Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Tuomas Hyttinen

Operaattoripaneelien ohjelmiston kehitys RTU-tuoteperheeseen

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

28.1.2021

Tekijä Otsikko Sivumäärä Aika	Tuomas Hyttinen Operaattoripaneelien ohjelmiston kehitys RTU-tuoteperheeseen 54 sivua 28.1.2021
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Smart Systems
Ohjaajat	DI Matti Öhman Lehtori Keijo Länsikunnas

Netcon 200 on Netcontrol Oy:n kehittämä, keskijänniteverkon jakelumuuntamoiden alaasemaksi tarkoitettu RTU-tuoteperhe. Netcon 200 -järjestelmä koostuu 1-4 laiteyksiköstä, joista kukin sisältää kiinteän operaattoripaneelin eli HMI:n. Paneeleja ohjaa pääyksikössä toimiva ohjelma, HMI-ajuri, joka asettaa paneelissa näkyvien indikaattoriledien tilat, käsittelee käyttäjän napinpainallukset ja toteuttaa niiden edellyttämät ohjaukset. Insinöörityön tavoitteena on ollut suunnitella ja toteuttaa Netcon 200 -järjestelmän HMI-ajuri. Ohjelmisto on toteutettu C++-kielisenä Linux-sovelluksena, ja sen tapahtumapohjainen arkkitehtuuri perustuu olio-ohjelmoinnin periaatteisiin.

Ohjelmisto koostuu luokista, kuten paneelikäsittelijöistä, kytkinkäsittelijöistä, ledikäsittelijöistä ja tietokantaluokasta, joista kukin vastaa omasta osa-alueestaan HMI:n toiminnassa. Tietokantaluokka kokoaa ajurin seuraamat signaalit ja sen käyttämät komennot omiin tietorakenteisiinsa ja jakelee tietoa muuttuneista signaalien arvoista käsittelijöille. Monet käsittelijäluokista on toteutettu tilakoneina ohjelmalogiikan selkeyttämiseksi. Sovelluksessa on myös käytetty ulkoisia apukirjastoja kuten Libcurlia HTTP-yhteyksien muodostamiseen ja Libeventiä tapahtumapohjaisen arkkitehtuurin pohjana. HMI-ajurille on tehty sekä manuaalista että automatisoitua testausta. Automaattinen testaus on toteutettu Catch-testiympäristössä yksikkötesteinä, jotka käännetään kohdelaitteen arkkitehtuurille sopivaksi testibinääriksi ja suoritetaan sen komentokehoitteessa.

Lopputuloksena voidaan sanoa ohjelmiston toimivan hyvin ja täyttävän sille asetetut vaatimukset. Ohjelmisto soveltuu käytettäväksi Netcon 200 -tuoteperheen laitteissa operaattoripaneelien ohjaukseen. HMI-ajurin suorituskyky on hyvä ja resurssien käyttö kohtuullista. Potentiaalisia kehityskohteita ovat esimerkiksi ajurin käynnistymisvaiheen järjestelmälle aiheuttamien CPU-käyttöpiikkien tasoittaminen, joidenkin suuriksi kasvaneiden luokkien jakaminen pienempiin yksiköihin ja testiautomaation kehittäminen.

Avainsanat	RTU, HMI, operaattoripaneeli, sähköverkkoautomaatio, olio-ohjelmointi, tapahtumapohjaisuus

Author	Tuomas Hyttinen
Title	Developing operator panel software for a line of RTU products
Number of Pages	54 pages
Date	28.1.2021
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Smart Systems
Instructors	Matti Öhman, M.Sc. (Tech.) Keijo Länsikunnas, Senior Lecturer

Netcon 200 is a product line of RTU devices developed by Netcontrol Oy that is particularly designed for use as a secondary substation in medium voltage distribution networks. A Netcon 200 system consists of 1 to 4 device units, each of which contains an HMI panel. To operate, these HMI panels require a software application called a HMI driver. The HMI driver controls each panel's indicator leds, handles button presses and carries out required actions that affect the process under control. The goal of this project has been to develop a HMI driver for Netcon 200. The software is implemented as a Linux-based C++ application and its event-based architecture is implemented in accordance with object-oriented design principles.

The software consists of classes like panel handlers, switch handlers, led drivers and a database class, each of which handles a particular set of responsibilities. Many of the classes are implemented as finite-state machines to better organize the application logic. Some external helper libraries are also used, such as Libcurl and Libevent. The HMI driver has gone through manual testing procedures and unit tests have been implemented for it.

The outcome of the project is a software application that fulfils the requirements set for it and is suited for use in a Netcon 200 system. The application performs well and does not use system resources excessively. Areas for future improvement include further optimization of resource usage, splitting large classes into smaller entities and developing the test automation further.

Keywords	RTU, HMI, operator panel, network automation, object-oriented programming, event-based programming

Sisällys

Lyhenteet

CPU	Central Processing Unit, tietokoneen suoritin eli prosessori.
HMI	Human-Machine Interface, rajapinta ihmisen ja koneen välillä.
HTTP	Hypertext Transfer Protocol, WWW-palvelinten ja selainten väliseen tiedonsiirtoon käytetty sovelluskerroksen verkkoprotokolla.
IEC	International Electrotechnical Commission, kansainvälinen sähköalan standardointiorganisaatio.
IP	Internet Protocol, verkkokerroksen protokolla, joka huolehtii tietoliikennepakettien reitittämisestä perille pakettikytkentäisessä verkossa mahdollistaen Internetin toiminnan.
MAC	Media Access Control, siirtoyhteyskerrokseen kuuluva alikerros. Hoitaa esimerkiksi laitteiden osoitteistuksen (MAC-osoitteet) ja kontrolloi lähetysvuoroja fyysisellä siirtomedialla.
NFE	Network Front-End, Netcontrol Oy:n kehittämä etäkäytön yhteyskeskitin- ja protokollamuuntajaohjelmisto.
PDU	Protocol Data Unit, protokollaspesifi datapaketti.
REST	Representational State Transfer, monissa web-palvelimissa käytettävä ohjelmistoarkkitehtuuri.

RTU	Remote Terminal Unit, mikroprosessoriohjattu laite, joka toimii rajapintana sähköaseman toimilaitteiden ja valvomojärjestelmän välillä.
SCADA	Supervisory Control And Data Acquisition system, teollisuudessa käytettävä prosessin ohjaus- ja seurantajärjestelmä.
TCP	Transmission Control Protocol, luotettavaan tiedonsiirtoon soveltuva kuljetuskerroksen tiedonsiirtoprotokolla.
UDP	User Datagram Protocol, yhteydetön kuljetuskerroksen tiedonsiirtoprotokolla.
UML	Unified Modeling Language, ohjelmistosuunnittelussa käytettävä graafinen mallinnuskieli.

1 Johdanto

Tämä insinöörityö on tehty toimeksiantona Netcontrol Oy:lle, joka on vuonna 1991 Suomessa perustettu, sähköverkkoautomaatioratkaisuja myyvä yritys. Yhtiön tuotevalikoima voidaan jaotella kahteen päähaaraan. Tietoliikenne- ja verkostoautomaatiopuolella siihen kuuluvat sähkö- ja ala-asemien kaukokäyttöyksiköt, radiomodeemit, keskittimet, kytkimet ja katkaisijat. Valvomokäyttöön Netcontrol tarjoaa omaa SCADA-järjestelmäänsä (*Supervisory Control And Data Acquisition system*), joka soveltuu energian tuotanto- ja jakelupisteiden valvontaan, ohjaukseen ja tiedonkeruuseen.

Pääasiassa keskijänniteverkon valvontaan ja ohjaukseen tarkoitettu Netcon 100 -tuoteperhe on tähän asti ollut yhtiön tarjoama ratkaisu myös pienten jakelumuuntamoiden ja kytkinasemien valvontaan ja ohjaukseen. Netcon 100 on fyysiseltä kooltaan melko suuri ja vaatii pienen kaapin kokoisen asennuskehikon. Se on myös nähty hinnoittelultaan ylimitoitetuksi kaikkein pienimmille asemille. Netcontrolilla on tunnistettu tarve kehittää uusi ratkaisu, joka olisi kustannustehokkuutensa ja kokonsa suhteen paremmin sopiva esimerkiksi näiden pienten muuntamoiden hallintaan. Tätä tuoteperhettä kehitetään nimellä Netcon 200.

Insinöörityön tavoitteena on ollut suunnitella ja toteuttaa Netcon 200 -tuoteperheeseen operaattoripaneeleita ohjaava ohjelmisto, johon tekstissä viitataan nimellä HMI-ajuri (*Human-Machine Interface*). Ohjelmisto on konfiguroitavissa tuoteperheen eri RTU-laitteisiin (*Remote Terminal Unit*), ja se mahdollistaa laitteen paikalliskäytön. Se reagoi käyttäjän näppäimenpainalluksiin ja järjestelmän hälytystilojen muutoksiin, välittää käyttäjän antamat ohjauskomennot ohjauslaitteille ja kontrolloi operaattoripaneelin ledejä pitäen huolta siitä, että paneelin indikaatiot vastaavat järjestelmän todellista tilaa.

HMI-ajuria ajetaan taustaprosessina laitteen Linux-käyttäjäavaruudessa. Se on ohjelmoitu C++-ohjelmointikielellä ja käännetty ristikääntäjällä laitteelle sopivaksi PowerPC-arkkitehtuurin binääritiedostoksi. Ajurin suunnittelussa on sovellettu olio-ohjelmoinnin periaatteita, ja se on suunniteltu modulaariseksi, jolloin sen eri osat ovat helpommin yksikkötestattavia ja uusia ominaisuuksia on helpompi lisätä jatkossa.

2 Sähköverkot

Tässä pääluvussa käydään läpi Suomen sähköverkon rakenteeseen ja sähköverkkoautomaatioon liittyviä perusasioita.

2.1 Sähköverkon infrastruktuuri

Suomessa kaikki sähköä tuottavat voimalaitokset ja sähköä kuluttavat tahot kytkeytyvät yhteiseen sähköverkkoon, joka koostuu Fingrid Oyj:n ylläpitämästä kantaverkosta, kantaverkkoon sähköasemien välityksellä kytkeytyvistä suur- ja keskijännitejakeluverkoista sekä jakeluverkoista edelleen kuluttajille tapahtuvasta pienjännitejakelusta [1, s. 54-55].

2.1.1 Kantaverkko

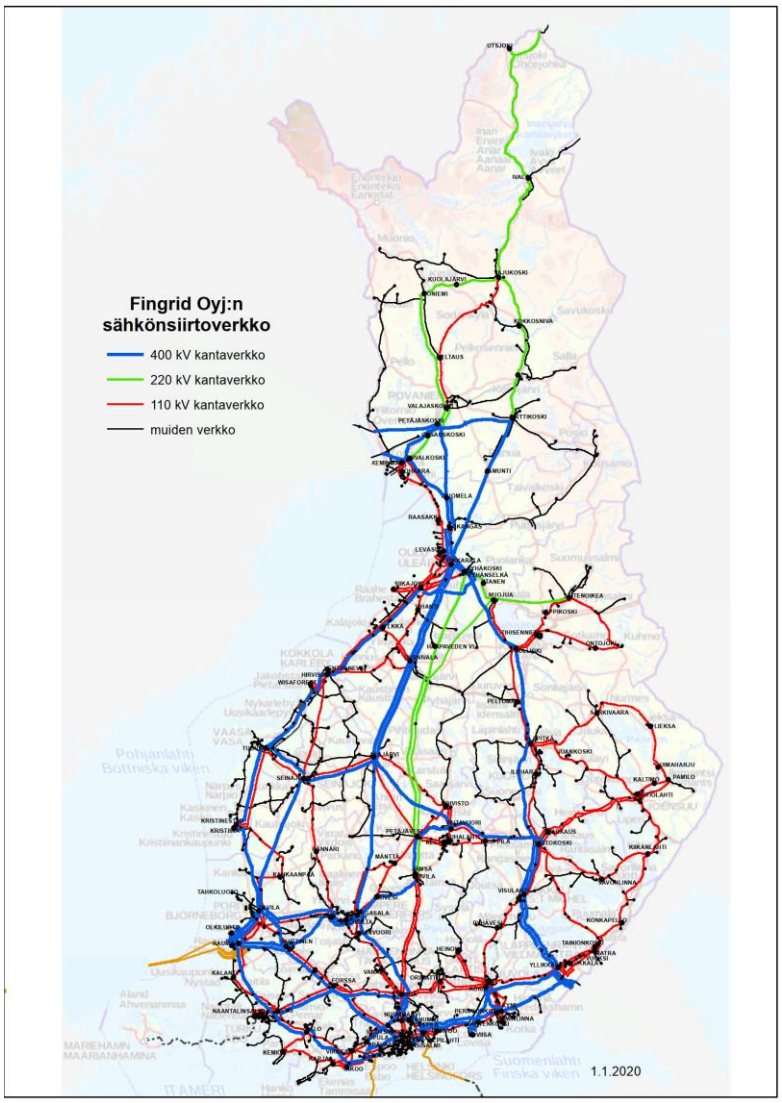
Suomen sähkönsiirron kantaverkko on Fingrid Oyj:n omistuksessa oleva, ylläpitämä ja kehittämä verkko, joka kattaa maantieteellisesti koko valtakunnan alueen ulottuen etelä-pohjois-akselilla Suomen eteläarannikolta Utsjoen tienoille asti ja länsi-itä-akselilla länsirannikolta Pohjois-Karjalaan asti.

