi myös konfiguroida liipaisimia minkä tapahtumien pohjalta viesti lähetetään. [27, s. 90] [28, s. 98]

Edellä esitetty viestin muoto on standardin mukainen abstrahoitu määritys. MMS-protokollan avulla viesti pakataan binäärimuotoon ja läheteään verkon yli tilaajalle. IEC 61850 standardin MMS-protokollan osa kertoo missä järjestyksessä bitit ovat [31]. Tilaaja on vastuussa viestin purkamisesta standardin mukaisesti ymmärrettävään muotoon.

3.6 Yhteenveto

Standardin mukaisesti viestit tilataan IED-laitteelta julkaisija-tilaaja-paradigman mukaan. Tähän standardi ei tarjoa muita vaihtoehtoja. Viestit tilataan IED-laitteelta olevilta RCB-luokan instansseilta. Rajoituksena instanssille oli, että se voi palvella vain yhtä tilaaja kerrallaan ja instansseja IED-laitteessa on rajallinen määrä. Tämän lisäksi IED-laite voi rajoittaa yhteyksien määrää kiinteään lukuun, jonka jälkeen se hylkää loput yhteydenotto-pyynnöt. RCB-instanssit ja yhteyksien määrät konfiguroidaan IED-laitteelle käynnistyksen yhteydessä ja niitä ei voi muuttaa ajon aikana [26]. Jos samaa datajoukkoa tarkkailee monta eri RCB-instanssia, lähetetään viestit IED-laitteessa tilaajille samasta tapahtumasta rinnakkain. Näitä tietoja tullaan myöhemmin käyttämään arkkitehtuurin ja ohjelmiston suunnittelussa.

Standardin abstrahoituja määrityksiä on mallinnettu eri tekniikoille. Tässä työssä kommunikointiin käytetään MMS-protokollaa, joka on TCP/IP-protokollaperheen päällä toimiva kommunikointiprotokolla. IED-laitteen kanssa kommunikoiva tilaajan täytyy purkaa binäärimuotoinen viesti ohjelmointikielen rakenteiksi. Diplomityössä toteutettu ohjelmisto käytti C-kielen *libIEC61850*-kirjastoa kaiken MMS-kommunikoinnin toteuttamiseen [40]. Kirjasto tarjoaa korkean tason funktioita ja datarakenteita tiedon käsittelyyn ilman tietoa MMS-protokollan toiminnasta.

4. JÄRJESTELMÄN SUUNNITTELU

Diplomityön suunnittelu aloitetaan ensin hajautetun järjestelmän arkkitehtuurin suunnittelusta. Tässä luvussa arkkitehtuuri suunnitellaan analysoimalla aikaisemmin läpikäytyjä hajautetun järjestelmän paradigmoja ja IEC 61850 -standardin asettamia määrityksiä. Tuloksena on arkkitehtuurisuunnitelma osaksi muuta järjestelmää, joka määrittää kommunikointiparadigmat kuinka viestit kulkevat IED-laitteilta järjestelmän muille komponenteille. Tulevissa luvuissa tätä suunnitelmaa tullaan tarkentamaan toteutukseen käytettävillä tekniikoilla, joka työn lopussa toteutetaan ohjelmistoksi.

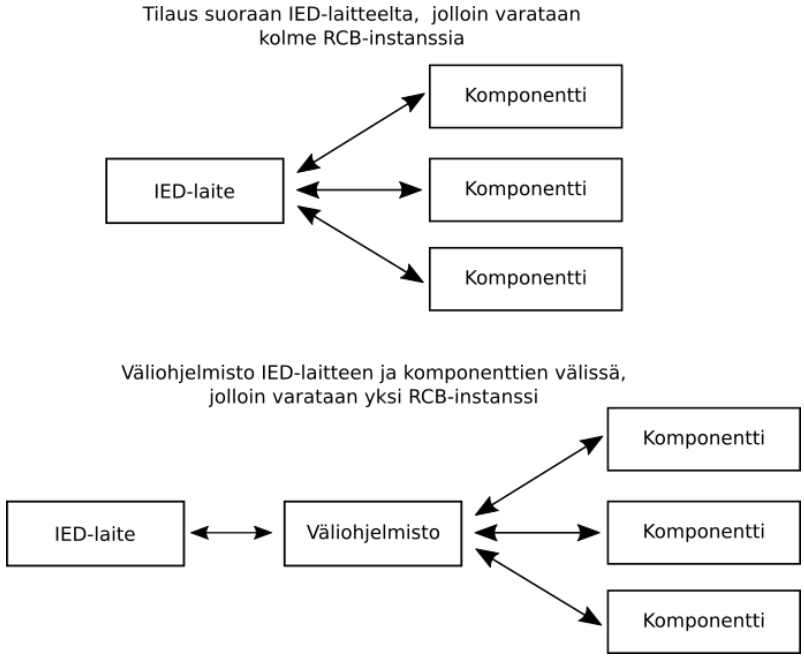
4.1 Arkkitehtuurin analyysi

IEC 61850 -standardi määrittää, että viestit IED-laitteelta tulevat julkaisija-tilaaja-paradigman mukaisesti. Viestit tilataan IED-laitteelta olevilta RCB-luokkien instansseilta. RCB-instanssi on kytketty laitteella olevaan datajoukkoon, jonka tiedoista ollaan kiinnostuneita. Rajoituksena standardi asetti, että tilaajan täytyy varata RCB-instanssi ja yksi RCB-instanssi voi palvella vain yhtä tilaajaa kerrallaan. RCB-instansseja IED-laitteissa on rajallinen määrä yhtä datajoukkoa kohti ja näitä voidaan myös käyttää aseman sisäiseen toimintaan. Tässä työssä käsitellyssä IED-laitteessa yhteen datajoukkoon oli esimerkiksi määritetty viisi eri RCB-instanssia. Seurauksena on, että kyseisen datajoukon voi tilata vain viisi saman aikaista tilaajaa. Lisäksi tässä työssä testattu IED-laite rajoitti avoimet yhteydet viiteen. Tästä ylimeneviä yhteyksiä ei avattu, ja IED-laite palautti standardin mukaisen negatiivisen vastauksen.

Vaatimuksissa määriteltiin, että IED-laitteen tilatut viestit täytyy saada jaettua järjestelmän muiden komponenttien kanssa. Työn tekohetkellä komponentteja järjestelmässä oli kaksi: mittaustiedon näyttämiseen ja tilatiedon tarkkailuun. Vaatimuksissa kuitenkin määriteltiin, että toteutuksessa haluttiin varautua tulevaisuuden varalle komponenttien suhteen. Eli tilanteeseen, jossa niitä olisi enemmän kuin kaksi ja niiden määrä voisi vaihtua tilaamisen aloitusten välillä.

Edelle mainituista IED-laitteen rajoituksista ja työlle asetetuista vaatimuksista voidaan tehdä johtopäätöksiä hajautuksen arkkitehtuurin suhteen. Samalle IED-laitteelle avattujen yhteyksien määrä halutaan pitää pienenä. Tämän lisäksi varattujen RCB-instanssien määrä jokaista datajoukkoa kohti halutaan myös pitää pienenä. Edellä mainittujen perusteella aseman resursseja halutaan varata mahdollisimman vähän ja vapauttaa niitä aseman muuhun toimintaa. Lisäksi vaaditut resurssit pitää pystyä määrittämään ennalta. Tämä helpottaa aseman insinöörin suunnittelutyötä, koska hän voi ottaa luvut huomioon IED-laitteiden konfiguroinnissa.

Edellä mainittuun tilanteeseen ratkaisuna olisi ollut, että järjestelmän yksittäiset komponentit tilaisivat niiden tarvitsemat tiedot suoraan IED-laitteelta. Tämä on esitetty kuvan 7 yläosassa. Ongelmana tässä kuitenkin on aikaisemmin mainittu yhteyksien määrän rajoitus IED-laitteelle ja se, että niiden määrä haluttiin minimoida ja pitää vakiona. Lisäksi viestit tilataan ja palautetaan komponentille MMS-protokollamäärityksien mukaisesti. Seurauksena on, että jokainen komponentti joutuu käsittelemään MMS-protokollan binääridataa itse tai kirjaston avulla. Viestin muoto olisi parempi olla ymmärrettävämpi. Näin viestin lukeminen eri tekniikoilla olisi helpompaa. Tämä asetettiin ohjelmiston vaatimuksissa ja oli myös yksi kohta työlle asetetuista tutkimuskysymyksistä.



***Kuva 7**. IED-laitteelta viestien tilaus suoraan ja väliohjelmiston avulla.*

Ratkaisuna edellä mainittuun tilanteeseen on kommunikoinnin epäsuoruuden lisääminen. Epäsuoruutta saadaan aikaan toteuttamalla erillinen ohjelmistokomponentti IED-laitteen ja muiden järjestelmän komponenttien väliin. Tätä komponenttia voisi kutsua myös nimellä *väliohjelmisto* (*middleware*) ja on esitetty kuvan 7 alaosassa. Väliohjelmisto tilaisi viestit IED-laitteelta ja halutuilta RCB-instansseilta. Samalla komponentti muokkaisi saapuvat viestit parempaan muotoon, joka olisi helpompi muiden järjestelmän komponenttien lukea. Väliohjelmiston avulla yhteyksien määrä IED-laitetta kohti saadaan yhteen. Tämän avulla myös saadaan minimoitua varattujen RCB-instanssien määrä datajoukkoa kohti yhteen. Tuloksena on vakiomäärät yhteyksiä ja RCB-instansseja datajoukkoa kohti, jotka ovat helpompi ennustaa ja konfiguroida etukäteen tarpeiden mukaan.

Vaatimuksissa mainittiin myös, että muu järjestelmä ohjaa viestien tilauksen aloittamista ja lopettamista. Väliohjelmistoa on mahdollista muun järjestelmän ohjata tilauksien mukaan. Väliohjelmisto ja epäsuoruus myös helpottavat viestien puskuroinnin toteuttamista komponenteille, joka myös oli vaatimuksena. Ilman epäsuoruutta, jokaisen komponentin tulee toteuttaa oma sisäinen puskuri viestien vastaanottamiseen.

Tässä kappaleessa analysoitiin IEC 61850 -standardin asettamia rajoja ja ohjelmistolle asetettuja vaatimuksia. Näistä analysoitiin mikä olisi toimiva ratkaisu tässä tilanteessa järjestelmän hajauttamiseen korkealla tasolla. Tuloksena järjestelmään päätettiin lisätä epäsuoruutta toteuttamalla väliohjelmisto IED-laitteen ja järjestelmän muiden komponenttien väliin. Tämän jälkeen pitää vielä miettiä vastaukset seuraaviin kysymyksiin:

- Mitkä kommunikointiparadigmat toteuttavat väliohjelmiston ja komponenttien vaatimukset?
- Mihin muotoon matalan tason MMS-viesti väliohjelmistossa tulee muuntaa jakoa varten?
- Mitkä ovat väliohjelmiston ja muiden komponenttien välisten liitoksien vahvuudet?
- Kuinka viestien puskurointi väliohjelmiston avulla toteutetaan?

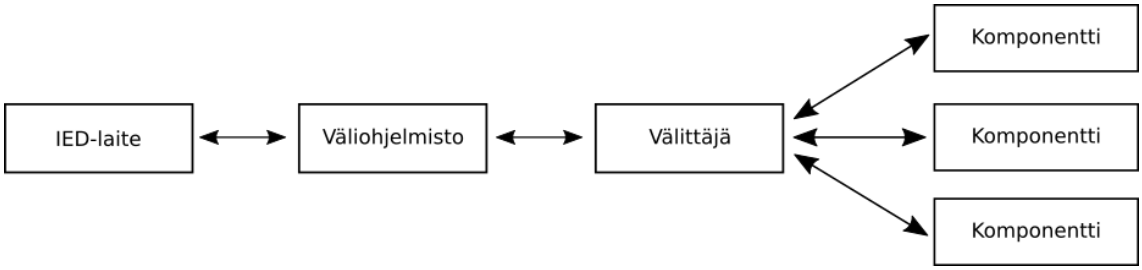
4.2 Osapuolten liitoksien analyysi

Aikaisemmin kappaleessa 2.1.2 käsiteltiin hajautetun järjestelmän osapuolten välisiä liitoksia. Erilaisten liitoksien luokittelut esiteltiin taulukossa 1. Lisäksi hajautetussa järjestelmässä kommunikointi osapuolten välillä voi olla suoraa tai epäsuoraa. Aikaisemmin kohdassa 4.1 päädyttiin lisäämään epäsuoruutta IED-laitteen ja järjestelmän muiden komponenttien väliin.

IED-laitteen ja viestien tilaajan välinen kommunikointi on suoraa ja se ei tapahdu välittäjän kautta. Lisäksi osapuolten välillä on vahva tila- ja aikaliitos. Väliliitos tulee, kun kummatkin osapuolet tietävät toistensa identiteetin (IP-osoitteen). Aikaliitoksessa osapuolien pitää olla olemassa samaan aikaan. Tilaajan pitää olla ottamassa viesti vastaan, kun IED-laite sen lähettää ja IED-laitteen pitää olla olemassa, kun tilaus tehdään. Kaikki edellä mainitut tulevat IEC 61850 -standardin määrityksien seurauksena ja näihin ei työn puitteissa voi vaikuttaa.

Työn aloitushetkellä järjestelmässä oli kaksi tietoa tarvitsevaa komponenttia. Vaatimusten mukaan haluttiin varautua tulevaisuuteen, jossa niitä olisi enemmänkin. Tästä seurauksena väliohjelmiston ja muiden järjestelmän komponenttien välinen suhde on yksimoneen. Vaatimuksena oli, että tietoa tarvitsevien komponenttien määrä pystyisi muuttumaan tilauksien välillä. Toteuttamalla suora kommunikointi väliohjelmiston ja järjestelmän muiden komponenttien väliin, väliohjelmiston täytyy tietää muiden komponenttien olemassaolosta ja kenelle viestit ohjataan. Tämä lisäisi osapuolten välistä riippuvuutta ja vähentäisi toteutuksen joustavuutta. Ratkaisuna on lisätä epäsuoruutta väliohjelmiston ja muiden komponenttien väliin välittäjällä. Tämä on esitetty kuvassa 8. Välittäjä vähentää osapuolten välistä riippuvuutta ja lisää toteutuksen joustavuutta. Se myös auttaa toteuttamaan asetettuja vaatimuksia, kuten viestien puskurointia ja uuden viestin ilmoitusta. Vaatimuksena oli, että IED-laitteelta tulevaa tietoa täytyy pystyä jakamaan komponenteille IED-laitteen perusteella. Välittäjän tuoma epäsuoruus auttaa myös tämän vaatimuk-

sen toteuttamisessa. Tällä vastuu väliohjelmistolta viestien jakamisesta saadaan siirrettyä välittäjän vastuulle.



Kuva 8. *Välittäjä väliohjelmiston ja komponenttien välissä lisäämään epäsuoruutta ja joustavuutta.*

Väliohjelmiston ja muiden komponenttien välille edellä mainitun perusteella halutaan heikko väliliitos. Väliohjelmiston ei tarvitse tietää järjestelmän muiden komponenttien identiteettiä ja tämä parantaa toteutuksen joustavuutta. Aikaliitos asetettujen vaatimuksien perusteella pystyisi olemaan vahva tai heikko. Kuitenkin muu järjestelmä ohjaa tilauksen aloittamista ja lopettamista. Seurauksena on, että väliohjelmisto ja muut komponentit ovat olemassa samaan aikaan. Tämä poistaa tarpeen heikolle aikaliitokselle.