Kantaverkko muodostuu 400, 220 sekä 110 kV:n siirtojohdoista ja niitä yhdistävistä sähköasemista. Kantaverkon siirtojännitteet ovat huomattavasti suurempia jakeluverkkoon nähden, sillä siirtojohdot ovat maantieteellisten etäisyyksien takia pitkiä. Järjestelmässä pyritään etäisyyksistä huolimatta aina mahdollisimman hyvään hyötysuhteeseen, mikä tarkoittaa, että siirron aiheuttamat häviöt tulee minimoida. [1, s. 54.] Siirtoverkon pääasiallinen häviö muodostuu pätöteho- eli virtalämpöhäviöstä, joka on verrannollinen johtimessa kulkevan virran neliöön. Kolmivaihejohtimessa pätötehohäviö P on siis

$$P = 3RI^2$$

jossa R on yksittäisen johtimen resistanssi ja I on johtimessa kulkeva sähkövirta. Koska siirrettävä teho on siirtojohdossa vaikuttavan jännitteen ja sähkövirran tulo, tarkoittaa tämä käytännössä sitä, että siirrettäessä suurempia tehomääriä on myös käytettävä suurempia jännitteitä, jotta häviöt eivät kasva kohtuuttomiksi. Siirtotarpeen lisäksi

optimaalista jännitettä kasvattaa siirtomatka. [1, s. 54, 351.] Kuva 1 havainnollistaa Suomen kantaverkon jakauman eri jännitetasoilla olevien verkkosegmenttien kesken.



Kuva 1. Fingrid Oyj:n siirtoverkko. Lähde Fingrid Oy.

2.1.2 Alue- ja jakeluverkot

Valtakunnan siirtoverkossa olevia, varsinaiseen kantaverkkoon kuulumattomia 110 kV:n verkkoja kutsutaan alueverkoiksi tai suurjännitteisiksi jakeluverkoiksi ja niitä ylläpitää Suomessa joukko paikallisia alueverkkoyhtiöitä. Niiden vastuulle kuuluu pääasiassa

sähkön siirto 110 kV:n jännitteellä omistajiensa muuntoasemille. Alueverkot kattavat yleensä suurempia maantieteellisiä alueita kuin millä jakeluverkot toimivat. Alueverkkoyhtiöt eivät kuitenkaan operoi varsinaisia jakelujännitteen (esimerkiksi 20 kV) verkkoja, eikä niiden verkkolupaan liity maantieteellisiä vastuualueita, joiden sisällä niillä olisi liittämis-, siirto- ja verkon kehittämisvelvoite, toisin kuin jakeluverkkoyhtiöillä. [1, s. 61-62.]

Jakeluverkoiksi kutsutaan verkkoja, joiden kautta sähkö siirtyy loppukäyttäjille. Jakeluverkkoja hallinnoi Suomessa noin 80 paikallista sähkönsiirtoyhtiötä. Jakeluverkot eroavat alueverkoista paitsi alhaisemmalla jännitetasollaan myös siten, että niitä hallinnoivien sähkönsiirtoyhtiöiden verkkolupa myönnetään tietylle maantieteelliselle vastuualueelle, jonka sisällä niillä on paikallinen monopoli sekä sen mukanaan tuomat lakisääteiset liittämis-, siirto- ja verkon kehittämisvelvollisuudet. Jakeluyhtiö on siis lain mukaan velvollinen vastuualueellaan kohtuullista korvausta vastaan myymään siirtopalveluja niitä tarvitseville, liittämään tekniset vaatimukset täyttävät sähkön käyttöpaikat sekä voimalat verkkoonsa sekä ylläpitämään ja kehittämään verkkoaan siten, että jakelun laatu ja luotettavuus säilyvät hyvinä. [1, s. 61-62; 2, §19, §20, §21.]

Jakeluverkot voivat käyttää kantaverkkoa liittymällä sähköaseman kautta joko suurjännitteiseen alueverkkoon tai suoraan kantaverkkoon. Jakeluverkkojen yleisin jännitetaso Suomessa on 20 kV, joskin myös esimerkiksi 10 kV:n jännitetasoa käytetään joissain kunnissa. Tyypillisesti jakeluverkko liittyy suurjänniteverkkoon 110/20 kV sähköaseman kautta ja jakaa sähkön kotitalouksiin 20/0,4 kV jakelumuuntamoiden välityksellä.

Jakelumuuntamossa keskijännitteisen jakeluverkon useimmiten 20 tai 10 kV:n jännite muunnetaan 400 V:n pienjännitteeksi, joka voidaan edelleen ohjata sähkön kuluttajille. Jakelumuuntamoita on Suomessa kirjoitushetkellä noin 130 000, ja suurin osa niistä on haja-asutusalueilla sijaitsevia ilmajohtoverkon pylväsmuuntamoita, joissa muuntaja on sijoitettu yleensä puisten pylväiden varaan usean metrin korkeudelle. Maakaapeloiduille alueille, eli toistaiseksi lähinnä taajamiin, rakennetaan sen sijaan puistomuuntamoita eli betoni- tiili- tai metallirakenteisia, pieniä, ikkunattomia rakennuksia, joissa muuntajat, kojeistot ja niitä ohjaava laitteisto on suojassa sääilmiöiltä ja pieneläimiltä. [3; 4.]

2.2 Sähköverkkoautomaatio

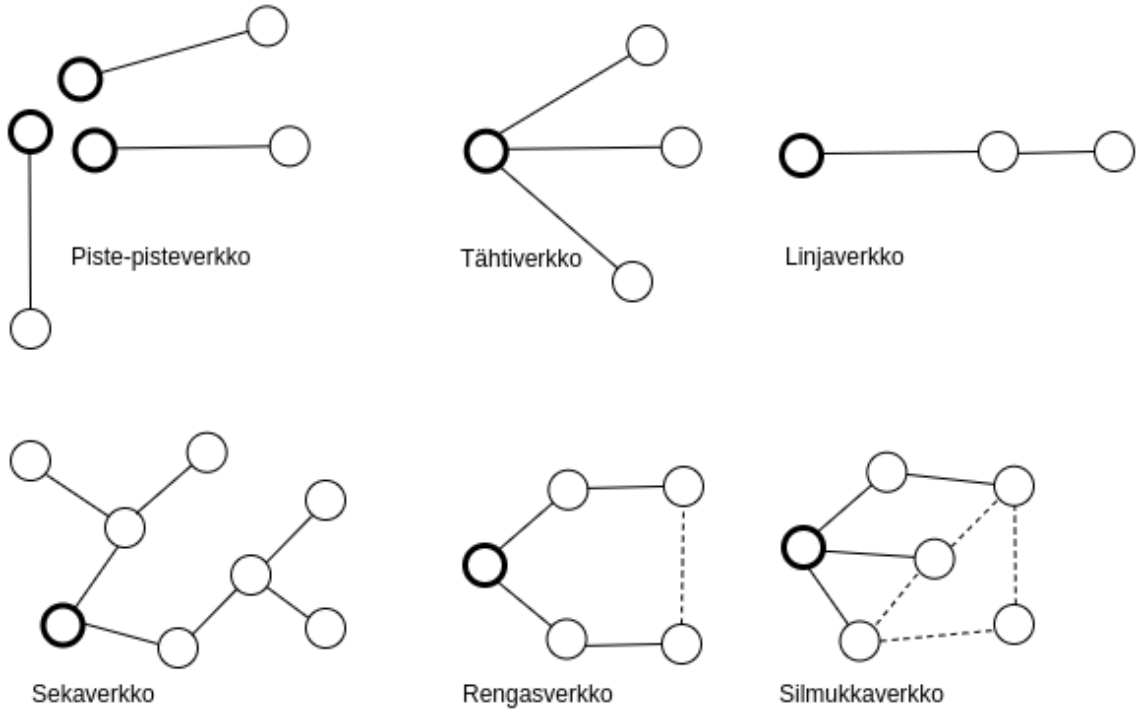
Suomessa sähköverkon käyttö on jo pitkään perustunut pitkälti kauko-ohjauksiin ja -mittauksiin sekä paikallisiin, automaation suorittamiin autonomisiin toimenpiteisiin. Siirto- ja jakeluverkot levittäytyvät laajoille maantieteellisille alueille, minkä takia niiden ohjaus ja valvonta keskitetysti on olennaista. Tästä käytetään nimitystä *kaukokäyttö*, ja siihen sisältyy verkon osa-alueiden kauko-ohjaus, -mittaus ja -säätö, automaattinen häiriötilanteista ilmoittaminen sekä automaattiset relesuojausjärjestelmät. Kaukokäytön avulla parannetaan verkon toimintavarmuutta, nopeutetaan häiriötiloista palautumista sekä säästetään merkittävästi henkilöstömenoissa, sillä sähköasemat ovat miehittämättömiä. Suomessa sähköverkon kaukokäyttö on hierarkkinen järjestelmä, jossa kantaverkkoyhtiö Fingrid hoitaa 400 kV:n ja 200 kV:n siirtoverkkojen sekä kantaverkkoon kuuluvien 110 kV:n verkkojen valvonnan ja ohjauksen omasta voimajärjestelmäkeskuksestaan käsin. Muuta osaa 110 kV:n verkosta operoivat alueverkkoyhtiöt omien alueidensa keskusvalvomoista käsin. Keskijänniteverkkojen osalta jakeluverkkoyhtiöt toimivat vastaavalla tavalla. [5, s. 385.]

2.3 Kaukokäyttöjärjestelmä kokonaisuutena

Varsinkin pienissä jakeluverkkoyhtiöissä kaukokäyttöverkko rakennetaan yleensä yhden keskusaseman ja useamman siihen liittyvän ala-aseman mallilla. Tässä mallissa keskusasema kommunikoi ala-asemien kanssa suorittaen ohjauksia ja valvoen niiden toimintaa. Se myös suorittaa keskitetysti ala-asemilta tulevan tiedon tallennuksen, tilastoinnin ja raporttien laadinnan. Tilastoitavia suureita voivat olla esimerkiksi tuntikeskitehot ja huippuvirrat. Keskusasemalla sijaitsee myös valvomo tietokonelaitteistoineen, joissa nykyään käytetään SCADA-ohjelmistoja. [5, s. 401-402.]

Kaukokäyttöjärjestelmän topologia voi olla rakenteeltaan piste-pisteverkko, linjaverkko, tähtiverkko, rengasverkko, silmukkaverkko tai sekaverkko. Tarkoituksenmukaisin topologia valitaan esimerkiksi käytettävissä olevan tiedonsiirtotien (kaapeli, radiolinkki), datansiirtotarpeen, sähköasemien ja valvomoiden välisten etäisyyksien sekä käytettävyys- ja turvallisuusmääräysten perusteella. Näistä tähti-, linja- ja rengasverkot soveltuvat pienempiin verkkoihin, kun taas alueellisesti suuremmissa verkoissa käytetään lähinnä

rengasrakennetta tai silmukoitua rakennetta. Rengas- ja silmukkatopologian etu on, että vaikka tiedonsiirtoyhteys kahden tai useamman vierekkäisen linkin välillä katkeaisi, tiedonsiirto kauempana oleviin linkkeihin onnistuu silti kiertämällä viallinen yhteysväli. [5, s. 392.]



Kuva 2. Kaukokäyttöjärjestelmien topologioita [5, s. 393]. Keskusasema merkitty paksureunaisella ympyrällä, ala-asemat ohutreunaisilla ympyröillä.

2.4 Sähköaseman automaatio

Sähköaseman automaatiojärjestelmä on kokonaisuus, joka mahdollistaa voimajärjestelmän manuaalisen hallinnan paikallisesti ja kaukokäyttöisesti sekä toteuttaa paikalliset automaattiset toiminnot. Se huolehtii myös tiedonsiirrosta valvomon ja sähköaseman välillä ja suorittaa tarvittavat protokollamuunnokset valvomon, aseman päätelaitteen ja paikallisten toimilaitteiden kuten kytkinlaitteiden välillä. [5, s. 386.]

Sähköasemalla sijaitseva tietoliikenteen päätelaite eli RTU toimii aseman solmukohtana ja välittää aseman kenttäkohtaiset tiedot oikeaan muotoon muunnettuna valvomon SCADA-järjestelmään ja vastaavasti valvomolta tulevat ohjaukset ja säädöt kentän toimilaitteille. Usein RTU-laitteeseen kuuluu myös HMI eli rajapinta, jonka kautta ihminen voi käyttää laitetta paikallisesti. [5, s. 387.]

2.5 Tietoliikenneprotokollat

Tässä alaluvussa esitellään joitakin HMI-ajurin toiminnan kannalta olennaisia tietoliikenneprotokollia.

2.5.1 IEC 60870-5

IEC 60870-5 on International Electrotechnical Commission (IEC) -organisaation standardoima, mm. SCADA-järjestelmissä käytettävää tiedonsiirtoa määrittelevä protokollaperhe.

Protokolla perustuu kolmen kerroksen malliin (*Enhanced Performance Architecture, EPA*), jossa määritellään sovelluskerroksen standardien lisäksi myös suositukset fyysiselle kerrokselle (*physical layer*) sekä joukko siirtoyhteyskerroksen (*data link layer*) siirtoproseduureja. Protokolla on alun perin suunniteltu käytettäväksi perinteisen sarjaliikenneprotokollan, kuten RS-232 päällä [6, s. 8-9]. Netcon 200 -järjestelmässä laiteketjun sisäinen viestintä on toteutettu protokollaperheeseen kuuluvan 60870-5-101 -standardin mukaisella sovelluskerroksen rakenteella. Fyysisen kerroksen ja siirtoyhteyskerroksen toiminnallisuus on kuitenkin toteutettu Ethernetin avulla. Standardin yksityiskohtia Netcon 200 -järjestelmän kontekstissa on kuvattu tarkemmin luvussa 5.8.

2.5.2 HTTP

Hypertext Transfer Protocol eli *HTTP* on sovelluskerroksen verkkoprotokolla, jonka varassa käytännössä kaikki WWW-sivustot toimivat. Se tarvitsee toimiakseen alapuolelleen luotettavan kuljetuskerroksen protokollan, eli käytännössä lähes aina TCP-

yhteyden (*Transmission Control Protocol*). HTTP perustuu palvelimen (*server*) ja asiakasohjelman (*client*) malliin. Siinä asiakasohjelma (yleensä web-selain) lähettää palvelimelle pyynnön, jossa se yksilöi jonkin resurssin, jolloin palvelin lähettää kyseisen resurssin sisältämän datan asiakasohjelmalle vastausviestissä. Resurssi voi olla esimerkiksi HTML-sivu, kuva, video tai mikä tahansa muu palvelimella sijaitseva tiedosto. Se voi olla myös palvelimella toimiva ohjelma, joka tuottaa asiakasohjelman pyytämän datan. [7, kappaleet 1.1, 1,3.] Asiakasohjelman pyynnöt jaotellaan alatyyppeihin, joita kutsutaan *HTTP-metodeiksi*. Taulukossa 1 on listattu osa yleisimmin käytetyistä metodeista.

Taulukko 1. Yleisimmin käytettyjä HTTP-metodeja. Käännetty suomeksi hieman muokaten. [7, kappale 1.4.1.]

GET	Lähetä nimetty resursi palvelimelta asiakkaalle
PUT	Tallenna asiakkaan lähettämä data nimettyyn palvelimen resurssiin
DELETE	Poista nimetty resurssi palvelimelta
POST	Lähetä dataa asiakkaalta palvelimella toimivaan ohjelmaan

2.5.3 REST

Representational State Transfer eli *REST* on web-palveluiden luomisessa käytetty ohjelmistoarkkitehtuuri, jota käytetään yleensä yhdessä HTTP-protokollan kanssa. Siinä määritellään joukko rajoituksia (*constraints*), joita noudattavia web-palveluita kutsutaan usein englanninkielisellä termillä *RESTful services*. [8.]

Ensinnäkin palvelun tulee noudattaa palvelin-asiakassovellusmallia, jossa palvelimella on selkeä vastuunjako ja erottelu dataa ylläpitävän tietokantakerroksen ja palvelun käyttäjärajapinnan välillä. Tällä pyritään parantamaan rajapinnan alustariippumattomuutta ja helpottamaan palvelimen eri ohjelmistokomponenttien toisistaan itsenäistä kehitystä. Toiseksi, asiakkaan ja palvelimen välisen kommunikaation tulee olla tilatonta (*stateless*). Tällä tarkoitetaan sitä, että asiakkaan lähettämän pyynnön tulee sisältää kaikki informaatio, jonka palvelin tarvitsee täyttääkseen pyynnön. Palvelin ei siis tallenna istunnon kontekstiin liittyvää dataa, vaan sen ylläpitäminen on asiakasohjelman vastuulla. [8.]

Palvelimen tulee tarjota rajapinta, jossa resurssit ovat asiakasohjelman saatavilla yksilöllisillä tunnisteilla (*Uniform Resource Identifier, URI*). Web-palvelimen kontekstissa tämä käytännössä tarkoittaa resurssin yksilöivää URL-osoitetta (*Uniform Resource Locator*). Palvelimen on myös merkittävä vastausviestinsä metatietoihin, voiko asiakasohjelma tallentaa vastauksen sisältämän datan omaan välimuistiinsa myöhempää käyttöä varten. Välimuistia käytettäessä voidaan saavuttaa nopeampia vasteaikoja, koska staattinen, muuttumaton data voidaan lukea suoraan välimuistista ilman, että sitä tarvitsee joka kerralla pyytää palvelimelta. [8.]

3 Ohjelmointikonseptit

Tässä pääluvussa käydään läpi kaksi HMI-ajurin arkkitehtuurin kannalta olennaista ohjelmistosuunnittelun konseptia, olio-ohjelmointi ja tilakone.

3.1 Olio-ohjelmointi

Olio-ohjelmointi (*Object-Oriented Programming, OOP*) on ohjelmointiparadigma, jolla pyritään laajojen ja monimutkaisten ohjelmistokokonaisuuksien parempaan hallintaan ja jäsentelyyn järjestämällä lähdekoodi luokiksi (*classes*), joista luodaan ilmentymiä eli olioita (myöhemmin tekstissä myös *objekti*, engl. *object*). Oliot ovat itsenäisiä kokonaisuuksia, jotka pystyvät ohjelmassa vuorovaikuttamaan ja kommunikoimaan keskenään niille määriteltyjen rajapintojen avulla. Suuri osa moderneista ohjelmointikielistä mahdollistaa

olio-ohjelmoinnin yhtenä tuettuna paradigmana. Esimerkiksi Java suorastaan pakottaa ohjelmoijan siihen rakenteensa takia [9].

Olio-ohjelmoinnin filosofiassa neljä perusperiaatetta ovat kapselointi (*encapsulation*), abstraktio (*abstraction*), perintä (*inheritance*) ja polymorfismi (*polymorphism*) [10]. Kapseloinnilla tarkoitetaan ohjelman sisältämän datan ja sitä käyttävien aliohjelmien ryhmittelyä itsenäisiksi kokonaisuuksiksi [11, s. 460]. Olio-ohjelmoinnissa tämä toteutetaan luokilla. Luokka on lähdekoodissa tehty formaali määrittely, joka kuvaa abstraktia datatyyppiä. Luokan kuvauksessa määritellään, millaista dataa se sisältää, funktiot, joilla tätä dataa käsitellään sekä erityiset metodit olion luomiseen ja tuhoamiseen (*konstruktori ja destruktori*).

Kapselonnin toinen tärkeä aspekti on luokan sisäiseen toteutukseen liittyvien yksityiskohtien piilottaminen luokan käyttäjältä [11, s. 460]. Tavat tehdä kapselointia käytännössä eroavat eri ohjelmointikielissä, mutta perusperiaate on sama: hyvin suunniteltu luokka tarjoaa selkeästi määritellyn rajapinnan, joka sisältää ainoastaan funktiot, jotka ovat tarpeellisia luokan käyttämiseen sen ulkopuolelta. Se, miten luokka käsittelee siihen tuotua dataa ja miten se sisäisesti toteuttaa ulkopuolelta saamansa pyynnöt, ei kuulu luokan rajapintaan vaan toteutukseen. C++-kielen standardissa rajaus toteutetaan luokan kuvauksessa määrittelemällä rajapintaan kuuluvat ominaisuudet julkisiksi (*public*) ja sisäiseen toteutukseen liittyvät funktiot ja data yksityiseksi (*private*). Yksityisiksi määriteltyjen jäsenfunktioiden ja -muuttujien käsittely luokan ulkopuolella tuottaa kielen standardin mukaisesti käännösvaiheessa virheen, joka estää ohjelmabinäärin muodostumisen. C++-ohjelmoinnissa toteutuksen ja rajapinnan rajanvetoa korostetaan yleensä myös kirjoittamalla ne eri tiedostoihin (niin sanottuihin *header* -tiedostoihin ja varsinaisiin lähdekooditiedostoihin).

Luokka paitsi kapseloi, myös abstraktoi sisältämiään toimintoja. Abstraktiolla tarkoitetaan tietojenkäsittelyn kontekstissa toteutukseen liittyvien yksityiskohtien kätkemistä arkkitehtuuritasolla; toimintojen kompleksisuus piilotetaan yksinkertaisten rajapintojen taakse [11, s. 112]. Abstraktion ansiosta oliota käyttävä ohjelmoija voi keskittyä siihen, mitä olio tekee huolehtimatta siitä, miten se toimii. Tämä helpottaa huomattavasti laajojen ohjelmien kompleksisuuden hallintaa. Abstraktion etu on myös, että rajapinnan

takana oleva toteutus (esimerkiksi algoritmi) voidaan vaihtaa ilman muutoksia muuhun ohjelmaan, kunhan rajapintaan ei tehdä muutoksia.

Ohjelmistokehityksessä on hyvin yleistä, että kehitettävät toiminnot rakentuvat aikaisemmin kehitetyn koodin päälle laajentaen sen toiminnallisuutta. Uusi ohjelmakomponentti voi olla pitkälti samankaltainen kuin edellinen, mutta vaatia joitain lisäyksiä, muutoksia tai tarkennuksia tiettyihin toimintoihin. Yksi vaihtoehto tuottaa tällainen hieman erilainen versio on kopioida jo olemassa olevan toiminnon lähdekoodi uuteen muokaten sitä vaadituilta osin. Tämä lähestymistapa tuottaa kuitenkin hankalasti ylläpidettävää koodia, jossa toiminnoille yhteisten vaatimusten muuttuessa muutokset on tehtävä erikseen jokaiseen ohjelmakomponenttiin. Tällainen kopioimismenettely on myös altis virheille. [12, s.460.]