Tässä kappaleessa analysoitiin hajautetussa järjestelmässä osapuolten välisten liitoksien vahvuutta. IEC 61850 -standardi asettaa rajoitteet IED-laitteen ja väliohjelmiston välille, joihin ei voida vaikuttaa. Väliohjelmiston ja muiden komponenttien väliin päädyttiin toteuttamaan epäsuora kommunikointi joustavuuden ja vaatimusten takia. Samojen osapuolten väliin halutiin toteuttaa heikko väliliitos ja vahva aikaliitos. Kysymyksenä jää miettiä millä hajautuksen paradigmoilla välittäjä tulee toteuttaa, jotta halutut ominaisuudet saadaan toteutettua?

4.3 Paradigmojen analyysi

Aikaisemmissa kappaleissa analyysien ja vaatimusten pohjalta päädyttiin kuvan 8 hajautetun järjestelmän arkkitehtuuriin. Tässä kappaleessa vertaillaan ja analysoidaan eri hajautuksen paradigmoja. Tarkoituksena on selvittää mitä paradigmoja tarvitaan, jotta asetetut vaatimukset voidaan toteuttaa. Näitä olivat viestien puskurointi, ilmoitukset komponenteille uuden viestin saapuessa ja viestien suodattamisen mahdollisuus IED-laitteen mukaan. Tässä kappaleessa käsitellyt paradigmat koskevat kuvan 8 välittäjää väliohjelmiston ja järjestelmän komponenttien välissä. Hajautetun järjestelmän eri paradigmat oli esitetty taulukossa 2.

Järjestelmään haluttiin epäsuoruutta välittäjällä. Tätä parhaiten tarjoavat epäsuorat kommunikointiparadigmat. Vaatimuksena oli, että viestejä pitää pystyä puskuroimaan myöhempään ajan hetkeen, jos komponentti ei niitä ehdi käsitellä. Tämän ominaisuuden parhaiten tarjoaa viestijonoparadigma. Viestejä haluttiin erotella IED-laitteen mukaan, ja komponentit voisivat päättää itse mistä ovat kiinnostuneita. Tätä ominaisuutta epäsuorista

paradigmoista tarjoavat joukkokommunikointi ja julkaisija-tilaaja. Joukkokommunikoinnissa komponentti pystyisi olemaan osa haluttuja ryhmiä. Julkaisija-tilaaja-paradigmassa komponentti pystyisi tilaamaan halutut viestit. Kummatkin edellä mainituista tarjoavat myös halutut ilmoitukset. Joukkokommunikointi- ja julkaisija-tilaaja-systeemissä vastaanottajat saavat ilmoituksen uuden viestin saapuessa. Jotta joukkokommunikoinnilla saadaan haluttu toiminnallisuus, tulee ryhmän olla avoin ja sen pitää mahdollistaa osapuolten liittyminen ja poistuminen ryhmästä. Lisäksi ryhmien pitää sallia olevan päällekkäisiä. Tämä sen takia, että komponentin on mahdollisesti saada viestejä useammasta lähteestä samaan aikaan.

Järjestelmään halutun epäsuoruuden takia suorat kommunikointiparadigmat eivät sovellu kovin hyvin hajautuksen toteuttamiseen. Näitä olivat prosessien välinen kommunikointi ja etäkutsut. Hajautuksessa osapuolien välillä haluttiin olevan heikko väliliitos ja vahva aikaliitos. Kummassakin edellä mainitussa paradigmassa liitokset ovat vahvoja. Suoralla kommunikoinnilla vastaanottavan osapuolen pitää toteuttaa viestipuskuri itse, verrattuna jos käytettävä epäsuora systeemi tarjoaisi puskurin sen sijaan. Tämä hankaloittaa uusien komponenttien kehitystyötä. Muu järjestelmä on toteutettu web-sovelluksena, josta seurasi vaatimus, että kommunikoinnin pitää toimia TCP/IP-protokollaperheen päällä tai vastaavalla. Prosessien välinen kommunikointi tapahtuu mm. pistokkeilla ja näin ollen on vaatimukseen nähden liian matalaa. Etäkutsut tarjoavat tähän paremman vaihtoehdon, mutta vaatimuksissa ei ollut tarvetta pyyntö-vastaus-kommunikoinnille. Viestien suunta on tässä tapauksessa yhden suuntaista IED-laitteelta järjestelmän komponentille.

Asetettuihin vaatimuksiin nähden epäsuorat paradigmat sopivat toteutukseen parhaiten. Näistä tarvitaan viestijonoparadigmaa viestien puskurointiin ja joukkokommunikointi tai julkaisija-tilaaja niiden lähettämiseen komponenteille. Joukkokommunikointi ja julkaisija-tilaaja paradigmat ovat kummatkin tilanteeseen sopivia vaihtoehtoja. Vielä pitää miettiä millä tekniikalla halutut paradigmat toteutetaan. Valitun tekniikan tulee yhdistää kaksi eri paradigmaa niin, että viestien oikea reitittäminen ja puskurointi ovat mahdollisia.

4.4 Kommunikointitekniikoiden vertailu

Hajautetun järjestelmän toteuttamiseen löytyy erilaisia avoimia standardeja kuten *AMQP* (*Advanced Message Queuing Protocol*) [2] ja *MQTT* (*Message Queuing Telemetry Transport*) [44]. Standardien tarkoitus on määrittää yhteiset säännöt osapuolten kommunikointiin hajautetussa järjestelmässä. Standardien pohjalta on toteutettu erilaisia viestien välitysohjelmistoja, johon muut osapuolet voivat ottaa yhteyttä standardin mukaisesti tekniikasta riippumatta. Kummatkin edellä mainitut standardit ovat määritetty toimivaksi TCP/IP-protokollaperheen päällä [48, s. 1] [1, s. 22].

MQTT on julkaisija-tilaaja-paradigmaan pohjautuva kommunikointistandardi. MQTT-määrityksien mukainen välittäjäpalvelin tarjoaa myös viestien puskuroinnin mahdollisuuden [48]. MQTT on pääasiassa tarkoitettu kommunikointiin, missä kaista on rajallista ja

kommunikoinnista huolehtivan koodin jalanjälki pitää olla pientä. Tästä syystä se on paljon käytetty standardi IoT-laitteissa (Internet of Things) ja kotiautomaatiossa [24, s. 9–11].

AMQP-standardi on tarkoitettu MOM:in toteuttamiseen (Message-Oriented Middleware) ja näin ollen mahdollistaa monen eri kommunikointiparadigman toteuttamisen [1, s. 6]. MOM tarkoittaa hajautetussa järjestelmässä *väliohjelmistoa* (middleware), jota käytetään lähettämään ja vastaanottamaan viestejä. MOM:in tarkoitus on tarjota heterogeeninen alusta viestien vaihtoon tekniikasta ja verkkoprotokollasta riippumatta [42]. AMQP-standardi mahdollistaa esimerkiksi viestijonot, etäkutsut ja erilaiset reitit ykset kuten pisteestä pisteeseen ja julkaisija-tilaaja. AMQP tarjoaa toteutukseen enemmän paradigma vaihtoehtoja kuin MQTT.

Tässä työssä toteutuksen tekniikaksi valittiin AMQP. AMQP tarjoaa kaikki vaaditut ominaisuudet välittäjän toteuttamiseen. AMQP ei mahdollista joukkokommunikoinnin toteuttamista, mutta mahdollistaa julkaisija-tilaaja-paradigman. Aikaisemmin julkaisija-tilaaja-paradigma todettiin toteutukseen sopivaksi vaihtoehdoksi joukkokommunikoinnin kanssa. AMQP tarjoaa viestien puskuroinnin jokaista tilaajaa kohden ja komponenttien on mahdollista myös suodattaa viestejä kiinnostuksensa mukaan. MQTT olisi myös todennäköisesti sopinut toteutukseen, koska se tarjosi kaikki vaaditut ominaisuudet. Työssä kuitenkin päädyttiin AMQP:n valintaan tekijän aikaisemman kokemuksen ja muiden työntekijöiden kanssa käydyn keskustelun tuloksena. Viestien lähetykselle järjestelmässä ei vaadittu takuita, mutta nämä olisi hyvä olla olemassa tulevaisuuden takia. AMQP tarjoaa takuumekanismit viestien lähettämiseen transaktioina ja vastaanottamiseen niiden kuittaamisella [1, s. 14,21]. Kuinka AMQP tarkalleen toimii, on tämän diplomityön ulkopuolella ja siitä kerrotaan vain lyhyesti kohdissa missä tietoa tarvitaan. Lukija lukea enemmän asiasta sen dokumentaatiosta [1] ja verkkosivulta [2].

4.5 Viestin formaatti

IED-laitteelle kommunikointi ja siltä tulevat viestit lähetetään MMS-protokollan binäärimäärityksien mukaan. Jotta tilaavien komponenttien olisi helppo lukea viestin sisältö, täytyy se muuntaa toiseen muotoon. Hajautetussa järjestelmässä viestejä voidaan lähettää lähettäjän formaatin tai yhteisen hyväksytyn formaatin mukaan. Jos viestin lähettäjä päättää formaatin, täytyy viestissä olla tieto sen muodosta, jotta vastaanottajan on mahdollista se lukea. Järjestelmän yhteisessä formaatissa tätä tietoa ei tarvita, koska kaikki osapuolet käyttävät samaa formaattia ilman poikkeuksia. Viestejä voidaan lähettää binääri- tai tekstimuodossa. Binäärimuodossa datarakenteet esitetään binääreinä ja tekstimuodossa ne esitetään tekstimuodossa. Tekstimuoto on yleensä pidempi esitysmuoto kuin binäärimuoto.

Aikaisempien kappaleiden pohjalta päädyttiin kuvan 8 mukaiseen arkkitehtuuriin. Arkkitehtuurissa väliohjelmiston tehtävä on tilata viestejä IED-laitteelta, muuntaa ne ymmär-

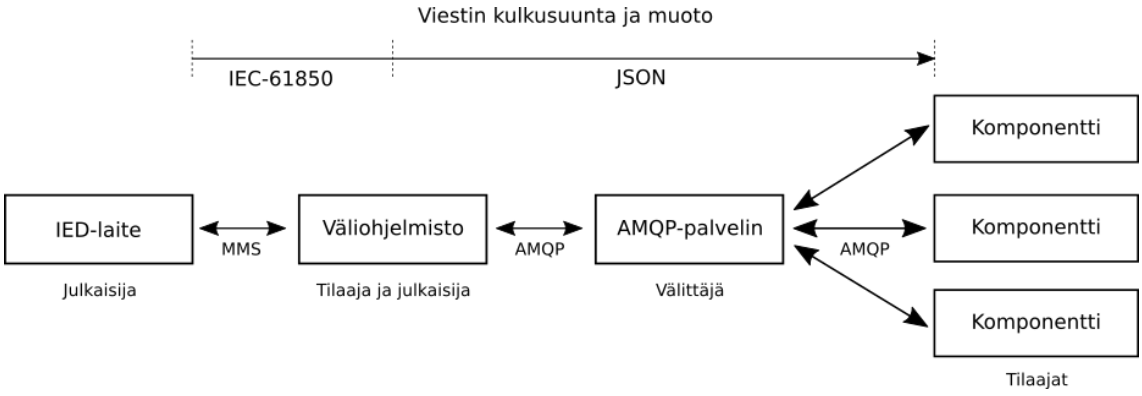
rettävään muotoon ja julkaista AMQP-välittäjäpalvelimelle. Järjestelmän komponentit tilaavat viestejä AMQP-palvelimelta tarpeidensa mukaan.

Järjestelmän komponenteille viestin lukeminen haluttiin pitää helppona ja viestin koolla ei ollut teknisiä rajoitteita. Tämän takia viestit päätetiin esittää mieluummin tekstiformaatissa. Tekstimuoto on helpompi ihmisen lukea esimerkiksi vikatilanteissa. Binääritiedon lukeminen yleensä tarvitsee erillisen ohjelman. Nykypäivänä web-teknologioissa on käytössä kaksi tunnettua tekstimuotoista esitystapaa, joita ovat *XML* (*Extensible Markup Language)*) [18] ja *JSON* (*JavaScript Object Notation*) [32].

Työssä viestien välityksen tekniikaksi valittiin JSON. Viestin muutoksen JSON-muotoon tekee väliohjelmisto, joka tilaa viestit IED-laitteelta. JSON on nopeampi ja kevyempi tiedonsiirtoformaatti kuin XML [47]. Lisäksi JSON on nykypäivänä paljon käytetty tiedonsiirron muoto web-rajapinnoissa kuten REST (Representational State Transfer). JSON:in lukeminen ihmiselle on helppoa ja sille on olemassa tuki valmiina monelle eri ohjelmointikielelle ilman erillistä kirjastoa. JSON on myös kasvattanut suosiotaan ajan saatossa XML:ään verrattuna [23]. Ja asiasta on kirjoitettu erilaisia blogiposteja kuten [62, 63, 52].

4.6 Yhteenveto

Eri analyysien ja vaatimuksien pohjalta päädyttiin lopulta kuvassa 9 olevaan järjestelmän arkkitehtuuriin. Kuvassa on esitetty viestin kulku järjestelmän läpi ja sen muoto. Lisäksi osapuolten väliin on merkitty niiden käyttämät kommunikointiprotokollat.



Kuva 9. *Suunniteltu korkean tason järjestelmän hajautus ja kommunikointiprotokollat osapuolten välillä.*

Arkkitehtuurissa väliohjelmisto tilaa viestejä IED-laitteelta IEC 61850 -standardin mukaisesti MMS-protokollan avulla. Viesti päädyttiin muuttamaan JSON-formaattiin helpomman luettavuuden saamiseksi ja sen muuntamisen hoitaa väliohjelmisto. Välittäjän kommunikointiprotokollaksi valittiin AMQP-standardi ja kommunikointiparadigmaksi julkaisija-tilaaja ja viestijono. Väliohjelmisto tilaa viestejä IED-laitteelta ja uudelleenjulkaisee ne JSON-muodossa välittäjälle. Järjestelmän muut komponentit tilaavat viestejä välittäjältä AMQP-standardin mukaisesti. Suunniteltua mallia tullaan tarkentamaan teknisesti ennen toteutusta.

5. DEMOVERSION TOIMINTA JA ANALYYSI

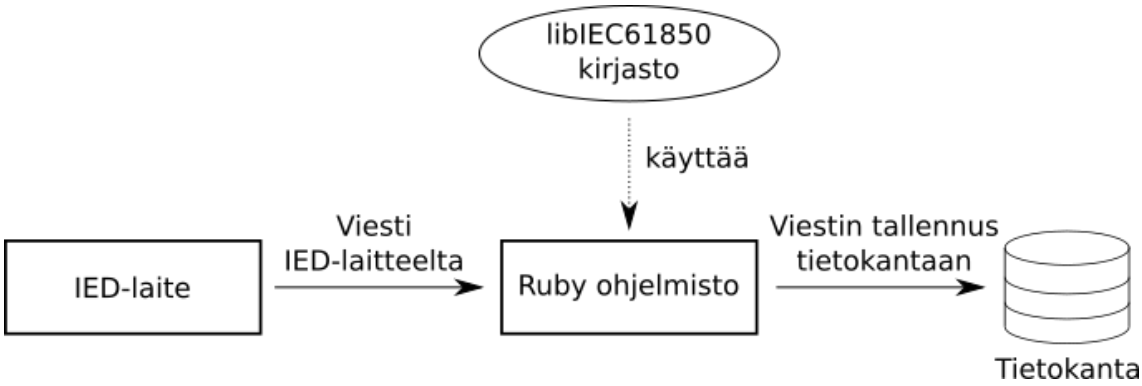
Ennen tämän työn aloitusta diplomityön tekijä oli kehittänyt ohjelmiston, joka kykeni viestien tilaukseen ja tallentamiseen. Ohjelma oli tarkoitettu demoksi tai prototyypiksi ennen varsinaista toteutusta. Demon tarkoituksena oli opettaa tekijälle IEC 61850 -standardia, todentaa viestien tilaamisen mahdollisuus ja tuoda esille toteutukseen liittyviä ongelmia. Ohjelma kykeni tilaamaan viestejä yhden IED-laitteen kaikilta RCB-instansseilta, prosessoimaan viestit ja tallentamaan ne relaatiotietokantaan.

Demoa ei ollut mahdollista käyttää tuotannossa osana muuta järjestelmää sen arkkitehtuurin ja toiminnan epävarmuuden takia. Tässä kappaleessa käydään läpi demon toimintaa ja siihen liittyviä ongelmia. Demosta on tarkoitus analysoida sen toiminnan epävarmuutta ja saada selville mistä se mahdollisesti johtui. Näitä tietoja käytetään myöhemmin apuna uuden version suunnittelussa.

5.1 Arkkitehtuuri

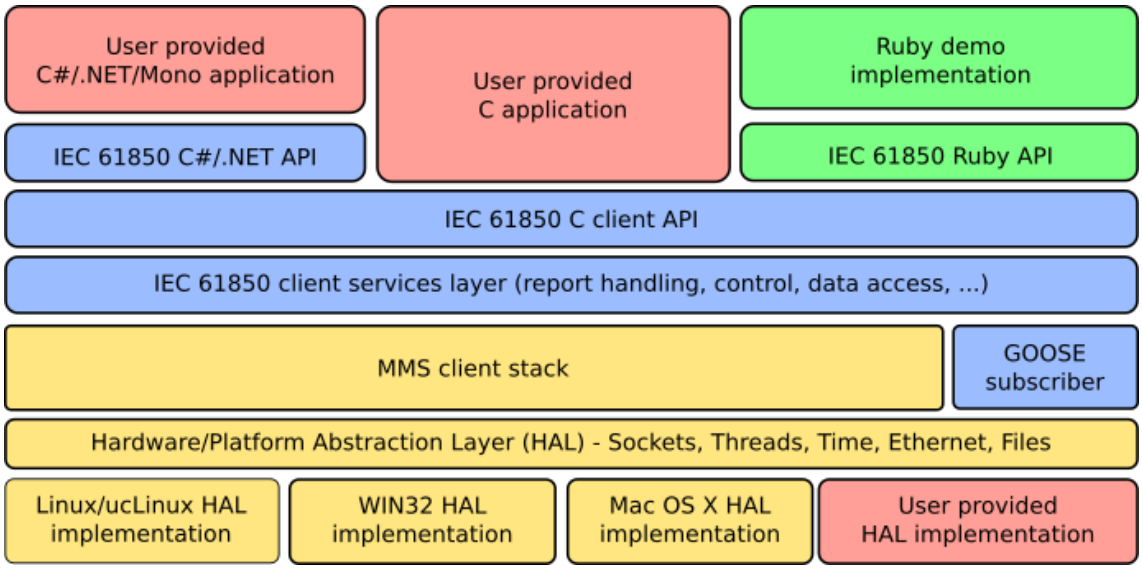
Demoversio oli ohjelmoitu Ruby-ohjelmointikielellä. Kuvassa 10 on esitetty demoversion arkkitehtuuri korkealla tasolla ja kuinka viesti IED-laitteelta siirtyy tietokantaan. Yksi ajettu demoversion prosessi pystyi tilaamaan yhden IED-laitteen kaikki RCB-luokkien instanssit. Instanssien tiedot luettiin relaatiotietokannasta. Ohjelmisto prosessoi viestit ja tallensi ne relaatiotietokantaan myöhempää käyttöä varten. Ruby-ohjelmistossa tärkeässä osassa oli *libIEC61850*-kirjasto [40]. 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 standardin määrittämien luokkien 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. Demoa toteutettiin käyttäen oikeaa IED-laitetta, joten kirjastosta tarvittiin vain sen asiakasohjelman ominaisuudet. Demon toteutuksen aikana kirjasto todettiin hyväksi vaihtoehdoksi ja sitä päätettiin käyttää myös uuden version toteutuksessa.

LibIEC61850-kirjasto on rakennettu käyttämään MMS-protokollaa tiedonsiirrossa IED-laitteen ja sen asiakasohjelman välillä IEC 61850 -standardin 8-1 osan mukaan. Kuvassa 11 on esitetty kirjaston kerrosarkkitehtuuri asiakasohjelmalle. Kirjastoon on toteutettu *laiteabstraktiokerros* (*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



Kuva 10. *Ruby:lla toteutetun demoversion arkkitehtuuri ja tiedonsiirto.*

käytettiin olemassa olevaa Linux HAL -toteutusta. Kuvassa 11 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 Ruby:lla toteutettu ohjelmisto.



Kuva 11. *LibIEC61850-kirjaston kerrosarkkitehtuurin komponentit, vihreällä Ruby toteutukseen lisätyt osat (pohjautuu kuvaan [38]).*

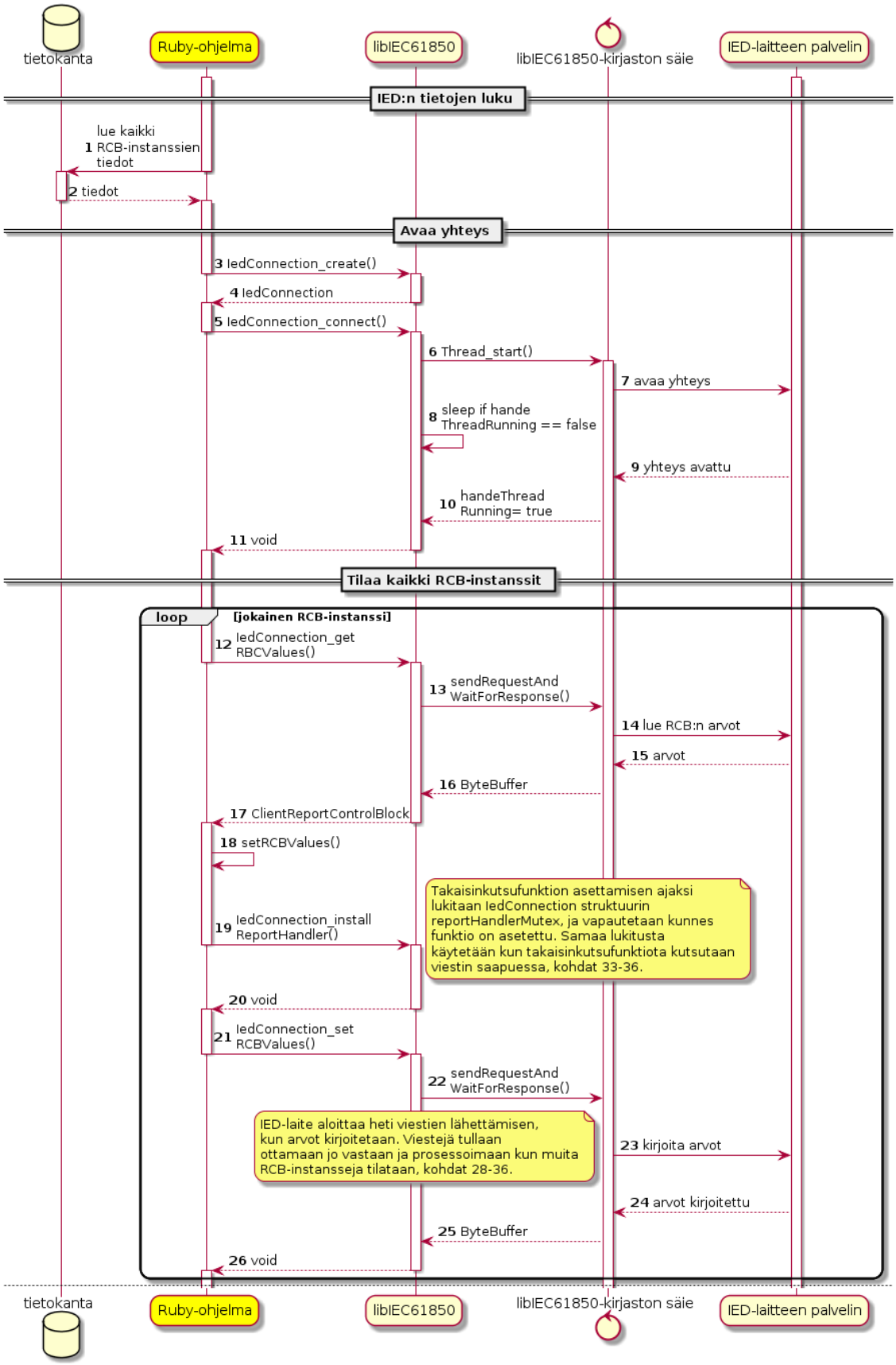
Ruby-koodista C-kielen funktioiden kutsuminen ei ole suoraan mahdollista, vaan kielten väliin täytyy toteuttaa liitos. Demoversiossa liitos oli tehty käyttäen Ruby:lle saatavaa *ruby-ffi*-kirjastoa [60] (*Foreign Function Interface*, lyhennetään *FFI*). Liitoksen avulla Ruby voi kutsua C-kielen funktioita ja käyttää sen struktuureita ja muuttujia. Demossa kirjasto huolehti matalan tason IEC 61850 asiat, ja Ruby-osuus keskittyi liitoksen avulla viestien tilaamiseen, käsittelyyn ja tallentamiseen.

5.2 Toiminta

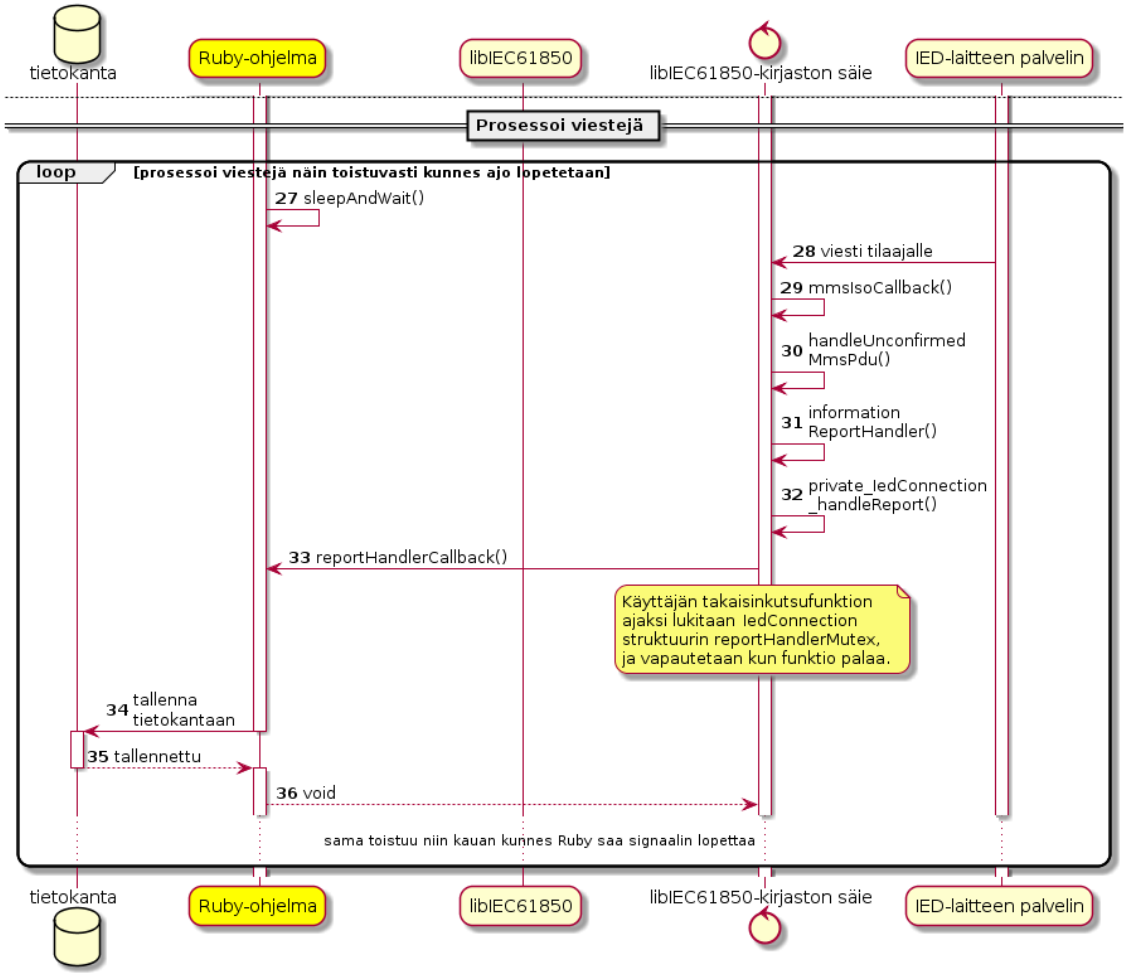
Ohjelman toiminta on esitetty sekvenssikaavioissa 12 ja 13. Sekvenssikaavio 12 jatkuu kuvassa 13. Kuvissa ohjelman kaksi eri silmukkaa on esitetty loop-laatikoilla. Sekvenssikaaviossa osallisena ovat tietokanta, Ruby-ohjelma, libIEC61850-kirjasto, libIEC61850-kirjaston natiivisäie ja IED-laitteen palvelinohjelma. Kirjaston natiivisäie on vastuussa yhteyden ylläpidosta ja datan siirtämisestä IED-laitteen ja kirjastoa käyttävän prosessin välillä. Sekvenssikaavioon on merkitty paksulla suorituksessa olevat palkit, esimerkiksi IED-laitteen palvelinohjelmisto on koko ajan suorituksessa. Kuvissa kaikki kohdat on numeroitu ja näihin viitataan tulevassa tekstissä.