Perintä on yksi olio-ohjelmoinnin tärkeimmistä konsepteista ja ratkaisu edellä kuvattuihin ongelmiin. Sillä tarkoitetaan ohjelmointikielen mekanismia, jolla luotava luokka *D* voidaan määritellä periytyväksi toisesta luokasta *B*. B:tä kutsutaan tällöin yläluokaksi (*base class*) ja D:tä sen alaluokaksi (*derived class*). Tällöin luokka D implisiittisesti sisältää kaikki luokassa B määritellyt toiminnot ja datan, joita B:ssä ei erikseen ole rajattu koskemaan vain sitä itseään. Alaluokkaan tehdään vain ne lisäykset ja muutokset, joiden osalta se poikkeaa perimästään yläluokasta. (12, s. 462.) Jos luokka D määrittelee samannimisiä toimintoja tai dataa kuin mitä B:ssä on määritelty, D:n versiot korvaavat sen itsensä osalta yläluokassa määritellyt versiot. Esimerkkikoodi 1 havainnollistaa perintää. Siinä luokka D perii B:stä funktion *foo()* ja kokonaislukumuuttujan *a*, mikä on nähtävissä stdout-tulosteesta ohjelmaa ajettaessa. Funktio *bar()* on määritelty sekä ylä- että alaluokassa, joten se ei periydy. Kutsuttaessa D-luokan bar() -nimistä funktiota ajetaan D:n oma toteutus siitä. Lisäksi D:ssä on laajennettu toiminnallisuutta B:hen verrattuna lisäämällä *baz()*-funktio.

```
#include <iostream>
using namespace std

class B {
public:
      B()  {  }
      void foo()           { cout << "foo(), määritelty luokassa B" << endl; }
      virtual void bar()   { cout << "bar(), määritelty luokassa B" << endl; }
      void print_a()       { cout << "a = " << a  << endl; }
protected:
    int a;
};


class D : public B {
public:
      D(int _a) { this->a = _a; }
      void bar() { cout << "bar(), määritelty luokassa D" << endl; }
      void baz() { cout << "baz(), määritelty luokassa D" << endl; }
};


int main()
{

    D d(6);
    d.foo();     /* Tulostaa: "foo(), määritelty luokassa B" */
    d.bar();     /* Tulostaa: "bar(), määritelty luokassa D" */
    d.baz();     /* Tulostaa: "baz(), määritelty luokassa D" */
    d.print_a(); /* Tulostaa: a = 6 */
    return 0;
}
```

Esimerkkikoodi 1. C++-kielinen esimerkki perinnästä.

Polymorfismilla tarkoitetaan sitä, että datatyypillä voidaan viitata myös siitä periytyneisiin tyyppeihin. Jos tarkastellaan edellä esitettyä esimerkkiä perinnästä, voidaan sanoa, että ”D on eräänlainen B”, koska kaikki operaatiot, jotka ovat valideja B:lle, ovat valideja myös D:lle. [12, s. 469.] Käytännössä tämä tarkoittaa, että tyyppiä B oleva osoitin- tai referenssimuuttuja voidaan ajon aikana määritellä osoittamaan D-tyypin objektiin, ja sen kautta voidaan käyttää kaikkia niitä D:n ominaisuuksia, jotka on määritelty B:ssä. Luokan B rajapinnasta nähdään, että funktio bar() on määritelty virtuaaliseksi (*virtual*). C++-kielessä yläluokassa virtuaalisiksi määritellyistä funktioista ajetaan aina luokkaa polymorfisesti käytettäessä alaluokan toteuttama versio, jos sellainen on olemassa. Toisin sanoen seuraava esimerkkikoodi toimii:

```
D d(5);              /* Luodaan D-tyyppinen olio d*/
B *b_ptr = &d;  /* Luodaan B-tyyppinen osoitin b_ptr viittaamaan d:hen*/

b_ptr->foo();        /* Tulostaa: "foo(), määritelty luokassa B" */
b_ptr->bar();        /* Tulostaa: "bar(), määritelty luokassa D"*/
b_ptr->print_a();    /* Tulostaa: "a = 5"*/
```

Esimerkkikoodi 2. Polymorfismi C++:ssa.

Alaluokka voitaisiin myös pakottaa toteuttamaan oma versionsa funktiosta määrittelemällä se puhtaasti virtuaaliseksi (*pure virtual*). Virtuaalisia funktioita käyttämällä yläluokan rajapinnassa voidaan tehdä yleistyksiä, joista alaluokat todellisuudessa toteuttavat omat, tarkennetut versionsa.

Polymorfismi on ohjelmistosuunnittelussa erittäin tehokas ominaisuus, sillä se mahdollistaa tiettyyn tyyppihierarkiaan kuuluvien olioiden käsittelemisen yhteisen, abstraktin rajapinnan kautta. Tällöin voidaan koota esimerkiksi monia erilaisia, yhteisestä yläluokasta periytyviä datatyyppejä samaan taulukkoon, jolloin säilytetään tyyppiturvallisuus, koska käännösvaiheessa pystytään varmistamaan, että kaikki taulukon tyypit ovat yhteensopivia rajapinnan kanssa. On kuitenkin huomattava, että edellä mainittu joustavuus tyyppien välillä ei viittaa minkäänlaiseen ajonaikaiseen mutaatioon itse objekteissa, vaan siihen, että tietyn datatyypin osoitin tai referenssi voidaan määrittää osoittamaan myös omaan alityyppiinsä. Polymorfiset arvon asetukset ovat siis valideja vain osoitin- ja referenssimuuttujilla. [12, s. 470-472.]

3.2 Äärelliset tilakoneet

Sulautetut ohjelmistot sisältävät usein komponentteja (esimerkiksi funktioita tai luokkia), joiden tulee suorittaa erilaisia operaatioita tai suorittaa niitä eri tavalla riippuen tietyistä komponentin sisäisistä ja sen ulkopuolisista ehdoista. Erilaiset ohjausjärjestelmät ovat esimerkkejä tällaisista sovelluksista. Ne vastaanottavat ulkopuolelta tietynlaisia syötteitä ja suorittavat toimenpiteitä, jotka vaikuttavat ohjattavaan prosessiin.

Kuvitellaan funktio, jonka tehtävänä on ohjata kahta kytkinkojeistoa. Se vastaanottaa syötteinä operaattoripaneelin painonappitapahtumista kertovia viestejä. Funktion

logiikka on toteutettu sarjana yksinkertaisia ehtolauseita. Pseudokoodina ilmaistu esimerkki funktion logiikasta on seuraavanlainen:

```
Jos (painiketapahtuma.painike_id == suorita) Sitten: ohjaa_kytkintä()
```

Edellä kuvattu ehtolause ei kuitenkaan yksin riitä takaamaan funktion oikeanlaista toimintaa, sillä suorituskomennon lisäksi tarvitaan tieto siitä, onko käyttäjä sitä ennen valinnut ohjattavan kytkimen. Ohjelmoijan on siis lisättävä logiikkaan tarkistus, onko käyttäjä valinnut jommankumman kytkimistä:

```
Jos (painiketapahtuma.painike_id == suorita && valittu_kytkin != NULL) Sitten:
     valittu_kytkin.ohjaa()
```

Tämän jälkeen kuitenkin huomataan, että mainitun tarkistuksen lisäksi ohjelman on aina tarkistettava ennen ohjauksen suorittamista, onko järjestelmä lukitussa tilassa. Jos näin on, ohjausta ei saa suorittaa, vaan sen sijaan on annettava käyttäjälle äänipalaute. Koodia on jälleen muutettava monimutkaisemmaksi:

```
Jos (painiketapahtuma.painike_id == suorita && valittu_kytkin != NULL) Sitten:
     Jos (lukittu == epätosi) Sitten:
          valittu_kytkin.ohjaa()
     Muuten:
          summeri.soita_virheääni()
```

Todellisuudessa ohjausjärjestelmien logiikka on usein vielä monimutkaisempaa. Erilaisia tarkistettavia ehtoja on paljon, mikä johtaa edellä kuvatussa lähestymistavassa sisäkkäisten ehtolausekkeiden määrän kasvuun. Koodista tulee tällöin usein hankalasti luettavaa ja ylläpidettävää.

Eräs erittäin tehokas, ja siksi laajalti käytössä oleva, keino mallintaa ja toteuttaa tällainen hiemankaan monimutkaisempi järjestelmä on kuvata se äärellisenä tilakoneena (*Finite-state Machine, FSM*). Automaatioteorian kontekstissa malliin viitataan äärellisenä automaattina (*Finite-state Automaton, FSA*). Tilakonemallissa järjestelmälle on määritelty rajattu joukko tiloja, jotka kuvaavat kaikkia niitä tilanteita, joissa järjestelmän voidaan kuvitella olevan. Se, mitä toimenpiteitä järjestelmä suorittaa, määräytyy paitsi saadun syötteen, myös järjestelmän senhetkisen tilan perusteella. Tila pitää sisällään tiedon siitä, mitä järjestelmässä on aikaisemmin tapahtunut. [13, s. 63-64.]

Tilakoneen määritelmään kuuluu olennaisena osana myös se, miten se siirtyy tilasta toiseen. Deterministinen äärellinen tilakone (*Deterministic Finite-state Automaton, DFA*), jollaisia tämän insinöörityön aiheena olevassa ohjelmistossa on käytetty, voi olla kerrallaan ainoastaan yhdessä tilassa. DFA muodostuu äärellisestä joukosta tiloja Q, äärellisestä joukosta mahdollisia syötteitä Σ, siirtymäfunktiosta δ, alkutilasta q_0, joka kuuluu joukkoon Q sekä joukosta lopputiloja F, joka on Q:n osajoukko. DFA kuvataankin usein automaatioteoriassa viisikkona A = (Q, Σ, δ, q_0, F), jossa A on DFA:n nimi. Siirtymäfunktio δ määrittelee sen, mihin tilaan jostakin tilakoneen tilasta q siirrytään syötteellä a. Funktio merkitään siis

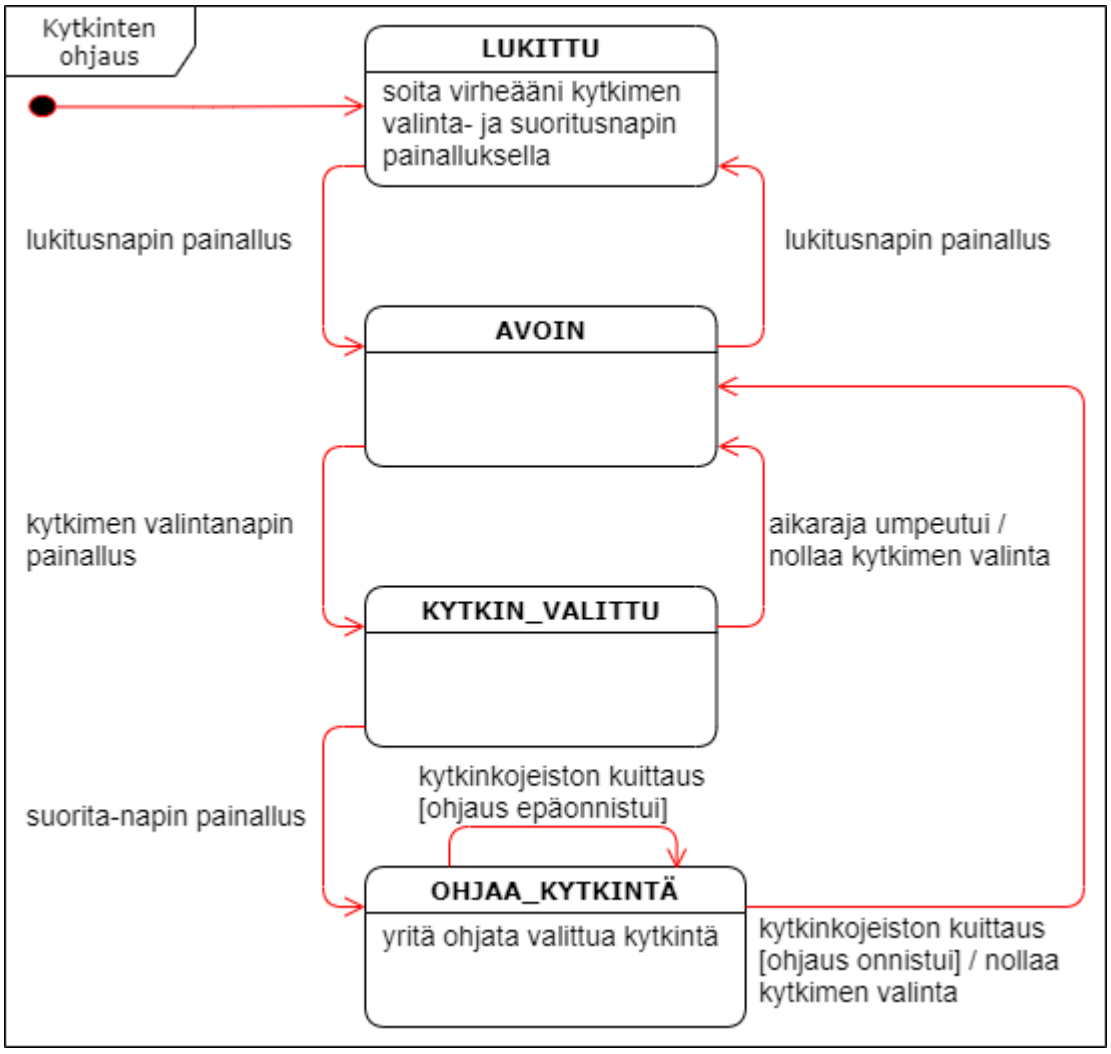
$$\delta(q, a) = p$$

jossa p on siirtymän jälkeinen tila. (14, s. 45-46.) Automaatioteoria tuntee myös epädeterministisen äärellisen automaatin (*Nondeterministic Finite-state Automaton, NFA*), joka voi olla kerrallaan useammassa kuin yhdessä tilassa. Siinä on siirtymäfunktion sijasta siirtymärelaatio. [14, s. 45-46, 55-56.]

Tilakoneiden logiikan kuvaamiseen voidaan käyttää esimerkiksi UML-standardin (*Unified Modeling Language*) mukaista tilakaaviota *(statechart diagram*), jossa tilat merkitään kulmista pyöristettyinä laatikoina, joiden sisään voidaan merkitä tilaan kuuluvia toimenpiteitä. Tilasiirtymät merkitään näiden laatikoiden välisinä nuolina. Nuolten viereen voidaan merkitä siirtymään liittyvää informaatiota syntaksilla

laukaisija [siirtymäehto] / seuraus.

Syntaksissa laukaisija tarkoittaa syötettä, joka voi johtaa kyseisten tilojen väliseen siirtymään. Siirtymäehto on ehto, jonka tulee toteutua, jotta siirtymä suoritetaan ja seuraus on jokin toimenpide, joka suoritetaan tilasiirtymän yhteydessä. [15, s. 321.] Edellä esitetty esimerkki kytkimiä ohjaavasta funktiosta selkeytyy, kun se toteutetaan tilakoneena ja esitetään kuvan 3 mukaisena UML-tilakaaviona.



Kuva 3. KytkinOhjaaja-luokan tilakoneen UML-tilakaavio.

C- ja C++-ohjelmointikielissä tällainen yksinkertainen tilakone voidaan toteuttaa käyttämällä *switch-case*-lauseketta ja tilojen mukaan nimettyjen kokonaislukuvakioiden listaa (*enum*). Esimerkkikoodi 3 havainnollistaa, miten *fsm()*-funktiota kutsuttaessa tarkastetaan ensiksi tilakoneen senhetkinen tila muuttujasta *nyk_tila* ja suoritetaan vain kyseiseen tilaan liittyvä koodi. Näin funktiossa ei tarvitse toteuttaa monimutkaista tarkistusten sarjaa. Jos nykyiselle tilalle ei ole määritelty saadun syötteen mukaisia toimenpiteitä, tilakoneesta poistutaan tekemättä mitään.

```
class KytkinOhjaaja {
public:
	enum tila {
		avoin,
		lukittu,
		kytkin_valittu,
		ohjaa_kytkinta,
	};
	enum kytkin {
		ei_valittu,
		k1
	};

	KytkinOhjaaja () : nyk_tila(lukittu), valittu_kytkin(ei_valittu) {}
	/*Konstruktori alustaa alkuperäisen tilan lukituksi
	ja nollaa kytkinvalinnan*/

	void fsm(Tapahtuma t) {
		switch(nyk_tila) {
		case avoin:
			if(t.id == lukitusnappi_painallus)
				nyk_tila = lukittu;
			else if(t.id == k1_valintanappi_painallus)
				nyk_tila = kytkin_valittu;
				valittu_kytkin = k1;
			break;
		case lukittu:
			if(t.id == lukitusnappi_painallus)
				nyk_tila = avoin;
			break;
		case kytkin_valittu:
			if(t.id == suoritanappi_painallus) {
				nyk_tila = ohjaa_kytkinta;
				aikatauluta_uusi_fsm_kutsu();
			}
			else if(t.id == aikaraja_umpeutui) {
				nyk_tila = avoin;
				valittu_kytkin = ei_valittu;
			}
			break;
		case ohjaa_kytkinta:
			bool onnistui = ohjaa_kytkinta(valittu_kytkin);
			if(onnistui) {
				nyk_tila = avoin;
				valittu_kytkin = ei_valittu;
			}
			else
				aikatauluta_uusi_fsm_kutsu();
			break;
		}
	}
private:
	tila nyk_tila;			/*Ylläpitää tilakoneen tilaa*/
	kytkin valittu_kytkin;	/*Ylläpitää tietoa kytkinvalinnasta*/
};
```

Esimerkkikoodi 3. Kuvitteellinen kytkinohjausjärjestelmä luokkana, joka sisältää luokan ulkopuolelta kutsuttavan tilakoneen.

4 Netcon 200 -tuoteperhe

Netcon 200 on esimerkiksi jakelumuuntamoiden ohjaukseen soveltuva RTU-tuoteperhe. Netcon 200 -järjestelmä on modulaarinen ja koostuu vähintään pääyksiköstä (*main unit*), jonka rinnalle voidaan lisätä toimintoja 1-3 laajennusyksiköllä (*extension unit*). Netcon 200 -yksikkö mahtuu pienehköön muovikoteloon, joka sisältää emolevyn, IO (*input-output*) -kortit, joihin kytketään mittauksiin ja ohjauksiin liittyvät signaalit sekä operaattoripaneelin eli HMI:n. Laiteketjun pääyksikköön kuuluu lisäksi pienen Linux-tietokoneen sisältävä gateway-kortti sekä virtalähdekortti (*PSU card*). HMI on kiinteä osa laitteen koteloa. Se koostuu joukosta aseman tilaa kuvaavia indikaattoriledejä sekä painonapeista, joilla laitetta operoidaan.

Jokaisessa laiteyksikössä on samanlainen emolevy (*base board*), joka sisältää neljä korttipaikkaa. Yksiköiden tyyppinimet muodostetaan niiden korttikombinaatiota kuvaavista kirjainyhdistelmistä. Insinöörityön kirjoitushetkellä olemassa olevia pääyksikkötyyppejä on kaksi: *GWDD-02* ja *GWDD-03*. Näistä GWDD-02 sisältää 2 ohjattavaa kytkintä ja GWDD-03:ssa kytkimiä on neljä. Molemmat yksiköt koostuvat virtalähdekortista, gateway-kortista ja kahdesta digitaalisesta IO-kortista (*DIO card*). Koska molemmissa yksiköissä on sama korttikombinaatio ja näin ollen sama määrä IO:ta, GWDD-02-yksikössä yleiskäyttöisiä digitaalituloja- ja lähtöjä jää vapaaksi useampia, koska ohjattavia kytkimiä on vähemmän. Yksiköt ovat muutoin identtisiä keskenään.

Järjestelmään kuuluvat yksiköt liitetään yhteen RJ45-kaapelilla laiteketjuksi (*daisy chain*) kuvan 4 osoittamalla tavalla. Kaapeli kuljettaa yksiköiden välisen Ethernet-liikenteen sekä tarjoaa laajennusyksiköille käyttöjännitteen. Laajennusyksiköt voivat sisältää joko lisää kytkimiä, geneerisiä digitaalituloja ja -lähtöjä tai mittaus- ja suojaustoimintoja. Kirjoitushetkellä järjestelmässä on tuki *VCCC-32*-tyyppiselle laajennusyksikölle. Se sisältää neljä mittauskorttia, joista yksi on jännite- ja kolme virtamittauksia varten.

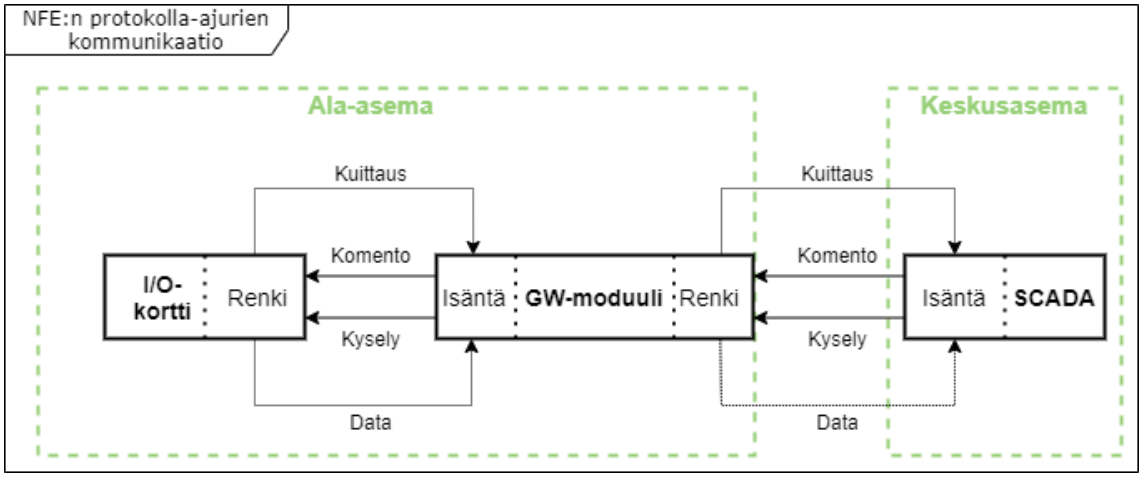
Kuva 4. Netcon 200 -järjestelmä GWDD-02- ja VCCC-32-yksiköillä. Lähde Netcontrol Oy.

4.1 NFE

Netcon NFE (*Network Front-End*) on Netcontrolin kehittämä etäkäytön yhteyskeskitin- ja protokollamuuntajaohjelmisto, joka on käytössä yhtiön valmistamissa gateway-laitteissa Netcon 200 mukaan lukien. NFE ylläpitää laitteen keskusmuistissa hetkellistietokantaa, joka sisältää laitteen hallinnoiman prosessin tilaa ja sen ohjaamista koskevia datapisteitä. Hetkellistietokannalla tarkoitetaan sananmukaisesti tietorakennetta, joka ainoastaan peilaa prosessin tilaa tarkasteluhetkellä. Tilojen historiasta ei siis pidetä kirjaa eikä niitä tallenneta tietokoneen massamuistiin. [16, s. 5.]

NFE:hen kuuluu useita erilaisia protokolla-ajureita, jotka kirjoittavat ja lukevat hetkellistietokannassa olevia datapisteitä ja joista moni toteuttaa jonkin tietyn kommunikaatioprotokollan. Protokolla-ajurit jakautuvat niin sanottuihin isäntä- ja renkityyppisiin ajureihin (*master/slave*). Ala-aseman gateway-moduulin isäntäajurit kommunikoivat suoraan

hallinnoitavan prosessin kanssa välittäen esimerkiksi SCADA-järjestelmästä lähteneitä komentoja toimilaitteille ja kirjoittamalla tietokantaan prosessin tilaa kuvaavia arvoja. Ala-aseman renkiajurit puolestaan lukevat tietokannan datapisteitä ja lähettävät niiden sisältämää tietoa SCADA-järjestelmässä toimivalle isäntäajurille sekä yrittävät kirjoittaa SCADA-järjestelmästä lähteneitä ohjauskomentoja NFE:n tietokantaan herättäen I/O-korttien suuntaan kommunikoivan isäntäajurin toimimaan. Kun johonkin lukuoperaatioon tai ohjaukseen osallistuvat isäntä- ja renkiajuri toteuttavat eri kommunikaatioprotokollia, tapahtuu protokollamuunnos. [16, s. 5.] Kuva 5 havainnollistaa isäntä- ja renkiajurien välistä kommunikaatiota.



Kuva 5. NFE:n protokolla-ajurien viestintä. UML-viestintäkaavio.

HMI-ajurin on toimiakseen seurattava useita Netcon 200 -järjestelmän tilaa kuvaavia datapisteitä, kuten kytkinkojeistojen ja erilaisten hälytysten tilatietoja, aseman hallintamoodia sekä ledien tilatietoja. Suurin osa näistä datapisteistä sijaitsee NFE:n tietokannassa, ja niihin viitataan tekstissä *signaaleina*. Signaalit ovat NFE:n käyttäjärajapintojen näkökulmasta kirjoitussuojattuja (*read-only*). Signaalien lisäksi tietokanta sisältää kirjoitussuojaamattomia (*read-write*) datapisteitä, joiden avulla laitteelle voi lähettää ohjauskomentoja, kuten ohjata kytkimen kiinni tai siirtää aseman kaukokäyttötilaan. Niihin viitataan tekstissä *komentoina*.

Yksi NFE:n keskeisistä komponenteista on HTTP-palvelin, joka tarjoaa tietyssä paikallisen tietokoneen TCP-portissa REST-rajapinnan edellä mainittujen toimintojen

suorittamiseksi. Rajapinta sisältää tarvittavat palvelut mm. datapisteiden arvojen lukemiseen ja kirjoittamiseen, datapisteiden nimien muuntamiseen NFE-osoitteiksi sekä niin sanotun tilaajarajapinnan (*subscription interface*), jonka toimintaan HMI-ajurin arkkitehtuuri laajalti perustuu.