IED-laiteella viestejä mahdollisesti generoidaan ja lähetetään samaan aikaan. Seurauksena niitä voi saapua libIEC61850-kirjastolle ennen kuin edellinen viesti on käsitelty. Tätä varten kirjasto toteuttaa sisäisen puskurin viestien vastaanottamiseen. Puskurista prosessoidaan seuraava viesti, kun edellinen on prosessoitu. Toisin sanoen kirjasto ottaa viestejä vastaan sitä mukaa kun ne saapuvat ja ottaa niitä puskurista yksi kerrallaan saapumisjärjestyksessä. Kirjasto varaa yhden puskurin avattua yhteyttä kohti. Jos ohjelma avaa vain yhden yhteyden, kirjaston käyttäjän ei tarvitse huolehtia rinnakkaisuudesta. [50]

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). 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_getRCBValues()` (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 takaisinkutsufunktiota kutsutaan viestin saapuessa (kohdat 33–36). Tilanteessa, jossa takaisinkutsufunktiota asetetaan samalla kun viesti on saapunut, joutuu toinen osapuoli odottamaan lukituksen vapautumista. Tämän jälkeen arvot kirjoitetaan takaisin IED-laitteelle funktiolla `IedConnection_setRCBValues()` (kohdat



Kuva 12. *Sekvenssikaavio kuinka Ruby-ohjelma avaa yhteydet ja tilaa kaikki IED-laitteen RCB-instanssit (jatkuu kuvassa 13).*



Kuva 13. *Sekvenssikaavio kuinka Ruby-ohjelma prosessoi ja tallentaa viestejä libIEC61850-kirjastoa käyttäen (jatkoa kuvalle 12).*

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, joka 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. [50]

5.3 Ongelmien analyysi

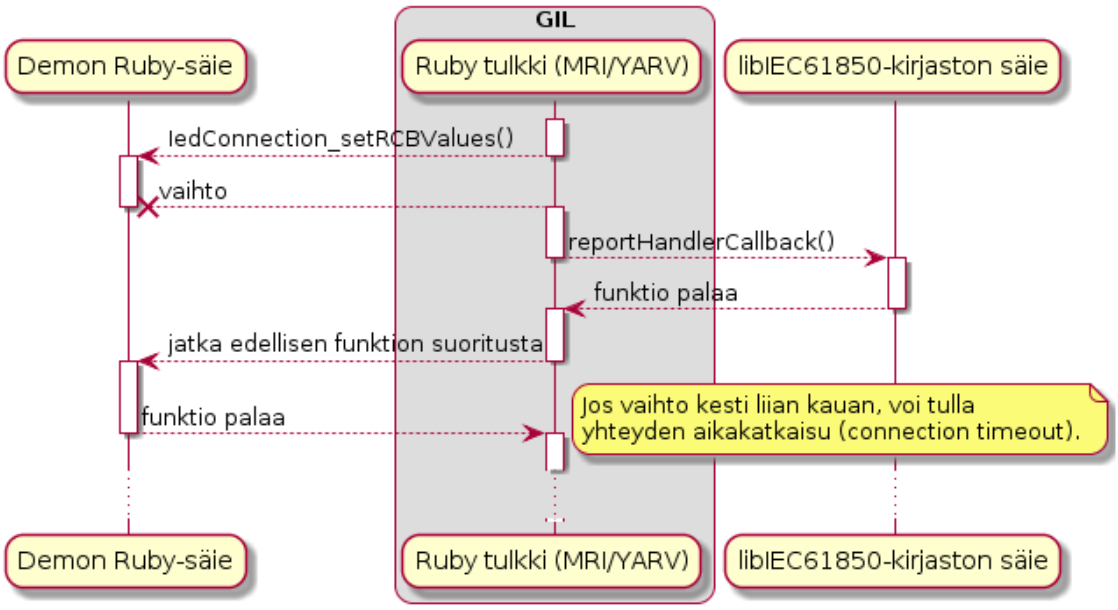
Demo oli toteutettu käyttäen *Ruby on Rails* -kehystä, lyhennetään *RoR*. RoR on tarkoitettu web-sovellusten kehittämiseen Ruby-kielellä. Demo toteutettiin suoritettavaksi RoR-

kehyksen tarjoamalla erillisen Ruby-koodin suorituksen mekanismilla [59]. Mekanismi sallii Ruby-koodin suorituksen RoR:in kontekstissa. Konteksti salli järjestelmän kirjastojen ja koodien käytön demossa. Huonona puolena tässä oli, että yksinkertaisen ohjelman suoritus vaatii järjestelmän kontekstin muistiin lataamisen ennen suoritusta. Tästä seurauksena yksi suoritettu demon prosessi varasi muistia noin 150 Mt, verrattuna uuteen RoR-projektiin, joka vei noin 67 Mt muistia. Ohjelman yksinkertaisuuteen nähden varatun muisti määrä on suuri ja sitä olisi mahdollista pienentää.

Demossa isoimpana ongelmana oli sen huono suorituskyky ja toiminnan epävarmuus RCB-instanssien määrän ollessa esimerkiksi noin 10. 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 noin 30 sekuntia 10 instanssin tilaamiseen. RCB-instanssien määrä vaihtelee IED-laitteen mukaan. Tässä työssä käsitellyt määrät olivat 3–13 instanssia.

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. Ruby:ssa käytettiin sen oletustulkkia *MRI/YARV* (*Matz's Ruby Interpreter*, lyhennetään *MRI* tai *Yet another Ruby VM*, lyhennetään *YARV*). Ruby versiosta 1.9 eteenpäin käyttää YARV-tulkkia, joka lisää kieleen esikäännöstä ennen varsinaista tulkausta. Toinen syy oli Ruby-kielen oletustulkissa oleva *globaali tulkkilukitus* (*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 [49, s. 131–133]. Kuvassa 14 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 13 kohdat 33–36) ja samalla tilataan muita RCB-instansseja (kuvassa 12 kohdat 12–26). Säie joutuu odottamaan lukituksen vapautusta takaisinkutsufunktion asettamisen ajan (kohdat 19–20). Ratkaisuna tähän olisi pitää takaisinkutsufunktio mahdollisimman lyhyenä suoritusajan suhteen. [37, 61]

Edellä mainittujen lisäksi demototeutuksessa oli muistivuoto huonon ohjelmoinnin takia. Muistivuoto on tilanne missä ohjelma varaa lisää muistia ilman, että sitä vapautetaan takaisin käyttöjärjestelmälle. Muistivuoto johtui todennäköisesti ohjelmointivirhees-



Kuva 14. *Ruby-tulkin globaalin lukituksen toiminta, joka vuorottaa ajossa olevia säikeitä riippumatta käyttöjärjestelmän vuorottajasta.*

tä ruby-ffi -kirjastolla. Kun liitos Rubysta tehdään C-kieleen, täytyy ohjelmoijan miettiä varatun muistin vapauttamista. Ruby-ohjelmoijan ei normaalisti tarvitse tästä huolehtia automaattisen roskien keruun ansiosta. Muistivuoto havaittiin kun ohjelma jätettiin suoritukseen pitemmäksi aikaa ja se oli varannut melkein kaiken käyttöjärjestelmän muistista itselleen. Tämän pystyi havaitsemaan helposti ohjelman suorituksen aikana Linuxin htop-ohjelmalla MEM%-sarakkeesta. Sarake kertoo prosessin käyttämän prosentuaalisen osuuden koko käyttöjärjestelmän muistista [55]. Luku kasvoi tasaisesti ja sen verran nopeasti, että sen havaitseminen oli mahdollista ilman pitempää suoritusaikaa.

5.4 Yhteenveto

Demon jatkokehitys toimivaksi olisi vaatinut paljon vaivaa huomioon ottaen tässä kappaleessa käsitellyt ongelmat. Tämän lisäksi sen kehityksessä ei ollut huomioitu järjestelmän hajautuksen vaatimuksia. Tiedon jakaminen muun järjestelmän kanssa tietokannan kautta ei ole hyvä ratkaisu. Se johtaisi tilanteeseen missä järjestelmän komponentit lukisivat viestejä tietokannasta jatkuvasti, ilman tietoa milloin uusi viesti on saapunut. Tästä aiheutuu turhaa kuormaa tietokannalle ja komponentin saama tieto ei välttämättä ole ajan tasalla. Isoilla muutoksilla demo olisi ollut mahdollista saada täyttämään asetetut vaatimukset ja sen ongelmat korjattua. Demo kuitenkin päätettiin korvata kokonaan uudella toteutuksella sen vaatiman työmäärän takia.

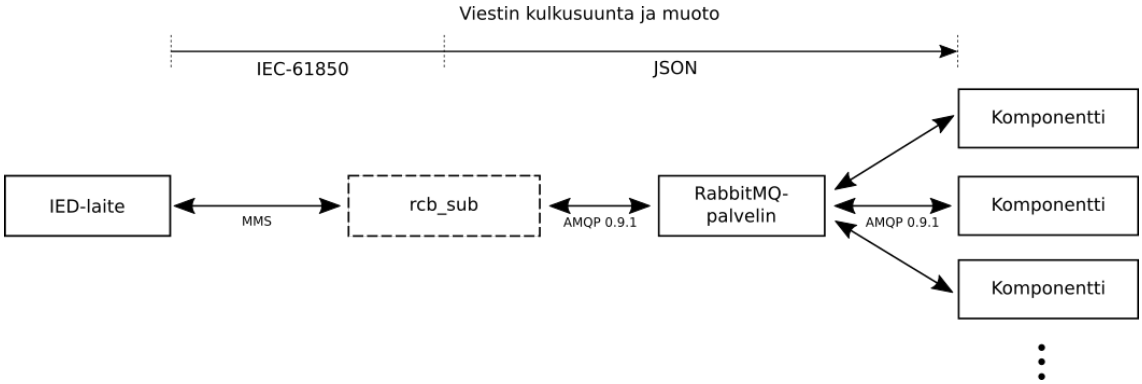
Ohjelmiston huonoon suorituskykyyn ja epävarmuuteen pääasiassa on syynä Ruby-kielen GIL ja suorituskyky tulkattavana kielenä. Varatun muistin määrän oli suuri yksinkertaista ohjelman ajamista varten. Syynä todennäköisesti oli RoR-kehyksen ajoympäristö, joka latasi muistiin muuta järjestelmää ja sen kirjastoja. Ohjelmassa oli muistivuoto, joka

todennäköisesti johtui Ruby:n ja C-kielen liitoksen huonosta ohjelmoinnista. Edellä mainittuja tietoja käytetään esimerkiksi apuna suunnittelussa tekniikoiden valinnassa. Demototeutuksen perusteella libIEC61850-kirjasto todettiin hyväksi ja sitä käytettiin myös uudessa versiossa. Suunnittelussa kysymyksenä jää miettiä mitkä tekniikat valitaan toteutukseen, jotta suorituskykyongelmat vältetään ja varatun muistin koko pidetään kohtuullisena. Muistivuoto ohjelmassa taas vältetään huolellisella ohjelmoinnilla.

6. TUOTANTOVERSION SUUNNITTELU

6.1 Kokonaiskuva

Luvun 4 arkkitehtuurin suunnittelun ja luvun 5 demon analyysin pohjalta päädyttiin kuvassa 15 esitettyyn systeemin arkkitehtuuriin. Kuva tarkentaa aikaisemmin suunniteltua arkkitehtuuria, joka esiteltiin kuvassa 9. AMQP-välittäjäpalvelin päädyttiin toteuttamaan RabbitMQ-ohjelmistolla, joka on AMQP-standardiin perustuva välittäjäohjelmisto [58]. Väliohjelmistolle annettiin nimeksi rcb_sub ja on merkitty kuvaan katkoviivalla. Tätä nimeä käytetään tästä eteenpäin tekstissä viittaamaan kyseiseen komponenttiin.



Kuva 15. *Suunnitellun järjestelmän arkkitehtuuri sekä viestin kulku ja muoto sen osapuolten läpi.*

Kuvassa vasemmalla on IED-laite, josta rcb_sub tilaa viestit MMS-protokollan avulla. Rcb_sub prosessoi saapuneet viestit JSON-muotoon ja uudelleenjulkaisee ne RabbitMQ-palvelimelle. Järjestelmän muut komponentit tilaavat JSON-viestejä välittäjäpalvelimelta tarpeidensa mukaan.

Rcb_sub päädyttiin toteuttamaan C-kielellä komentorivipohjaiseksi ohjelmistoksi. Muu järjestelmä ohjaa komponentin suoritusta ja syöttää tilaukseen tarvittavat tiedot sille komentoriviparametreinä. Rcb_sub pystyi tilaamaan yhden IED-laitteen halutun määrän RCB-instansseja. AMQP-standardista on olemassa eri versioita ja valittu RabbitMQ-ohjelmisto käytti versiota 0.9.1. Rcb_sub käytti demosta tuttua libIEC61850-kirjastoa hoitamaan matalan tason IEC 61850 -standardin toiminnallisuuden.

6.2 AMQP-välittäjäpalvelin

AMQP-pohjaisen välittäjäpalvelimen toteutukseen löytyy erilaisia ohjelmistoja mm. RabbitMQ, Apache Qpid ja StormMQ. Työssä AMQP-pohjaisen palvelimen toteuttamiseen

valittiin RabbitMQ. RabbitMQ on ilmainen avoimen lähdekoodin välittäjäpalvelin ja sille on olemassa kattava tuki monelle eri kielelle asiakasohjelmiston toteuttamiseen [16]. Vertailun perusteella se vaikutti toteutukseen hyvältä vaihtoehdolta.

AMQP-standardista on julkaistu monta eri versiota, ja työn tekohetkellä viimeisin versio oli 1.0. RabbitMQ-ohjelmisto on suunniteltu käytettäväksi standardin version 0.9.1 kanssa, ilman asennettuja lisäosia. Versioiden välinen ero on 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 [57]. Tämä ei kuitenkaan sen käyttöä haitannut, koska versio 0.9.1 kattaa kaikki suunnitellut hajautetun järjestelmän paradigmat. Paradigmoja olivat viestijono ja julkaisija-tilaaja. RabbitMQ:ta voi käyttää AMQP version 1.0 kanssa erillisellä lisäosalla. RabbitMQ lupaa jatkaa version 0.9.1 tukemista, jolloin sitä on myös mahdollista käyttää jatkossakin [57].

6.3 Tilauksen orkestrointi ja tiedon välitys

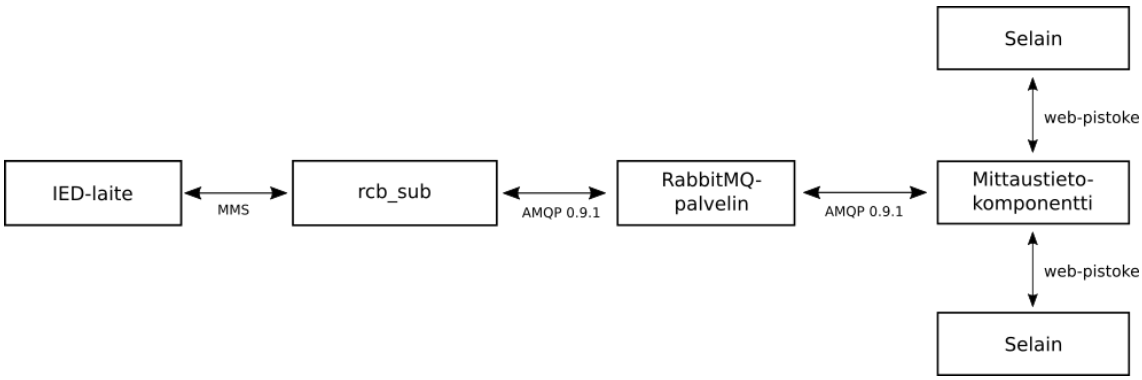
Muun järjestelmän on tarkoitus ohjata rcb_sub-ohjelman suoritusta. IED-laitteilta viestien tilauksia käyttäjä voi ohjata järjestelmän käyttöliittymästä. Tilauksen aloittaessa järjestelmä käynnistää rcb_sub-ohjelmiston omana prosessinaan yhtä IED-laitetta kohti. Tilauksen loputtua järjestelmä lopettaa rcb_sub-prosessin suorituksen. Suorituksen aikana tulevat virheet ohjataan järjestelmälle, joka voi toimia tarvittaessa niiden mukaan, esimerkiksi käynnistää prosessin uudestaan. Kaikkien rcb_sub-prosessien on tarkoitus ohjata JSON-viestit saman RabbitMQ-palvelimen läpi muulle järjestelmälle.

Rcb_sub-ohjelmaa oli tarkoitus ajaa taustaprosessina, jolloin siihen ei tarvittu käyttöliittymää. Tämän takia se päätettiin toteuttaa komentorivipohjaisena ohjelmana. Muu järjestelmä oli rakennettu suoritettavaksi Linux-käyttöjärjestelmän päällä, joten rcb_sub toteutettiin myös samalle käyttöjärjestelmälle. Jotta rcb_sub voi tehdä tilauksen IED-laitteelle ja tietää mitkä RCB-instanssit tilataan, täytyy järjestelmän tarjota tämä tieto. Tarvittavaa tietoa ovat IED-laitteen ja AMQP-palvelimen IP-osoitteet, tilattavien RCB-instanssien viitteet ja arvot, viestien julkaisuun välittäjäpalvelimelle tarvittavat tiedot. RCB-instanssille kirjoitettavat arvot sisältävät vaihtoehtoiset kentät ja liipaisimet. Julkaisuun tarvittavia tietoja ovat käytettävän *vaihteen* (*exchange*) nimi ja *reititysavain* (*routing key*). Vaihde on AMQP-palvelimella käsite, johon tilaajat tekevät tilauksia ja on vastuussa viestien reitittämisestä oikeille tilaajille. Reititysavain on viestin tunniste millä se julkaistaan. Tämän ja tilaajan tekemän tilauksen mukaan vaihde reitittää viestit oikeille tilaajille. Toisin sanoen reititysavain sisältää IED-laitteen tunnisteen. Tämän perusteella tilaaja voi tilata haluamansa IED-laitteen viestit.

Tiedon välittämiseen prosessien välillä on olemassa monia eri tapoja. Jos tieto on pysyvää ja siinä ei ole muutoksia, yksi vaihtoehto olisi ollut konfiguraatiotiedosto. Järjestelmässä kuitenkin tilattavien RCB-instanssien määrä ja IED-laitteen tiedot voivat muuttua. Tämän takia päädyttiin käynnistyksen yhteyteen annettuihin komentoriviparametreihin.

Muu järjestelmän osa, joka rcb_sub-prosessin käynnistää, voi kaiken tarvittavan tiedon antaa prosessille parametreillä käynnistyksen yhteydessä. Vaikka tieto tilauksien välillä muuttuu, prosessi käynnistetään aina viimeisimmillä tiedoilla. Ohjelmalle ei ollut asetettu vaatimusta, että tietoja pystyisi muuttamaan tilauksen aikana. Jos tietoja tarvitsee muuttaa, lopetetaan edellinen tilaus ja käynnistetään uusi prosessi uusilla parametreilla. Sama periaate on myös järjestelmän käyttöliittymässä. Myöhemmin tulevaisuudessa tarpeen vaatiessa voidaan siirtyä dynaamisen tilauksen muutokseen, mutta tällä hetkellä sille ei ollut tarvetta.

AMQP ei tarjoa mahdollisuutta julkaisujen mainostukseen, kuten käsiteltiin julkaisija-tilaaja-paradigman yhteydessä [1]. Järjestelmän komponenttien pitää saada tieto olemassa olevista julkaisijoista muulta järjestelmältä. Tämä tieto järjestelmässä siirretään komponenteille tietokannan kautta. Kuvassa 16 on esitetty käyttötapausesimerkki, jossa mittaustietoa käyttävä komponentti tilaa viestejä ja lähettää sen käyttäjän selaimen käyttöliittymään web-pistoketta pitkin [19]. Selaimessa JavaScript-koodi päivittää käyttöliittymän komponentteja saadun tiedon mukaan.



Kuva 16. *Esimerkkikäyttötapaus, jossa mittaustietoa tilaava komponentti lähettää tietoa selaimen käyttöliittymään web-pistokkeen avulla.*

6.4 Suorituskyky ja kielen valinta

Ennen koko ohjelman uudelleenkirjoitusta, kokeiltiin demoa korjata vaihtamalla Ruby-tulkkia. Ruby:n oletustulkki yritettiin vaihtaa *JRuby*-tulkkiin [33]. Tavoitteena vaihdossa oli saada Ruby-ohjelma toimimaan ilman globaalia tulkkilukitusta (GIL). JRuby on Ruby-tulkki, joka suorittaa Ruby-koodia *Java-virtuaalikoneen* (*Java Virtual Machine*, lyhennetään *JVM*) päällä. JRuby mahdollistaa säikeiden suorituksen rinnakkain JVM:n omilla säikeillä ja näin ollen suorituksen pitäisi olla nopeampaa [64]. Aidolla rinnakkaisuudella ohjelman suoritus ei olisi pysähtynyt viestin saapuessa takaisinkutsufunktion suorituksen ajaksi. Tämä ei vielä olisi kuitenkaan ratkaissut kaikkia ohjelmassa olevia ongelmia, kuten muistivuotoa ja hitaampaa suorituskykyä verrattuna käännettävään kieleen. Tämä toteutus ei kuitenkaan toiminut, ja yrityksen jälkeen päätettiin palata suunnitelmaan kirjoittaa koko ohjelma uudestaan. JRuby ei tukenut kaikkia projektin käyttämiä kirjastoja.

Seurauksena olisi ollut saman projektin ylläpitäminen kahdelle eri tulkille tai asennettavien kirjastojen erottaminen. Kaikkiaan oli helpompaa kirjoittaa ohjelma alusta toisella tekniikalla.

Uuden toteutuksen kieleksi valittiin C-kieli. Isona syynä kielen valintaan oli sen suorituskyky. C-koodi käännetään alustalle suoraan konekäskyiksi, joiden suoritus on nopeampaa kuin tulkattavan kielen, kuten Ruby ja Python. Valintaan vaikutti myös tekijän iso mieltymys matalan tason ohjelmointiin ja C-kieleen. Kielen valinnan yhteydessä varmistettiin 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 toteutettu C-kielellä.

6.5 JSON-viestin rakenne

IED-laitteelta saapuva viesti päädyttiin muuntamaan JSON-muotoon helpompaa luettavuutta varten. Liitteessä A on esitetty viestin rakenne. JSON:n noudattaa pääosin standardin mukaista viestin rakennetta. Erona standardin malliin on, että muuttujiin päätettiin lisätä viite, tyyppi ja koko bitteinä. Nämä tiedot todettiin tarpeellisiksi, koska niiden avulla tilaajan on helpompi ymmärtää viestin sisältöä. Tarvittavat lisätiedot luetaan IED-laitteelta erillisellä palvelukutsulla ennen tilauksen aloittamista. Nämä tiedot yhdistetään saapuneen viestin kanssa ja sijoitetaan JSON-rakenteeseen.

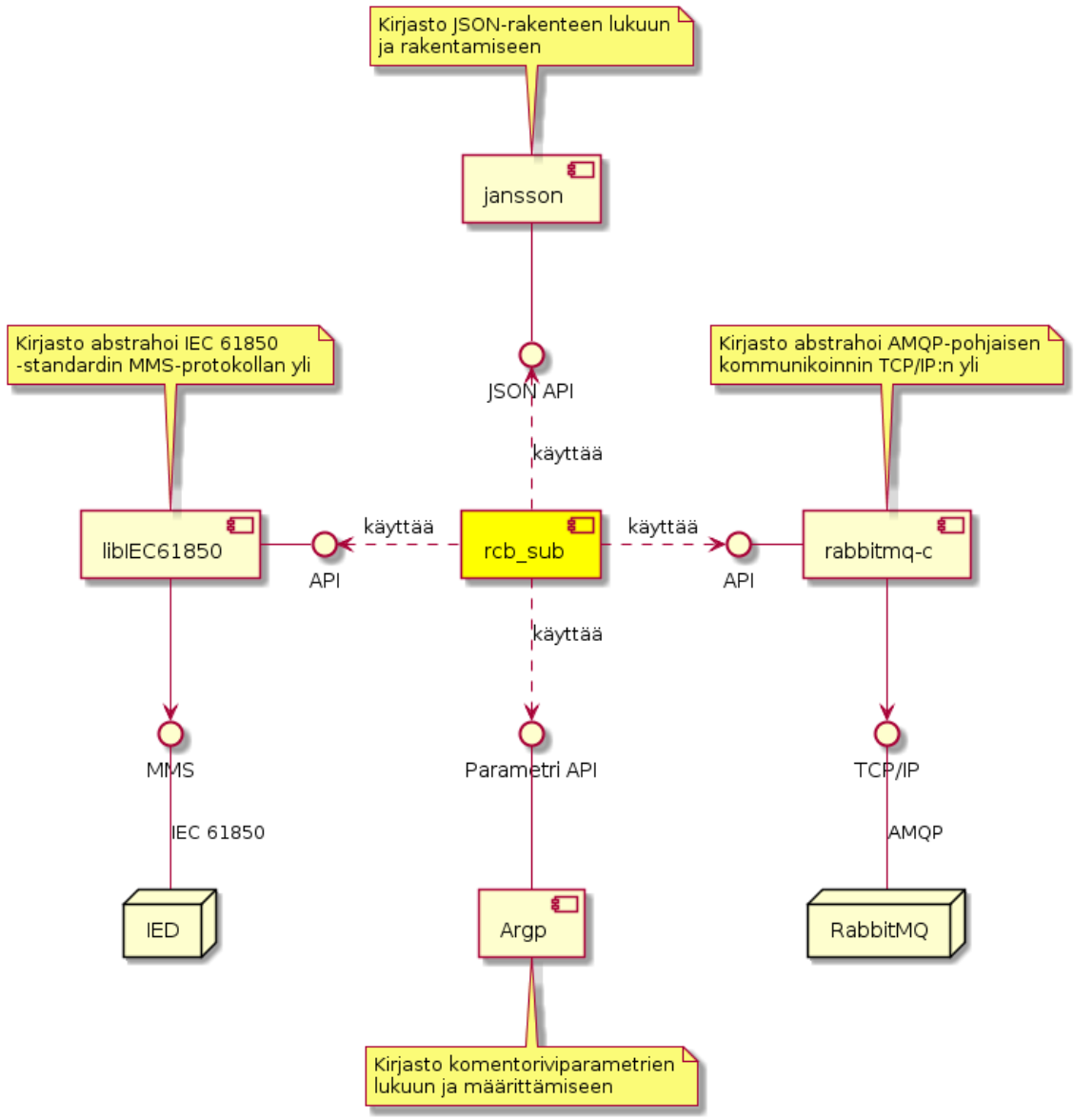
Standardin mukainen viesti sisältää vaihtoehtoisia kenttiä, joita tilaaja voi konfiguroida RCB-instanssille ennen tilauksen aloittamista. Tällä tilaaja voi poistaa viestistä tarpeettomaksi koettua tietoa. Kuitenkin JSON-viestiin haluttiin lisätä kaikki mahdolliset kentät selkeyden takia. Jos kenttä puuttui IED-laitteelta saapuneesta viestistä, asetettiin sen arvoksi JSON-viestiin null-arvo. Esimerkinä tästä on liiteen A rivillä 4 oleva *confRevision*-kenttä, jonka arvoksi on asetettu null.

Lisätyt kentät sisältävät joitakin poikkeuksia. Kokoa bitteinä ei ole lisätty *boolean* ja *utc-time* tyyppisille muuttujille, koska tätä tietoa ei saa IED-laitteelta. *Bit-string* tyypille lisättiin kaksi arvo-kenttää *valueLittleEndian* ja *valueBigEndian* yhden sijaan, koska se on mahdollista lukea eri bittijärjestyksellä (little ja big endian). Aikayksiköt päätettiin antaa suoraan samassa formaatissa kuin alkuperäisessä viestissä. Viestin päätason aikaleima (rivi 5) on millisekunteja *UNIX*-ajanlaskun alusta 1. tammikuuta 1970 klo 00:00:00 UTC (epoch) tähän hetkeen. Muissa muuttujissa tyypiltään *utc-time*, luku on sekunteja samasta UNIX-ajanlaskusta tähän hetkeen [28, s. 26–27].

7. TOTEUTUS

7.1 Yleiskuva

Kuvassa 17 on esitetty komponenttikaavio rcb_sub-ohjelmasta ja sen käyttämistä kirjastoista. Kuvasta voi nähdä miten eri komponentit ovat relaatiossa keskenään ohjelman kanssa ja mitkä osat kommunikoivat IED-laitteen ja RabbitMQ-palvelimen kanssa.



Kuva 17. *Rcb_sub-ohjelman komponenttikaavio.*

Toteutukseen valittiin seuraavat kirjastot:

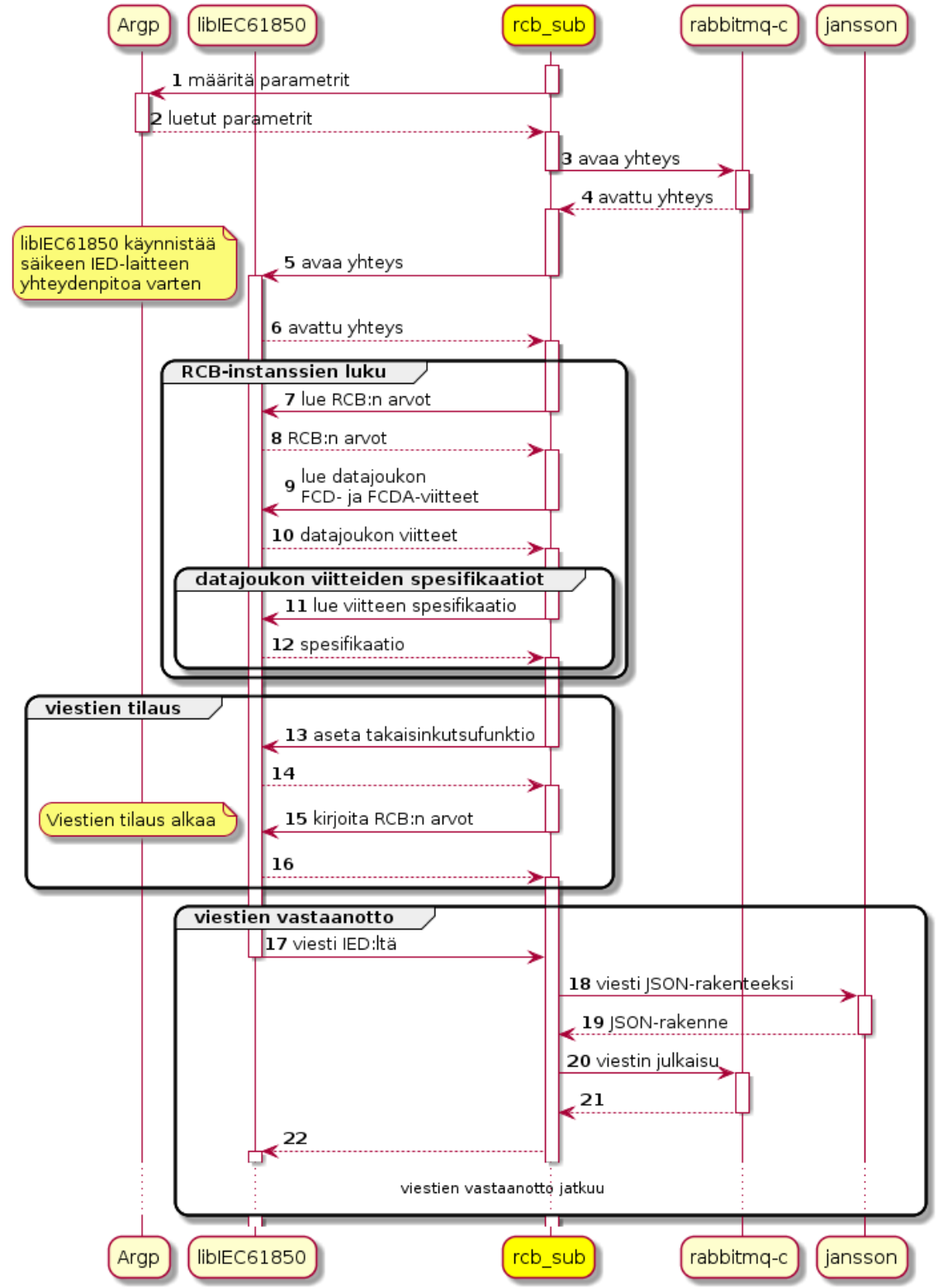
- *libIEC61850* [50],
- *rabbitmq-c* [56],
- *jansson* [12], ja
- *Argp* [51].