Jokaisella datapisteellä on tietyn loogisen kaavan mukaan muodostettu nimi (*tag*). Nimet ovat ihmisen tulkittavaksi tarkoitettuja, datapistettä kuvailevia merkkijonoja. Datapisteisiin viittaaminen niiden arvoja ohjelmallisesti lukiessa tai kirjoittaessa tapahtuu kuitenkin käyttämällä niiden NFE-osoitteita eli numerosarjoja, jotka yksilöivät datapisteet tietokannan sisällä. Rajapintaa käyttävän ohjelman konfiguraatiotiedostossa on käytännöllisempää määritellä datapisteet nimen perusteella, joten sen on saatava jostain käännös nimistä niitä vastaaviin osoitteisiin. Tätä varten REST-rajapinnassa on *mltconv*-palvelu.

Yksittäisen signaalin tilan voi kysyä rajapinnasta *npcread*-palvelun avulla, kun taas komennon lähettäminen tapahtuu käyttämällä *npcwrite*-komentoa (yksittäinen komento) tai *mltwrite*-komentoa (useita komentoja). HMI-ajurin kannalta käytännöllisempi tapa vastaanottaa informaatiota palvelimelta on kuitenkin tilaajarajapinnan kautta.

Mallissa rajapinnan käyttäjän määrittelemät signaalit rekisteröidään ensin tilausryhmäksi (*subscription group*). Tämän jälkeen ryhmän sisältämien signaalien tilaa voidaan seurata niin sanotulla long polling -tekniikalla. Siinä käyttäjä lähettää palvelimelle ryhmään liittyvän kyselyn, jolloin palvelin jättää käyttäjän avaaman yhteyden auki ja lähettää dataa vastausviestissä heti, kun sitä on saatavissa. Sen sijaan, että ajuri kyselisi jatkuvasti, ovatko signaalien arvot muuttuneet, se käyttää suurimman osan ajasta odottaen, että rajapinta ilmoittaa sille, kun näin tapahtuu. Tilausryhmän käyttäjän tulee määritellä sen sisältämät signaalit vain ryhmän luomisen yhteydessä, ja itse kyselyt tehdään ryhmälle annettua tunnusta käyttämällä. Kyselyiden muodostaminen on siis kevyt toimenpide.

Rajapinta sisältää tätä varten *subreg*- ja *subpoll*-palvelut. Näistä subreg ottaa parametrina niiden signaalien osoitteet, joiden tilaa rajapinnan käyttäjä haluaa seurata ja luo niistä tilausryhmän. Palvelu palauttaa vastausviestissä ryhmän tunnuksen kokonaislukuna. Tämän jälkeen voidaan kutsua subpoll-palvelua antamalla sille parametrina ryhmän tunnus. Kun palvelin vastaanottaa subpoll-pyynnön, se alkaa seurata kaikkia kyseiseen ryhmään kuuluvia signaaleja, ja jos niistä yhden tai useamman tila muuttuu, se

lähettää pyyntöön vastauksen, joka sisältää muuttuneiden signaalien osoitteet, uudet arvot, mahdolliset lippubitit sekä tapahtumien aikaleimat. Muuttuneet signaalit on eroteltu toisistaan rivinvaihdolla. Subpoll-pyyntö vanhenee 30 sekunnissa. Mikäli yksikään signaali ei tämän aikarajan umpeutumiseen mennessä ole muuttunut, palvelin lähettää tyhjän vastausviestin.

4.2 WebGUI

Netcontrolin valmistamissa RTU-tuotteissa Netcon 200 mukaan lukien on järjestelmän konfigurointia ja testaamista varten myös selaimessa toimiva graafinen käyttöliittymä, *WebGUI*. Tuotteen käyttöönoton yhteydessä riittävillä oikeuksilla varustettu käyttäjä voi WebGUI:n kautta tehdä järjestelmän toimintaan vaikuttavia määrityksiä kuten luoda erilaisia summahälytyslohkoja ja muuttaa IO-parametreja. Onnistuneen määrittelyn ja tietojen tallennuksen jälkeen käyttäjä aktivoi uudet asetukset, jolloin WebGUI luo käyttäjän syötteiden ja valmiiden dokumenttipohjien avulla järjestelmän eri sovellusten (esimerkiksi NFE, HMI-ajuri) tarvitsemat konfiguraatiotiedostot ja tallentaa ne laitteen massamuistiin. Tämän jälkeen se uudelleenkäynnistää prosessit, jotta uudet asetukset tulevat voimaan.

WebGUI:n kautta käyttäjä pystyy myös esimerkiksi tarkistamaan järjestelmän ohjelmistokomponenttien ja laiteohjelmiston versiotiedot sekä käyttämään järjestelmää niin sanotun virtuaalisen HMI:n kautta, jossa laiteketjun yksiköiden operaattoripaneelit piirretään selainikkunaan ja käyttäjä voi ohjata niitä klikkailemalla painikkeita hiirellä.

5 HMI-ajurin arkkitehtuuri

Netcon 200 -tuoteperheen HMI-ajuri on Linux-käyttäjäavaruudessa toimiva C++-ohjelma, joka ylläpitää operaattoripaneelien toimintaa. Ajuri vastaanottaa käyttäjän syötteitä eli käytännössä eri nappien painalluksia ja suorittaa niiden ja järjestelmän senhetkisen tilan mukaisia toimenpiteitä. Se esimerkiksi toteuttaa painonapein annettuja kytkinkojeistojen ohjauksia kirjoittamalla ohjauskomentoja NFE:n hetkellistietokantaan sekä ohjaa paneelin indikaattoriledejä.

Ajuri on suunniteltu olio-ohjelmoinnin periaatteita noudattaen. Tämä on katsottu ajuria suunnitellessa parhaaksi toteutustavaksi monestakin syystä. Oliotyylinen abstraktio parantaa lähdekoodin ymmärrettävyyttä, tekee siitä modulaarisempaa ja ennen kaikkea helpottaa ajurin toiminnallisuuden laajentamista jatkossa. Viimeksi mainittu on erityisen tärkeä ominaisuus, koska Netcon 200 on suunniteltu monikäyttöiseksi ja moneen tarkoitukseen mukautettavaksi. On tärkeää, että ohjelmisto on kohtuullisella työmäärällä laajennettavissa tukemaan uusia HMI-paneeleita.

HMI-ajurin lähdekoodi jakautuu luokkiin, jotka kaikki kapseloivat sisäänsä jonkin konseptuaalisen kokonaisuuden. Tyyppihierarkia on rakennettu perintää hyödyntäen siten, että usein yhteisiä ominaisuuksia jakavat luokat perivät yhteisen rajapinnan, jolloin niitä voidaan käsitellä yhdessä polymorfismin keinoin. Tässä pääluvussa on kuvattu ajurin toiminnan kannalta keskeiset komponentit ja se, miten ne toimivat yhdessä. Luvun lopussa on myös kuvattu, miten ajuri viestii järjestelmän muiden osien kanssa ja miten ajurin sisäiset komponentit vuorovaikuttavat keskenään.

5.1 Käsittelijät

Ajurin tehtäviin kuuluu useita melko laajoja kokonaisuuksia, joiden suoritusvastuu on suunnitteluvaiheessa katsottu järkeväksi jakaa erilaisille käsittelijöille (*handler*). Käsittelijät ovat luokkia, jotka ohjaavat ja valvovat omaa osa-aluettaan ajurin toiminnassa ja niillä on yhteisessä rajapinnassa standardoitu tapa vastaanottaa informaatiota. Kuvassa 5 esimerkkejä käsittelijän alaluokista ovat paneelikäsittelijät (*panel handler*), kytkinkäsittelijä (*switch handler*), ledikäsittelijä (*led driver*) ja hallintamoodin käsittelijä (*control mode handler*).

Kaikki edellä mainitut käsittelijät periytyvät abstraktista luokasta HandlerBase, jonka rajapinta on kuvattu esimerkkikoodissa 4 hieman yksinkertaistettuna.

```
class HandlerBase {

public:
      HandlerBase();
      virtual ~HandlerBase();

      virtual void receiveEvent(int signalId, SignalReaderBase *reader) = 0;
      virtual void receiveMessage(ButtonMsg *msg);
};
```

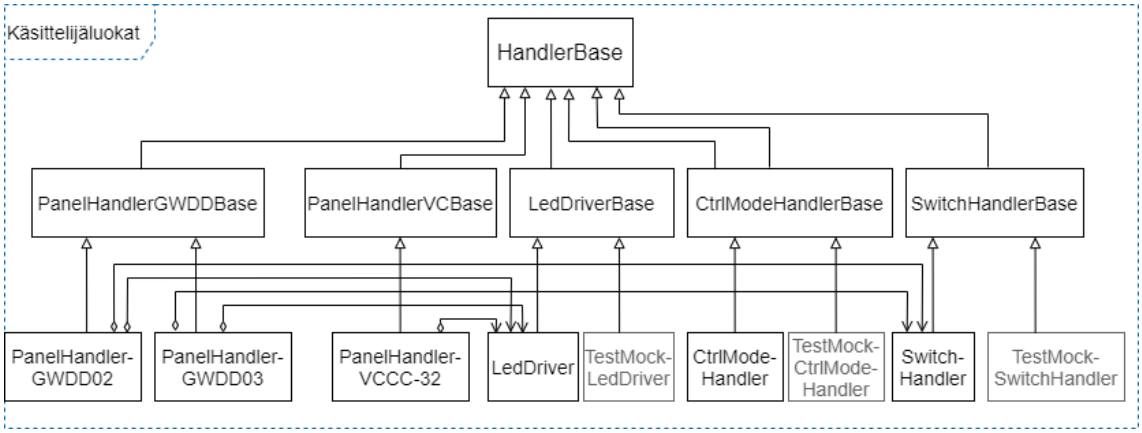
Esimerkkikoodi 4. HandlerBase-luokan rajapinta, yksinkertaistettu.

Esimerkkikoodi 4 havainnollistaa rajapinnan tärkeimmän piirteen ohjelman kokonaisuuden kannalta: kyvyn vastaanottaa dataa standardiformaatissa. Jokaisen käsittelijän on määriteltävä oma toteutuksensa *receiveEvent()*-funktiosta, jota se käyttää signaalien tilamuutoksien käsittelyyn. Käsittelijätyypeissä, jotka sisältävät monimutkaista logiikkaa, receiveEvent()-funktion sisään on upotettu kutsu käsittelijäluokassa määriteltyyn tilakonefunktioon. Tilakoneeseen pohjautuvalla toteutuksella on pyritty parantamaan paneelikäsittelijän koodin luettavuutta ja muokattavuutta.

Tarvittaessa käsittelijä voi myös määritellä itselleen *receiveMessage()*-funktion. Viimeksi mainittu on merkityksellinen ainoastaan paneelikäsittelijöille, koska se on tarkoitettu painikeviestien vastaanottamiseen. Sen toteutus on HandlerBase-luokassa määritelty tyhjäksi. Käsittelijät, jotka eivät tarvitse funktiota, jättävät sen määrittelemättä, jolloin ne perivät pääluokan tyhjän version.

Käsittelijät ovat pitkälti itsenäisiä yksiköitä. Ne suorittavat toimenpiteitä, jotka määräytyvät käsittelijän sisäisen tilan ja sen vastaanottaman datan perusteella ja valvovat itse näiden toimenpiteiden aiheuttamia seurauksia järjestelmässä. Niillä on kuitenkin keskinäisiä omistussuhteita. Esimerkiksi paneelikäsittelijä voi omistaa tietyn määrän kytkinkäsittelijöitä, joita se käyttää kytkinkäsittelijälle määritellyn rajapinnan avulla. Kaikki käsittelijätyypit on suunniteltu siten, että ne konfiguroivat itsensä olion luomisen yhteydessä. Käsittelijälle annetaan konstruktorin parametrina osoitin sitä itseään koskevaan XML-elementtiin, joka sisältää valtaosan käsittelijän konfiguraatiosta. Tätä varten jokaisen käsittelijän tulee toteuttaa tarvitsemansa jäsentelyfunktiot (*parser functions*). Suurin osa käsittelijän konfiguraatiosta liittyy sitä koskeviin signaaleihin ja komentoihin. Luettuaan signaalien ja komentojen nimet konfiguraatiostaan käsittelijä rekisteröi ne ajurin sisäiseen tietokantaan.

Kuva 6 havainnollistaa käsittelijäluokkien välistä omistusta ja perintää. Varsinaiseen tuotantokoodiin kuuluvat luokat on siinä merkitty mustin laatikoin ja yksikkötestien käyttämät näköisluokat harmaalla. Perintä on merkitty onttokärkisellä nuolella alaluokasta yläluokkaan. Omistussuhde (*aggregation*) on merkitty omistavasta luokasta omistettuun luokkaan osoittavalla avokärkisellä nuolella, jonka alkupäässä on ontto timanttikuvio.



Kuva 6. Käsittelijätyyppisten luokkien perintä- ja omistussuhteet UML-luokkakaaviona.

5.1.1 Paneelikäsittelijä

Paneelikäsittelijä (*panel handler*) on nimensä mukaisesti luokka, joka vastaa jonkin laiteketjun yksikön HMI-paneelin kokonaisuudesta. Paneelikäsittelijä seuraa hallitsemaansa paneeliin liittyviä signaaleja ja painikeviestejä, joiden tilaajaksi se on rekisteröitynyt ja kontrolloi niiden perusteella vastuullaan olevia kytkinkäsittelijöitä, ledejä, hälytysten tiloja, äänimerkkisummeria ja niin edelleen.

Kullekin paneelityypille toteutetaan oma HandlerBase-luokasta periytyvä alaluokkansa, joka kykenee lukemaan ja validoimaan itselleen tarkoitetun konfiguraatioelementin, rekisteröimään siinä määritellyt signaalit ja komennot tietokantaan sekä alustamaan oman ledikäsittelijänsä ja mahdolliset kytkinkäsittelijänsä. Luokassa toteutetaan myös kyseisen paneelityypin toimintaa ohjaava tilakonefunktio. Vaikka kaikissa paneelityypeissä on keskinäisiä eroja, monissa niistä on myös keskenään identtisiä toimintoja. Tämän takia käsittelijäluokkien suunnittelussa on pyritty hyödyntämään polymorfismia ja toteuttamaan paneelien jakama toiminnallisuus niille yhteisessä yläluokassa. Esimerkiksi GWDD-02- ja GWDD-03-paneelit eroavat toisistaan ainoastaan siten, että

jälkimmäisessä on käytetty osa laiteyksikön digitaalisista tuloista ja lähdöistä kahteen ylimääräiseen kytkimeen, jolloin vastaavasti geneerisiä tuloja ja lähtöjä on vähemmän kuin GWDD-02:ssa. Muilta osin logiikka on identtistä, joten yhteiset toiminnot on toteutettu luokassa *PanelHandlerGWDDBase*, josta taas periytyvät varsinaiset paneelikäsittelijäluokat *PanelHandlerGWDD02* ja *PanelHandlerGWDD03*. Samaa periaatetta on noudatettu VCCC-32-paneelin kanssa. *PanelHandlerVCBase*-yläluokassa määritellään perustoiminnallisuudet paneelille, joka voi sisältää yhdestä neljään jännite- ja virtamittausmoduuleita ja niihin liittyviä hälytyksiä ja indikaatioita. Yläluokasta perivät paneelit toteuttavat niiden pohjalta oman kokoonpanonsa. VCCC-32-paneelin osalta se tarkoittaa yhtä jännite- ja kolmea virtamittausmoduulia.

5.1.2 Kytkinkäsittelijä

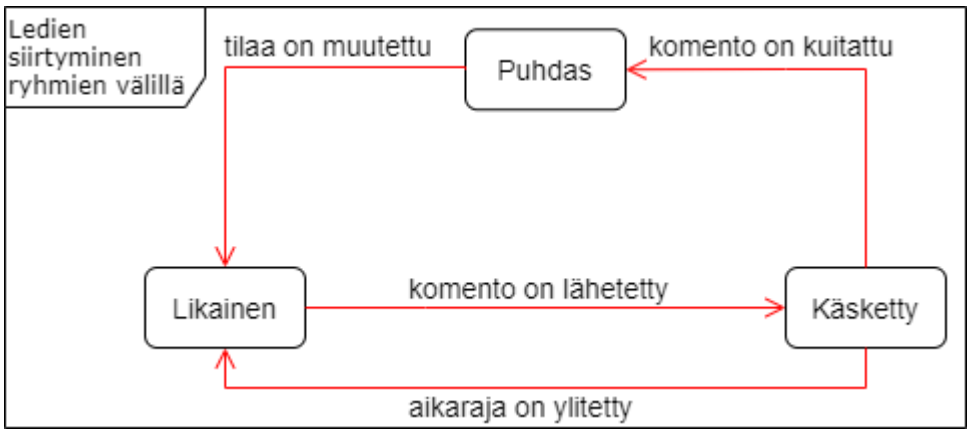
Tietyt paneelityypit kuten GWDD-02 ja GWDD-03 sisältävät ohjattavia kytkimiä, joten niitä ohjaava paneelikäsittelijäluokka omistaa jokaista kytkintä kohden kytkinkäsittelijän. Kytkinkäsittelijäluokka seuraa vastuullaan olevan kytkinkojeiston tilaa ja ilmoittaa tilamuutoksista paneelikäsittelijälleen. Kytkinkäsittelijän avulla paneelikäsittelijä voi myös lähettää ohjauskomentoja kytkimille. Kytkinkäsittelijä saa muiden käsittelijöiden tapaan konstruktorin parametrina osoittimen itseään koskevaan konfiguraatioelementtiin, josta se jäsentelee kytkimen tilaa indikoivan signaalin ja kytkimen ohjauskomennon nimet, luo niiden mukaiset signaali- ja komento-objektit ja rekisteröi ne tietokantaluokkaan. Tietokantaluokka on kuvattu tarkemmin luvussa 5.2.

Paneelikäsittelijä voi pyytää ohjauskomennon lähettämistä kytkinkäsittelijän setState()-funktiota kutsumalla. Kytkinkäsittelijän lähetettyä komennon se jää odottamaan kuittausta kytkimen tilamuutoksesta receiveEvent()-kutsun muodossa. Sen saatuaan se ilmoittaa muutoksesta paneelikäsittelijälle kutsumalla sen *receiveSwNotification()*-funktiota. Näin paneelikäsittelijä pystyy välittömästi reagoimaan muutokseen esimerkiksi päivittämällä kytkimen indikaatioledit osoittamaan uutta tilaa.

5.1.3 Ledikäsittelijä

Jokainen paneelikäsittelijä omistaa oman ledikäsittelijän, jolle se määrittelee ajurin alustuksen yhteydessä kaikki käyttämänsä ledit. Tämän jälkeen paneeli voi vaihtaa ledien tiloja käyttämällä ledikäsittelijän *setLedState()*-funktiota antamalla sille parametreina yksilöllisen leditunnisteen ja ledin uuden kohdetilan. Toiminto viimeistellään kutsumalla ledikäsittelijän *update()*-funktiota, jolloin varsinainen lediviesti lähtee kohdelaitteen käsiteltäväksi. Tämä mahdollistaa sen, että paneelin näkymä voidaan ”piirtää” valmiiksi asettamalla kaikkien muuttuvien ledien tilat, jonka jälkeen kaikki lediviestit upotetaan samaan viestikehykseen ja lähetetään kerralla.

Ledikäsittelijän logiikassa jokainen lediobjekti kuuluu yhteen kolmesta ryhmästä: puhtaisiin (*clean*), likaisiin (*dirty*) tai käskettyihin (*in transition*). Likaisten joukkoon merkitään ledit, joiden tilaa ledikäsittelijän tietorakenteissa on juuri muutettu setLedState()-funktiolla, eikä se enää vastaa ledin todellista tilaa. Puhtaiksi luetaan ne ledit, joiden tilaa ei ole muutettu sen jälkeen, kun se on viimeksi luettu NFE:n tietokannasta. Update()-funktiota kutsuttaessa käsittelijä käy läpi likaiset ledit ja lähettää niiden uuden halutun tilan mukaiset ledikomennot kyseiselle laiteyksikölle. Komentojen lähetyksen jälkeen ledit merkitään käsketyiksi siihen asti, että tilamuutoksille saadaan kuittaukset NFE:n REST-palvelimelta, jolloin ne voidaan merkitä puhtaiksi. Mikäli kuittausta ei saada tiettyyn aikarajaan mennessä, ledit merkitään taas likaisiksi ja kutsutaan update()-funktiota uudelleen, mikä aiheuttaa komentojen uudelleenlähetyksen. Kuvan 7 tilakaavio kuvaa, miten lediobjekti siirtyy eri ryhmien välillä.



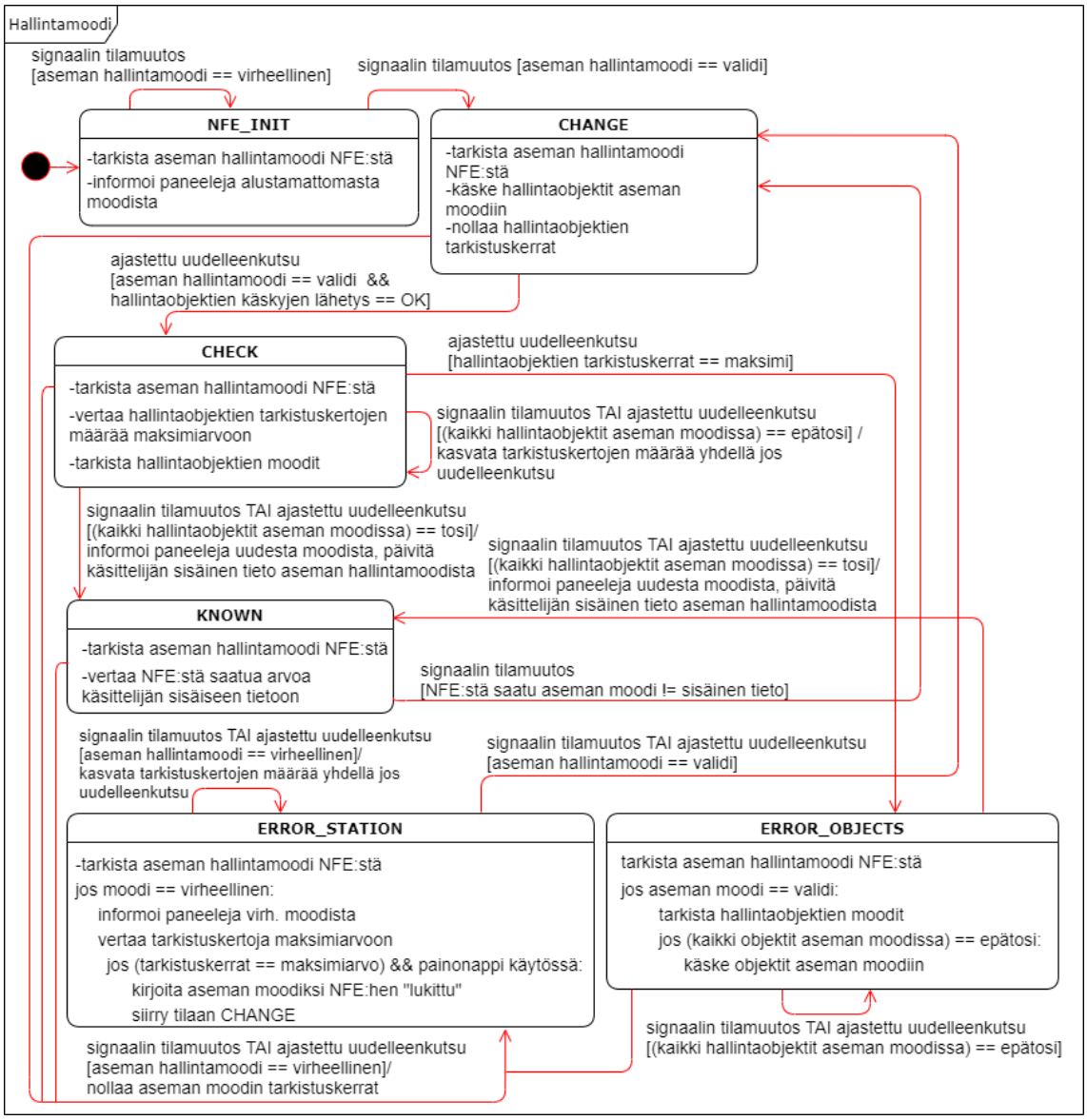
Kuva 7. Ledien siirtyminen ryhmien välillä. UML-tilakaavio.