Kaikki käytetyt kirjastot ovat toteutettu C-kielelle. Kirjastojen tarkoitus on abstrahoida tietyn asian käyttö, ja tarjota käyttäjälle siitä helppokäyttöinen ja ymmärrettävä rajapinta. LibIEC61850-kirjasto abstrahoi IEC 61850 -standardin käyttöä ja hoitaa matalan tason MMS-protokollan kommunikoinnin [50]. Samaa kirjastoa käytettiin myös demoversiossa ja kirjaston kerrosarkkitehtuuri esitettiin aikaisemmin kuvassa 11. Kuvassa 17 libIEC61850 kommunikoi suoraan IED-laitteen kanssa MMS-protokollaa käyttäen. Rabbitmq-c-kirjasto abstrahoi RabbitMQ-palvelimen käytön ja hoitaa matalan tason AMQP-pohjaisen kommunikoinnin [56]. Toteutuksessa rabbitmq-c kommunikoi suoraan RabbitMQ-palvelimen kanssa. Jansson-kirjasto abstrahoi JSON-rakenteiden lukua ja käsittelyä C-kielelle [12]. Kirjastoa käytettiin muuntamaan IEC 61850 -standardin viesti JSON-muotoon. JSON-rakenne on nähtävissä liitteessä A. Argp-kirjasto auttaa ohjelman komentoriviparametrien määrittämisessä ja käsittelyssä [51]. Argp-kirjastolla voidaan toteuttaa *UNIX*-tyyliset *parametrit* (*arguments*) ja *valitsimet/vivut* (*options/switches*) [4].

Kuvassa 18 on esitetty rcb_sub-ohjelman sekvenssikaavio pääpiirteisestä toiminnasta. Ohjelman noudattaa osin demon toimintaperiaatetta (kuvat 12 ja 13). Seuraavaksi käydään läpi ohjelman pääpiirteinen toiminta ja myöhemmin jokainen kohta tarkemmin läpi kappaleessa 7.2. Ohjelman suoritus alkaa lukemalla annetut parametrit ja vivut 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 halutaan 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, jotka lisätään JSON:iin myöhemmin. 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. 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).

7.2 Ohjelman toiminta

Tulevissa kappaleissa käydään läpi tarkemmin rcb_sub-ohjelman toimintaa. Kappaleiden järjestys noudattaa kuvassa 18 olevan sekvenssikaavion järjestystä ja jokainen kappale



Kuva 18. *Sekvenssikaavio rcb_sub-ohjelman kokonaistoiminnasta.*

tarkentaa tiettyä osaa siitä. Toisin sanoen ohjelmaa käydään tarkemmin läpi sen suorituksen järjestyksessä.

7.2.1 Parametrisointi

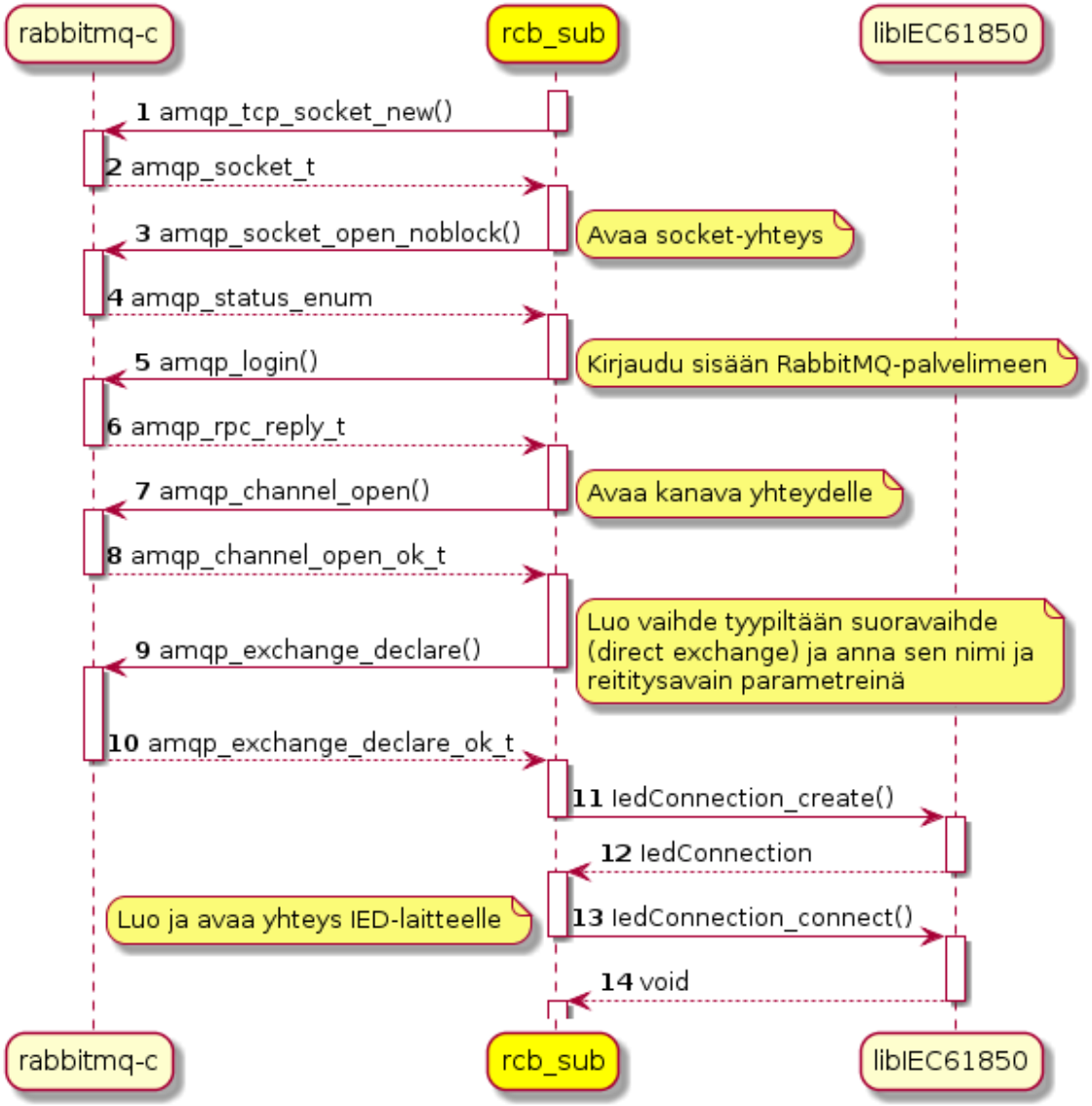
Ohjelma parametrisoitiin Argp-kirjastolla. Kirjasto tarjoaa rajapinnan komentoriviparametrien käsittelyyn ja määrittämiseen. Parametrien muodot ovat tutut muista Linux-käyttöjärjestelmän parametreista ja samaa periaatetta käytettiin tässäkin ohjelmassa. Kirjasto myös lisäsi ohjelmaan automaattisesti aputekstin käyttäjää varten. Aputeksti sisältää tietoa ohjelman parametreista ja niiden käytöstä. Aputekstin pystyi tulostamaan vivulla `--help`. Liitteessä B on esitetty ohjelman tulostama aputeksti. Liitteestä voi myös nähdä kaikki ohjelman parametrit ja lyhyen selityksen mihin kutakin käytetään. Aputeksti ei sinänsä ollut tarpeellinen, koska muu järjestelmä hallitsee ohjelman suoritusta ja parametrien antamista. Se kuitenkin päätettiin lisätä pienen vaivan vuoksi ja toimii hyvänä dokumentaationa myöhemmin.

Ohjelmiston parametrien ja vipujen voidaan ajatella koostuvan kolmesta eri ryhmästä (liite B rivit 1–4). Ensin päätason vaihtoehtoiset vivut `OPTIONS` (rivi 1). Pakolliset parametrit `EXCHANGE` ja `ROUTING_KEY` (rivi 2). Viimeisenä ryhmänä `RCB_REF` parametri ja siihen liittyvät vivut `RCB_OPTIONS` (rivi 3). Näitä ryhmiä voi olla n-kappaletta, mutta vähintään yksi. Liitteessä B riveillä 71–72 on esitetty esimerkki, joka tilaa viestit IED-laitteelta osoitteesta 192.168.2.220. AMQP-vaihteen nimi on *testexchange* ja reititysavaimen nimi on *testkey*. IED-laitteelta tilataan RCB-instanssi viitteellä MY_LD0/LLN0 .BR.rcbMeas01. Instanssille asetetaan yleinen kysely (`-g1`), liipaisimet (`-t27`) ja viestin vaihtoehtoiset kentät (`-o16`). Liipaisimet ja vaihtoehtoiset kentät annetaan numeroarvoilla summaamalla niitä yhteen. Vaihtoehdot näkee ohjelman aputekstistä (esimerkiksi liipaisimet riveillä 53–58).

Suurin osa `OPTION` vivuista ovat 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. Parametri `RCB_REF` määrittää viitteen tilattavaan RCB-instanssiin IED-laitteella. Tätä seuraa vaihtoehtoinen `RCB_OPTIONS` vipu, joka määrittää edeltävän instanssin kirjoitettavat arvot ennen tilausta. Sillä voidaan määrittää mm. käytetyt vaihtoehtoiset kentät (`--opt-fields`), käytetyt liipaisimet (`--trigger`).

7.2.2 Yhteyksien muodostus

Parametrien luvun jälkeen ohjelma muodostaa yhteydet ensin RabbitMQ-palvelimelle ja sen jälkeen IED-laitteelle. Kuvassa 19 on esitetty sekvenssikaavio, joka näyttää mitä kirjaston funktioita ohjelma kutsuu missäkin järjestyksessä. Funktiot ja niiden parametrit voi tarkemmin tarkistaa kirjastojen omista dokumentaatioista [39] [56]. Tämä tarkentaa yleiskuvasta 18 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.



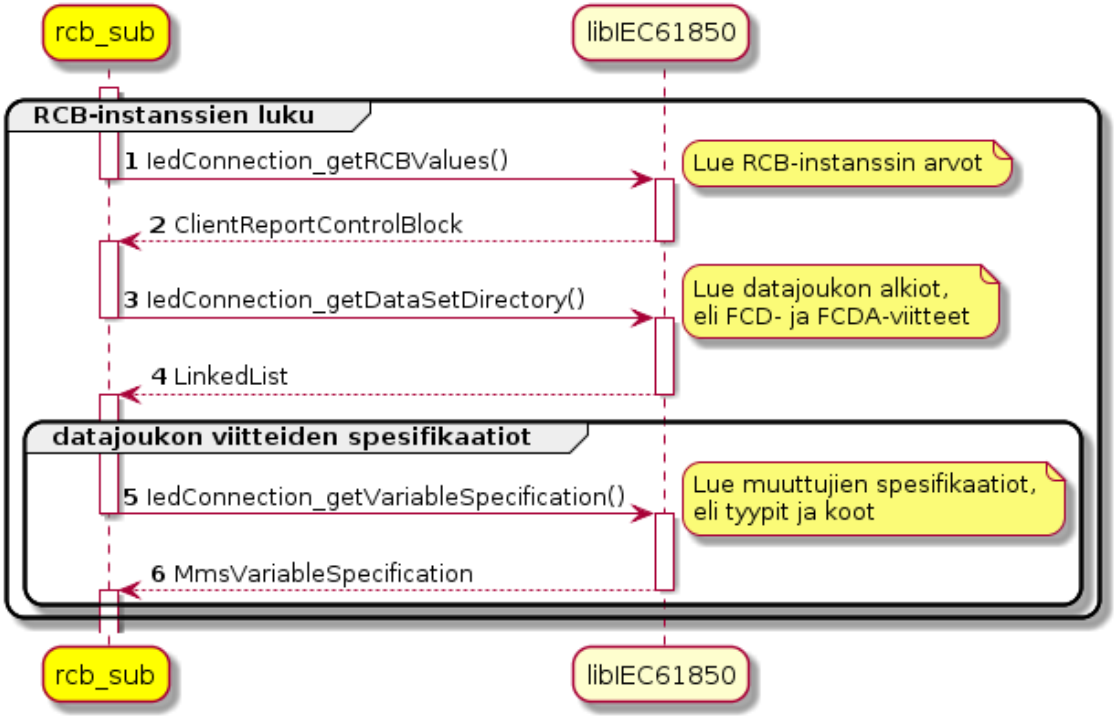
Kuva 19. *Sekvenssikaavio kuinka rcb_sub avaa yhteydet RabbitMQ-palvelimelle ja IED-laitteelle.*

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 sen, että sama yhteys voidaan jakaa monen säikeen kanssa.

7.2.3 IED:n muuttujien tietojen luku

Yhteyksien muodostamisen jälkeen ohjelma käy läpi silmukassa jokaisen parametrina annetun RCB:n viitteen. Lukee RCB:n datajoukon viitteet ja selvittää sen jokaisen muuttujan spesifikaatiot. Kuvassa 20 on esitetty sekvenssikaavio toiminnasta. Kuva tarkentaa yleiskuvassa 18 kohtia 7–12.

Ensin RCB:sta luetaan sen tiedot IED-laitteelta. RCB:ltä saadaan tieto mihin datajouk-



Kuva 20. *Sekvenssikaavio kuinka rcb_sub lukee RCB-instanssin arvot ja muuttujien spesifikaatiot.*

koon se on liitetty. Tämän jälkeen selvitetään kaikki datajoukossa olevien muuttujien spesifikaatio. Luetut tiedot tallennetaan ja niitä käytetään täydentämään muuttujien tietoja JSON-viestiin. Jokaiseen muuttujaan lisätään sen viite, tyyppi ja koko bitteinä. Esimerkkinä tästä on liitteessä A riveillä 21–22.

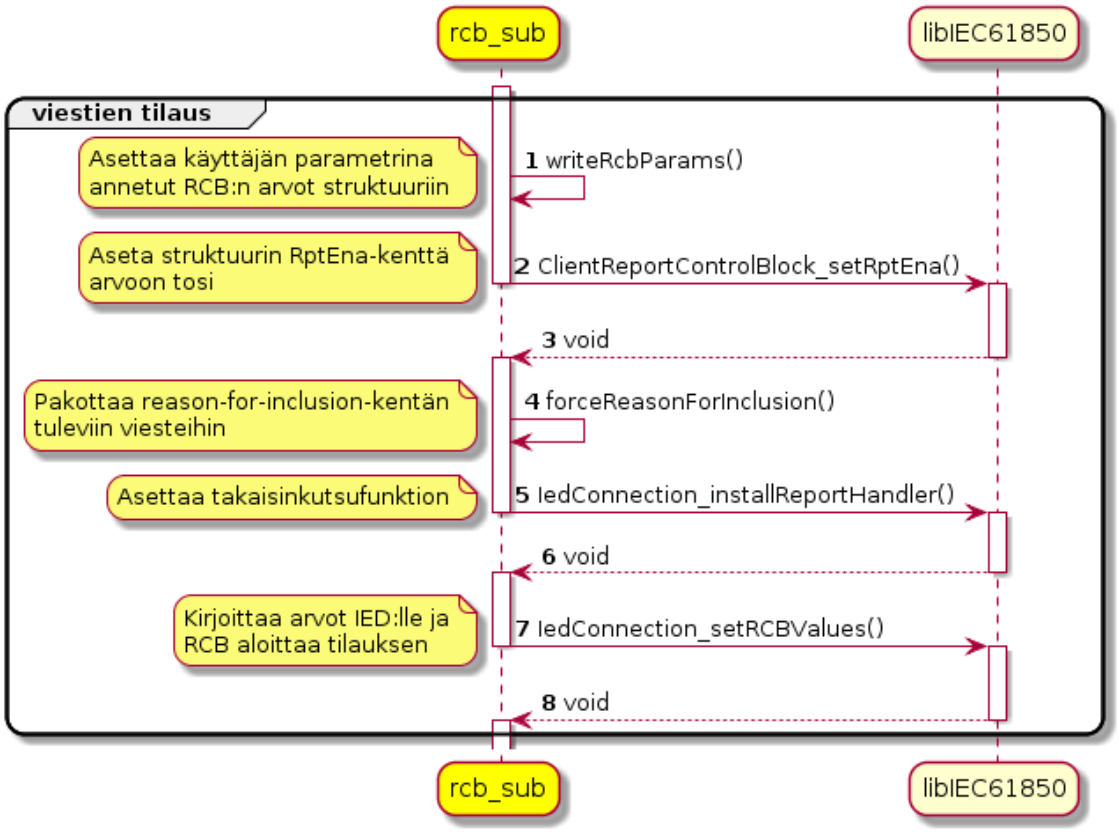
7.2.4 Viestien tilaus

Ohjelman luettua kaikki muuttujien spesifikaatiot. Ohjelma tilaa silmukassa parametrina annetut RCB-instanssit. Kuvassa 21 on esitetty sekvenssikaavio toiminnasta. Kuva tarkentaa yleiskuvassa 18 kohtia 13–16.

Jokainen IED-laitteen RCB-instanssi konfiguroidaan ja varataan ennen tilausta. Ohjelma kirjoittaa parametrina annetut liipaisimet ja vaihtoehtoiset kentät instanssiin. Jokaiselle instanssille täytyy myös asettaa takaisinkutsufunktio, jota kirjasto tulee kutsumaan viestin saapuessa. RCB-instanssi aloittaa tilauksen heti kun se varataan. Seurauksena on, että ohjelma tulee käsittelemään viestejä samalla, kun muita RCB-instansseja tilataan.

7.2.5 JSON:in muodostaminen ja julkaisu

Viestin saapuessa libIEC61580-kirjasto kutsuu asetettua takaisinkutsufunktiota. Takaisinkutsufunktio muuttaa viestin JSON-muotoon ja lisäsi siihen aikaisemmin luetut muuttujien viitteet, tyypit ja koot. Tämän jälkeen JSON-julkaistiin RabbitMQ-palvelimelle. Kuvissa 22 ja 23 on esitetty sekvenssikaaviolla, kuinka ohjelma muuttaa viestin JSON:iksi

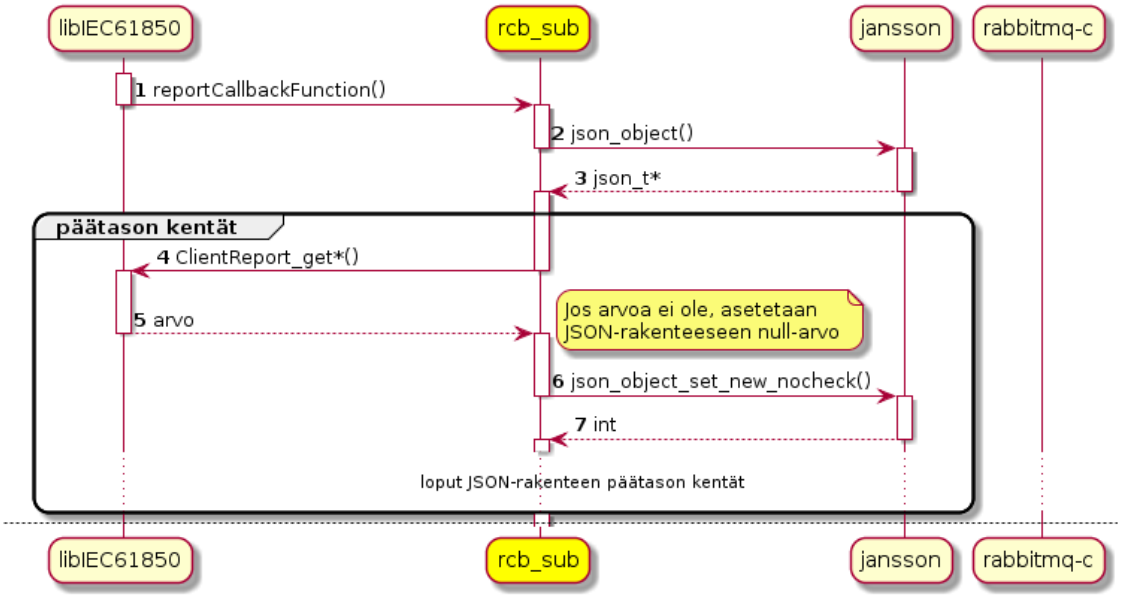


Kuva 21. *Sekvenssikaavio kuinka rcb_sub tilaa RCB-instanssit.*

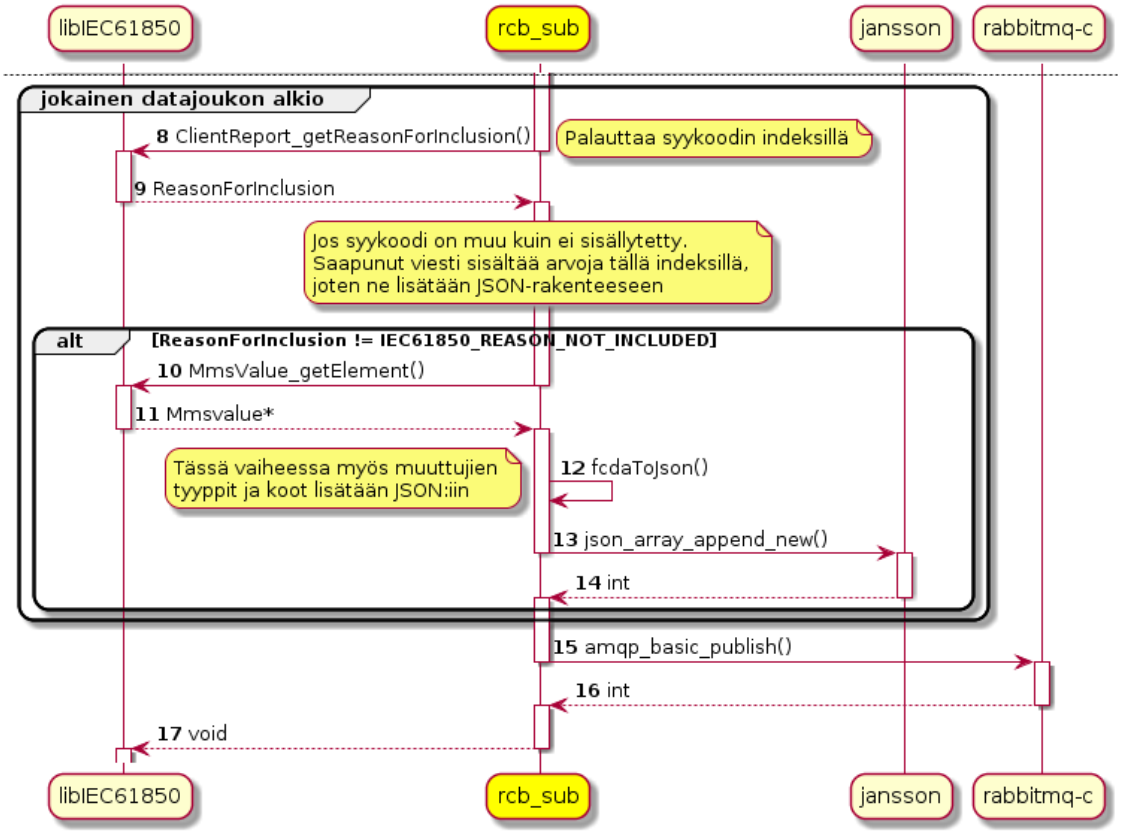
ja julkaisee RabbitMQ:lle. Kuva 22 jatkuu kuvassa 23. Kuva 22 tarkentaa yleiskuvan 18 kohtia 17–19 ja kuva 23 kohtia 20–22. Aikaisemmin mainittiin, että libIEC61850-kirjasto toteuttaa sisäisen puskurin viestien vastaanottoon ja käsittelee siitä yhden viestin kerrallaan. Kirjasto varaa yhden puskurin avattua yhteyttä kohti. Puskurista käsitellään seuraava viesti, kun edellinen takaisinkutsufunktion suoritus on palannut. Rcb_sub avaa vain yhden yhteyden IED-laitteeseen. Seurauksena on, että viestejä ei prosessoida rinnakkain missään vaiheessa suoritusta.

Kuvassa 22 suoritus alkaa, kun libIEC61850-kirjasto kutsuu takaisinkutsufunktiota viestin saapuessa. Funktiolle annetaan parametrina saapunut viesti *ClientReport*-struktuurin instanssina. 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` vivun parametrilla. Jos arvoa viestissä ei ole, korvataan se null-arvolla JSON:iin. Tämän jälkeen suoritus jatkuu kuvasta 22 kuvaan 23.

Päätason viestin kenttien jälkeen ohjelma käy läpi silmukassa viestiin sisällytetyt muuttujien arvot. Tässä vaiheessa ohjelma yhdistää aikaisemmin luetut muuttujien spesifikaatiot viestin muuttujiin. IEC 61850 -standardin mukainen viesti sisältää vain taulukon muuttujien arvoja. Jotta muuttujan arvon ja aikaisemmin luetun spesifikaation yhdistäminen on mahdollista. Ohjelma pakottaa vaihtoehtoisen syykoodi-kentän päälle viestiin. Tämän avulla saadaan saadaan selville tarvittava muuttujan indeksi, joka mahdollistaa tiedon yh-



Kuva 22. *Sekvenssikaavio kuinka rcb_sub muodostaa JSON:nin päätason kentät.*



Kuva 23. *Sekvenssikaavio kuinka rcb_sub lisää JSON:iin muuttujat viestistä.*

distämisen. Kokonaan muutettu viesti julkaistaan RabbitMQ:lle ja takaisinkutsufunktio palaa.

8. TULOSTEN ARVIOINTI JA POHDINTA

Diplomityössä toteutettu ohjelma ei ole ollut tuotannossa osana muuta järjestelmää vielä kovin kauan. Kuitenkin tähän mennessä se on toiminut ongelmitta. Varmasti ei voida arvioida, että ohjelmassa ei tulisi ongelmia tulevaisuudessa, mutta ainakin alun perusteella tulokset näyttävät toimivilta. Tältä osin voidaan arvioida, että diplomityö pääsi asetettuihin tavoitteisiin onnistuneesti ja halutut vaatimukset saatiin täytettyä. Kuitenkin toimivuudesta huolimatta toteutuksessa on kohtia mitä voitaisiin parantaa, tehdä toisin ja jatkokehittää. Nämä ovat kuitenkin tulevaisuudessa yrityksen sisällä tehtäviä työtehtäviä tai mahdollisesti toisen diplomityön aiheita.

Työn aikana järjestelmän hajautukseen pohdittiin eri paradigmojen sopivuutta ja huomattiin, että siihen sopisivat julkaisija-tilaaja-, joukkokommunikointi- ja viestijono-paradigmat. Toteutukseen valittiin AMQP-standardi, joka mahdollisti julkaisija-tilaaja- ja viestijono-paradigmat, mutta ei suoraan ollut tarkoitettu joukkokommunikointiin. Tämän takia joukkokommunikointi jätettiin pois ja korvattiin julkaisija-tilaaja-paradigmalla. Kuinka hyvin joukkokommunikointi olisi sopinut toteutukseen ei ole tarkkaa tietoa. Kuitenkin julkaisija-tilaajan-paradigma jatkaa IEC 61850 -standardin määrittämää julkaisija-tilaaja-kommunikointia IED-laitteen kanssa ja näin ollen sopii toteutukseen. Toteutukseen myös harkittiin MQTT-standardia AMQP:n sijaan, joka on pelkästään julkaisija-tilaaja-kommunikointiin tarkoitettu protokolla. AMQP kuitenkin valittiin tekijän aikaisemman kokemuksen ja muiden yrityksessä olevien henkilöiden keskustelun pohjalta. Koska joukkokommunikointi jätettiin pois toteutuksesta, olisi MQTT voinut sopia toteutukseen paremmin kuin AMQP sen keveyden takia.

IED-laitteelta tuleva viesti päätettiin muuntaa JSON-muotoon XML:än sijaan. Vertailua kahden välillä tehtiin ja päätös oli aikaisemmin tutkimuksen perusteella selvä. JSON-muoto on kevyempi kuin XML ja sopii viestin muotona hajautettuun järjestelmään. Suunniteltu JSON-rakenne on toiminut käytön ajan tarpeiden mukaan. Kuitenkin siinä oli kohtia mitä pystyisi toteuttamaan toisin. Esimerkiksi *bit-string* tyypin bittijärjestys (endian) voi vaihdella muuttujien välillä ja tämän takia siitä JSON-viestiin julkaistiin kaksi eri arvoa *valueLittleEndian* ja *valueBigEndian* (liite A rivit 23–24). Tällä vastuu muuttujan oikein lukemisesta siirretään tilaajalle. Standardissa kuitenkin on määritetty, kuinka päin muuttuja esitetään, jos se on tyyppiä bit-string. Tämän vastuun voisi siis siirtää rcb_sub-ohjelman puolelle ja tarjota JSON-viestissä pelkkä *value*-kenttä, niin kuin kaikille muillekin muuttujille. Tämä lisäisi JSON-rakenteen yhtäläisyyttä ja helpottaisi sen lukua. Lisäksi aikatyypit *utc-time* JSON-viestissä päätettiin antaa siinä muodossa missä ne tulevat IED-laitteelta, eli millisekunteja UNIX-ajanlaskusta (epoch). JSON ei määritä käytettävää aikaformaattia, mutta JSON-rajapintoihin suositellaan käytettäväksi ISO 8601 -standardin aikaformaattia [34].

Ennen tuotantoversion toteutusta demon ongelmia analysoitiin. Saatujen tuloksien pohjalta toteutuksen ohjelmointikieleksi valittiin C-kieli sen suorituskyvyn takia. Todennäköisesti suurin syy demon huonoon suorityskykyyn oli Ruby-oletustulkin GIL, joka rajoittaa yhden säikeen suorituksen kerrallaan ja estää rinnakkaisuuden. Tämän lisäksi kuormaa aiheutti IED-laitteelta tulevien viestien määrä ja libIEC61850-kirjaston lukitus funktiokutsuissa. C-kielen valinta oli hyvä ratkaisu. Ohjelman aika kaikkien RCB-instanssien tilaamiseen saatiin alas noin 30 sekunnista alle 15 sekuntiin. Suurin osa ajasta tulee IED-laitteelle tehtävien kutsujen määrästä ja niiden odottamisesta. Demon muistinkäyttö Ruby on Rails -ympäristössä oli noin 150 Mt. Rcb_sub:in muistin käyttö saatiin noin 4 kt, joka on todella iso muutos aikaisempaan nähden. Tekniikan valinnan suhteen päätökset onnistuivat hyvin.

Ohjelman kehityksen aikana noudatettiin C-ohjelmoinnin ohjeistoa. Tarkoituksena välttää sen yleisimpiä virheitä, esimerkiksi tekstin formatointihyökkäys [20] ja muistin ylivuoto [10]. Tähän käytettiin apuna GCC-kääntäjän vipuja esimerkiksi `-Wall` ja `-Wextra` [21]. Huolellisesta ohjelmoinnista huolimatta järjestelmään tulee tietoa ulkopuoliselta IED-laitteelta, joka voi sisältää vahingollista tietoa. Tämä osuus jätettiin pois, koska se ei kuulunut tämän diplomityön aiheen piiriin. Ohjelman tietoturva kuitenkin täytyy tarkistaa läpi tulevaisuudessa.

Järjestelmä aloittaa tilauksen käynnistämällä yhden rcb_sub-prosessin per IED-laite. Tieto prosessille annettaan komentoriviparametreilla. Tilauksen muuttuessa, prosessi täytyy käynnistää uudelleen. Työn tekohetkellä ratkaisu sopi tarkoituksiin hyvin. Ratkaisuna olisi myös voinut toteuttaa yhden rcb_sub-prosessin, joka pystyisi tilaamaan monta eri IED-laitetta rinnakkain, tai yhden IED-laitteen tilausta voisi muuttaa ilman, että prosessia täytyy käynnistää uudelleen. Tähän toteutustapaan tiedonsiirto komentoriviparametreilla ei enää onnistuisi vaan tarvittaisiin jokin muu kommunikointitapa. Tähän sopisi aikaisemmin käsitellyt prosessien väliset kommunikointiparadigmat, esimerkiksi pistokkeet. Toteutuksessa monen eri rcb_sub-prosessin tilaamat viestit ohjataan saman RabbitMQ-palvelimen kautta eri reititysavaimilla. Järjestelmän skaalautuessa isommaksi joutuu RabbitMQ isomman kuorman alle, joka todennäköisesti muodostuu pullonkaulaksi. Tarkkoja rajoja tähän ei vielä tiedetä ja nykyinen keskitetty RabbitMQ-palvelin todettiin riittäväksi tarkoituksiin tällä hetkellä. Kuitenkin tulevaisuudessa tämä on asia mikä täytyy ottaa huomioon.

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 yksikkötestien kirjoittamiseen. Ympäristön pystytys vaatii erillisen kirjaston projektin yhteyteen, jolla yksikkötestit kirjoitetaan. Yksikkötestit ovat tärkeä osa ohjelman ylläpitoa ja toiminnan varmistamista muutosten jälkeen. Testiympäristön ja testien toteuttaminen jäi tulevaisuuden kehitystyöksi.

Tässä diplomityössä suunniteltu arkkitehtuuri ja ohjelmistoratkaisut voivat toimia muis-

sakin saman tyylisissä järjestelmissä, missä kommunikoidaan IEC 61850 -standardin mukaisesti ja tietoa julkaistaan eteenpäin. Periaatteessa suunnitelmasta olisi mahdollista toteuttaa oma kokonaisuutensa, jota olisi mahdollistaa käyttää muunkin järjestelmän kanssa. Tämä kuitenkin vaatisi tarkempaa suunnittelua ja edellä pohdittujen vaihtoehtojen käsittelyä.

9. YHTEENVETO

Diplomityön tuloksena saatiin ohjelmistokomponentti osaksi isompaa sähköasemiin liittyvää järjestelmää. Komponentti kykeni tilaamaan viestejä IED-laitteelta IEC 61850 -standardin mukaisesti, muuntamaan viestit JSON-muotoon ja jakamaan sen muun järjestelmän kanssa. Viestien jako järjestelmässä toteutettiin AMQP-standardiin pohjautuvalla välittäjäpalvelimella, joka käyttää julkaisija-tilaaja- ja viestijono-kommunikointiparadigmoja. Toteutetun systeemin arkkitehtuuri esitettiin kuvassa 15. Arkkitehtuurissa muu järjestelmä on vastuussa tilauksien orkestroinnissa ja rcb_sub-prosessien suorituksesta.

Työssä ratkaisua lähdettiin etsimään tarkastelemalla ensin hajautetun järjestelmän teoriaa, sen kommunikointiparadigmoja ja IEC 61850 -standardin määrityksiä. Saatuja tietoja apuna käyttäen suunniteltiin arkkitehtuuri osaksi muuta järjestelmää. Huomattiin, että viestijonoparadigma tarvittiin viestien puskurointiin ja kommunikointiin sopi joukkokommunikointi- tai julkaisija-tilaaja-paradigma.

Järjestelmän kommunikointiin valittiin AMQP-standardi, jonka myötä julkaisija-tilaaja-paradigma päätyi toteutukseen joukkokommunikoinnin sijaan. AMQP ei suoraan ollut tarkoitettu joukkokommunikoinnin toteuttamiseen. IEC 61850 -standardi määritti, että viestit IED-laitteelta tilataan julkaisija-tilaaja-paradigman mukaan. Toteutetun ohjelmiston ja valittujen paradigmojen voidaan sanoa jatkavan IED-laitteen tilausmekanismia ja näin ollen sopivat toteutukseen. Toteutus sallii monen tilaajan tilata sama viesti, mitä IEC 61850 -standardi ei ilman erillistä RCB-instanssia mahdollistanut. Viestin sisältö päädyttiin muuttamaan JSON-muotoon helpomman luettavuuden takia verrattuna MMS-protokollan binääriseen esitysmuotoon. Ohjelman tekemä JSON-muoto on nähtävissä liitteessä A.

Ennen diplomityön aloitusta tekijä oli yrityksessä toteuttanut demon ohjelmiston toimivuudesta. Demo oli tie oppia IEC 61850 -standardin toimintaa ja perehtyä aiheeseen tarkemmin ennen oikeaa toteutusta. Demossa oli kuitenkin ongelmia, mitkä haittasivat sen jatkokehitystä. Diplomityössä analysoitiin demon ongelmia, joita olivat huono suorituskyky, muistivuoto ja toiminnan epävarmuus. Toiminnan epävarmuuteen ja huonoon suorituskykyyn oli syynä Ruby-kielen oletustulkissa oleva globaali tulkkilukitus (GIL/GVL). Ja muistivuoto aiheutui huonon Ruby-koodin ja C-kielen integraatiosta.

Toteutetun ohjelman suorituskykyä saatiin paremmaksi valitsemalla matalamman tason C-ohjelmointikieli. C on käännettävä kieli verrattuna Ruby:n tulkattavaan kieleen. Lisäksi C voi hyödyntää käyttöjärjestelmän säikeitä ilman rajoituksia verrattuna Ruby-tulkin globaaliin lukitukseen. Muistivuoto saatiin korjattua huolellisella ohjelmoinnilla ja varmistamalla, että muisti vapautettiin, kun sitä ei enää tarvittu. Kielen valinnalla ohjelman muis-

tinkäyttö saatiin entiseen nähden pienemmäksi. Ruby:llä toteutettu demo käytti muistia noin 150 Mt ja rcb_sub käytti noin 4 kt. Toteutetussa ohjelmassa ei ollut demossa havaittavia ongelmia ja on osoittanut tuotannossa toimivaksi muun järjestelmän kanssa.

Toteutettu ohjelmisto on tuotannossa osana muuta järjestelmä ja diplomityön kirjoittamisen valmiiksi saamiseen asti on toiminut ongelmitta. Kappaleessa 8 arvioitiin ja pohdittiin saatuja tuloksia. Näiden pohjalta voidaan sanoa, että diplomityössä suunniteltu toteutus pääsi asetettuihin tavoitteisiin ja tutkimuskysymyksiin löydettiin vastaus. Toteutus ei ole kuitenkaan täydellinen ja siinä on jatkokehitettävää. Näihin kohtiin tullaan yrityksessä palaamaan tulevaisuudessa ja muuttamaan niitä, mikäli tarve vaatii. Tämän diplomityön tulokset tarjoavat myös apua samankaltaisten järjestelmien suunnitteluun ja toteutukseen.

LÄHTEET

[1] 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

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

[3] G. R. Andrews, Foundations of multithreaded, parallel, and distributed programming, nide 11, Addison-Wesley Reading, 2000.

[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] R. Baldoni, A. Virgillito, Distributed Event Routing in Publish/Subscribe Communication Systems: a Survey, DIS, Universita di Roma La Sapienza, Tech. Rep, vsk. 5, 2005.

[6] K. P. Birman, ISIS: A system for fault tolerant distributed computing, Cornell University, Department of Computer Science, tekn. rap., huh. 1986.

[7] K. P. Birman, The Process Group Approach to Reliable Distributed Computing, Commun. ACM, vsk. 36, nro 12, jou. 1993, s. 37–53.

[8] A. D. Birrell, B. J. Nelson, Implementing remote procedure calls, ACM Transactions on Computer Systems (TOCS), vsk. 2, nro 1, 1984, s. 39–59.

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

[10] Buffer overflow attack, OWASP verkkosivu. Saatavissa (viitattu 7.11.2018): https://www.owasp.org/index.php/Buffer_overflow_attack

[11] A. Butterfield, G. E. Ngondi, A Dictionary of Computer Science, tam. 2016. Saatavissa: http://www.oxfordreference.com/view/10.1093/acref/9780199688975.001.0001/acref-9780199688975

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

[13] G. Cabri, L. Leonardi, F. Zambonelli, Mobile-agent coordination models for internet applications, Computer, vsk. 33, nro 2, 2000, s. 82–89.

[14] 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.

[15] G. V. Chockler, I. Keidar, R. Vitenberg, Group Communication Specifications: A Comprehensive Study, ACM Comput. Surv., vsk. 33, nro 4, jou. 2001, s. 427–469.

[16] Clients and Developer Tools, RabbitMQ verkkosivu. Saatavissa (viitattu 5.11.2018): https://www.rabbitmq.com/devtools.html

[17] P. T. Eugster, P. A. Felber, R. Guerraoui, A. M. Kermarrec, The many faces of publish/subscribe, ACM computing surveys (CSUR), vsk. 35, nro 2, 2003, s. 114–131.

[18] Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C verkkosivu, mar. 2008. Saatavissa (viitattu 4.11.2018): https://www.w3.org/TR/xml/

[19] I. Fette, A. Melnikov, The websocket protocol, tekn. rap., 2011.

[20] Format string attack, OWASP verkkosivu. Saatavissa (viitattu 7.11.2018): https://www.owasp.org/index.php/Format_string_attack

[21] GCC Manual: Options to Request or Suppress Warnings, GNU verkkosivu. Saatavissa (viitattu 8.11.2018): https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

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

[23] Google Trends: XML ja JSON. Saatavissa (viitattu 15.10.2018): https://trends.google.com/trends/explore?date=today%205-y&geo=US&q=json,xml

[24] G. C. Hillar, MQTT Essentials-A Lightweight IoT Protocol, Packt Publishing Ltd, 2017.

[25] 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

[26] IEC 61850-6 Communication networks and systems for power utility automation – Part 6: Configuration description language for communication in elect-

rical substations related to IEDs, International Electrotechnical Commission, International Standard, jou. 2009, 215 s. Saatavissa (viitattu 15.6.2018): https://webstore.iec.ch/publication/6013

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] E. Intenational, The JSON Data Interchange Syntax, tekn. rap., jou. 2017. Saatavissa (viitattu 4.11.2018): http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

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

[34] JSON API Specification: Recommendations, JSON:API verkkosivu. Saatavissa (viitattu 7.11.2018): https://jsonapi.org/recommendations/

[35] 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.

[36] M. Kerrisk, The Linux programming interface : a Linux and UNIX system programming handbook, No Starch Press, San Francisco, 2010. Saatavissa: https://tut.finna.fi/Record/tutcat.196912

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

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

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

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

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

[42] Message-Oriented Middleware (MOM), Oracle verkkosivu. Saatavissa (viitattu 2.11.2018): https://docs.oracle.com/cd/E19340-01/820-6424/aeraq/index.html

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

[44] MQTT kotisivu, MQTT verkkosivu. Saatavissa (viitattu 2.11.2018): http://mqtt.org/

[45] S. Mullender *et al.*, Distributed systems, nide 12, acm press United States of America, 1993.

[46] G. Mühl, L. Fiege, P. Pietzuch, Distributed Event-Based Systems, Springer, 2006, 384 s.

[47] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta, Comparison of JSON and XML data interchange formats: a case study, Caine, vsk. 9, 2009, s. 157–162.

[48] OASIS, MQTT version 3.1.1, Protocol Specification, lok. 2014. Saatavissa (viitattu 2.11.2018): http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[49] 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

[50] 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

[51] 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

[52] 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

[53] J. Postel, User Datagram Protocol, tekn. rap., 1980.

[54] J. Postel, Transmission Control Protocol, tekn. rap., 1981.

[55] The Power-User's Guide to htop, maketecheasier verkkosivu. Saatavissa (viitattu 3.11.2018): https://www.maketecheasier.com/power-user-guide-htop/

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

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

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

[59] The Rails Command Line, Ruby on Rails verkkosivu. Saatavissa (viitattu 3.11.2018): https://guides.rubyonrails.org/command_line.html#rails-runner

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

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

[62] 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

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

[64] 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.*