Edellä mainitut lediryhmät on toteutettu linkitettyinä listoina, koska siten ledikäsittelijän koodissa on helppoa hakea kaikki tiettyyn ryhmään kuuluvat lediobjektit, ja ledien siirtäminen ryhmästä toiseen on tehokasta.

5.1.4 Hallintamoodin käsittelijä

Hallintamoodista vastaa hallintamoodin käsittelijä (*control mode handler*). Se on luokka, joka huolehtii siitä, että järjestelmä siirtyy pyydettäessä hallitusti moodista toiseen. Aseman hallintamoodi (*station control mode*) on koko järjestelmän laajuinen tila, joka määrittää sen, miten ketjun laitteissa oleville kytkimille voidaan antaa ohjauskomentoja. Mahdollisia hallintamoodeja on kolme: kaukokäyttö (*remote*)- ja paikalliskäyttötila (*local*) sekä lukittu tila (*locked*). Paikalliskäyttötilassa kytkinkomentoja vastaanotetaan ainoastaan HMI-paneelin kautta, kaukokäyttötilassa ainoastaan etäkomentoina SCADA-järjestelmästä. Lukitussa tilassa kytkimien ohjaus on estetty kokonaan. Hallintamoodi ei ole ainoastaan HMI-ajurin sisäinen konsepti. Jokaiselle järjestelmän kytkimelle on NFE:ssä datapiste, jolla asetetaan kytkimen yksilöllinen hallintamoodi em. vaihtoehtojen mukaisesti. Datapistettä ohjaamalla asetetaan kytkimeen liittyvä laiteohjelmistokerroksen (*firmware layer*) hallintaobjekti haluttuun moodiin. Jos kytkimelle yritetään antaa hallintamoodin vastaisia ohjauskomentoja, hallintaobjekti ei toteuta ohjauksia.

Yksittäisten kytkimien hallintamoodien lisäksi NFE:ssä on datapiste myös koko aseman hallintamoodille. Hallintamoodin käsittelijän tehtävä on seurata tätä datapistettä. Kun siinä tapahtuu muutos, käsittelijä lähettää kaikkien kytkimien hallintaobjekteille komennon siirtyä aseman moodia vastaavaan tilaan. Kun kaikkien hallintaobjektien tilan todetaan vastaavan uutta hallintamoodia, käsittelijä ilmoittaa siitä paneelikäsittelijöille lähettämällä niille sovelluksen sisäisen signaalin. Tällöin kukin paneeli voi suorittaa hallintamoodin edellyttämät toimenpiteet, kuten päivittää ledipaneelinsa tai lukita kytkinten ohjaukseen liittyvät painikkeet.



Kuva 8. Hallintamoodin käsittelijän UML-tilakaavio.

5.2 Tietokanta

Tietokanta (*database*) on HMI-ajurin luokka, joka ylläpitää listaa kaikista sovelluksen seuraamista signaaleista ja sen käyttämistä komennoista. Jokaisen käsittelijätyyppisen luokan konstruktori rekisteröi tietokantaan omat signaalinsa käyttäen tietokannan registerSignals()-funktiota. Funktio käy läpi rekisteröitävät signaalit ja lisää omaan kirjanpitoonsa ne, joita yksikään toinen käsittelijä ei vielä ole lisännyt. Se myös lisää

rekisteröivän käsittelijän kyseisen signaalin tilaajien (*subscribers*) joukkoon. Kun kaikki käsittelijät ovat rekisteröineet signaalinsa ja komentonsa, tietokanta suorittaa ajurin alustustoimenpiteiden mukaisesti käännöksen datapisteiden nimistä NFE-osoitteisiin, luo signaaleista tilausryhmän ja aloittaa ryhmän seuraamisen subpoll-palvelua käyttäen.

Kun tilausryhmän signaaleissa tapahtuu tilamuutoksia, subpoll-pyyntöön liittyvä HTTP-yhteys palauttaa vastausviestin, jonka jakelija välittää merkkijonona tietokannalle kutsumalla sen parseValues()-funktiota. Funktio jäsentelee viestistä yksittäiset, tilaa muuttaneet signaalit ja päivittää niiden arvot ja lippubitit omiin tietorakenteisiinsa. Tämän jälkeen se käy läpi jokaiseen muuttuneeseen signaaliin merkityt tilaajat ja kutsuu niiden receiveEvent()-funktiota. Myös kytkimien ohjauskomennot ja hallintamoodien asetuskomennot kulkevat tietokannan kautta ja ne rekisteröidään tietokantaan käyttäen *registerCommands()*-funktiota.

5.3 Signaaliobjekti

Jokaiseen ajurin seuraamaan NFE:n datapisteeseen liittyy tietokantaluokan ylläpitämä signaaliobjekti. Signaali (*signal*) on luokka, joka pitää sisällään datapisteen nimen, NFE-osoitteen, hetkellisen arvon ja lippubitit sekä listan signaalin tilaajista. Kaikki signaalityypit periytyvät SignalBase-luokasta, joka määrittelee paitsi yhteisen rajapinnan, myös oletustoteutukset rajapinnan funktioille. Näin perivät luokat määrittelevät ainoastaan tarvitsemansa, yläluokkaan kuulumattomat tai siitä toteutukseltaan poikkeavat funktiot. Esimerkiksi kuitattavan hälytyssignaalin *setValue()*-funktio eroaa oletustoteutuksesta, koska arvon päivittämisen lisäksi sen tulee käydä läpi tilaajien lista ja asettaa kuittaamisen tarvetta ilmaisevat lippumuuttujat. SignalBase-luokka on abstrakti, eli siitä itsestään ei voi luoda olioita. Varsinaiset signaaliobjektit luodaan yläluokasta periytyvistä aliluokista.

```
class SignalBase {
public:
      SignalBase();
      virtual ~SignalBase() = 0;

      virtual std::string getTag() const;
      virtual std::string getAddr() const;
      virtual std::string getValue() const;
      virtual std::string getFlags() const;
      virtual int setValue(const std::string &val);
      virtual int setFlags(const std::string &flags);
      virtual int addSubscriber(const Subscriber sub);
private:
      std::vector<Subscriber> subscribers;
};
```

Esimerkkikoodi 5. SignalBase-luokan rajapinta, yksinkertaistettu.

Signaalin ja sitä vastaavan NFE-tietokantapisteen yhdistävä tunniste on NFE-osoite. Tämän takia tietokanta ylläpitää signaalitaulukon lisäksi myös hajautustaulua (*hashmap*), jossa avaimet ovat merkkijonomuotoisia osoitteita ja arvot osoittimia signaaliobjekteihin. Näin subpoll-vastausviestejä jäsentelevä funktio pystyy tehokkaasti erottelemaan viestistä tilaa muuttaneiden pisteiden osoitteet ja hakemaan niitä vastaavat signaaliobjektit hajautustaulun avulla.

5.4 Tilaaja

On olemassa tilanteita, joissa useampi kuin yksi käsittelijä rekisteröi tietokantaan saman signaalin, jolloin ne kaikki saavat tiedon signaaliin liittyvistä tilamuutoksista. Kullakin käsittelijällä on signaalille oma yksilöllinen tunnisteensa (*signal id*), jonka avulla se receiveEvent()-kutsun saatuaan tietää, mikä signaali on kyseessä. Tämä tunniste on kuitenkin merkityksetön muille käsittelijöille. Niille tunniste saattaa joko viitata toiseen signaaliin tai sitä ei välttämättä ole edes määritelty. Tämän takia HMI-ajuriin on lisätty abstraktiokerros signaalin ja käsittelijän väliin. Se on toteutettu luokkana nimeltä tilaaja (*subscriber*). Tilaaja on tietorakenne, joka sisältää osoittimet käsittelijään ja signaalitulkkiin sekä käsittelijän määrittelemän signaalitunnisteen.

```
class Subscriber {
public:
      Subscriber(HandlerBase *handler, int signalId, sigreader_t reader);
      ~Subscriber();
private:
      HandlerBase *handler;
      SignalReaderBase *reader;
      int signalId;
friend class Database;
};
```

Esimerkkikoodi 6. Tilaaja-luokan rajapinta, yksinkertaistettu.

Kun käsittelijä luo uuden signaalin, se luo samalla uuden tilaajaobjektin lisäten siihen osoittimen itseensä sekä signaalitunnisteen, jolla se itse tunnistaa kyseisen signaalin. Ennen kuin käsittelijä rekisteröi signaalin tietokantaan, se lisää juuri luomansa tilaajaobjektin signaalin tilaajalistaan. Tietokannan rekisteröintifunktio tunnistaa useammin kuin kerran rekisteröidyt signaalit. Se ei tällöin luo niistä enää uusia signaaliobjekteja, vaan kopioi niiden tilaajat jo olemassa olevaan signaaliin.

Kun jonkin signaalin tila muuttuu, tietokanta käy läpi jokaisen sen tilaajalistassa olevan tilaajaobjektin ja kutsuu siihen merkityn käsittelijäobjektin receiveEvent()-funktiota käsittelijän itse määrittelemällä tunnisteella sekä signaalitulkilla. Tietokantaluokka on määritelty tilaajaluokan ”ystäväksi” *friend*-avainsanalla, joten se kykenee lukemaan ja kirjoittamaan tilaajaluokan yksityisiä muuttujia:

```
for(auto sub = sig.subscribers.begin(); sub != sig.subscribers.end(); ++sub) {
          sub->handler->receiveEvent(sub->signalId, sub->reader);
}
```

Esimerkkikoodi 7. Tilaajaobjektin käyttö signaalin tilan muuttuessa.

5.5 Signaalitulkki

Paitsi että käsittelijöillä on signaaleilleen yksilölliset tunnisteet, niillä voi olla myös toisistaan poikkeavia tapoja tulkita tietyn signaalin tilaa. Otetaan esimerkiksi digitaaliseksi tuloksi määritelty GPIO-signaali, joka NFE:n tietokannassa voi saada arvon 1 (pois päältä) tai 2 (päällä). Oletetaan, että kyseisen signaalin on rekisteröinyt tietokantaan kaksi käsittelijää. Toisen käsittelijän konfiguraatiossa on signaalin kohdalla merkintä, että sitä

käsitellään käänteisenä (*inverted*). Tälle käsittelijälle siis arvo 1 tarkoittaa, että signaali on päällä, ja arvo 2 tarkoittaa, että se on pois päältä.

Käsittelijän näkökulmasta on kuitenkin toivottavaa, että se voi käsitellä signaaleitaan geneerisesti, ilman, että sen koodissa tarvitsisi huomioida yksityiskohtia kuten yksittäisen signaalin käänteisyys. Tämän vuoksi signaaliobjektia luodessaan käsittelijä luo samalla signaalin lukemiseen tarkoitetun signaalitulkin (*signal reader*). Signaalitulkki on luokka, joka liitetään tiettyyn signaaliin osoittimen avulla. Tulkki sisältää tarvittavat jäsenfunktiot arvon lukemiseksi signaaliobjektista. Näin arvojen lukemiseen liittyvät yksityiskohdat voidaan kapseloida signaalitulkin sisälle, ja käsittelijän koodi yksinkertaistuu.

```
enum binState {
      off  = 1,
      on   = 2
};
class SignalReaderBase {
public:
      SignalReaderBase();
      virtual ~SignalReaderBase();
      virtual void setSignal(const SignalBase *sig) = 0;
      virtual int getInt();      /*Palauttaa signaalin arvon kokonaislukuna*/
      virtual float getFloat();  /*Palauttaa signaalin arvon liukulukuna*/
      virtual binState getBin(); /*Palauttaa päällä/pois -arvon*/
      virtual int getFlags();    /*Palauttaa signaalin lippubitit*/
};
class SignalReader : public SignalReaderBase {
public:
      SignalReader(bool inv = false) : invert(inv) {}
      ~SignalReader() {}

      void setSignal(const SignalBase *s) {/*Alaluokalle spesifiä koodia*/}
      binState getBin() { /*Palauttaa käänteisen arvon jos invert = tosi*/ }
private:
      const bool invert;
};
/*Esimerkki SignalReader -luokan käytöstä*/
Signal sharedSig;
SignalReader reader1;
reader1.setSignal(&sharedSig);
SignalReader reader2(true);
reader2.setSignal(&sharedSig);

sharedSig.setValue("1");
reader1.getBin();    /*Palauttaa arvon off*/
reader2.getBin();    /*Palauttaa arvon on*/
```

Esimerkkikoodi 8. Signaalitulkin toiminta.

5.6 Komento-objekti

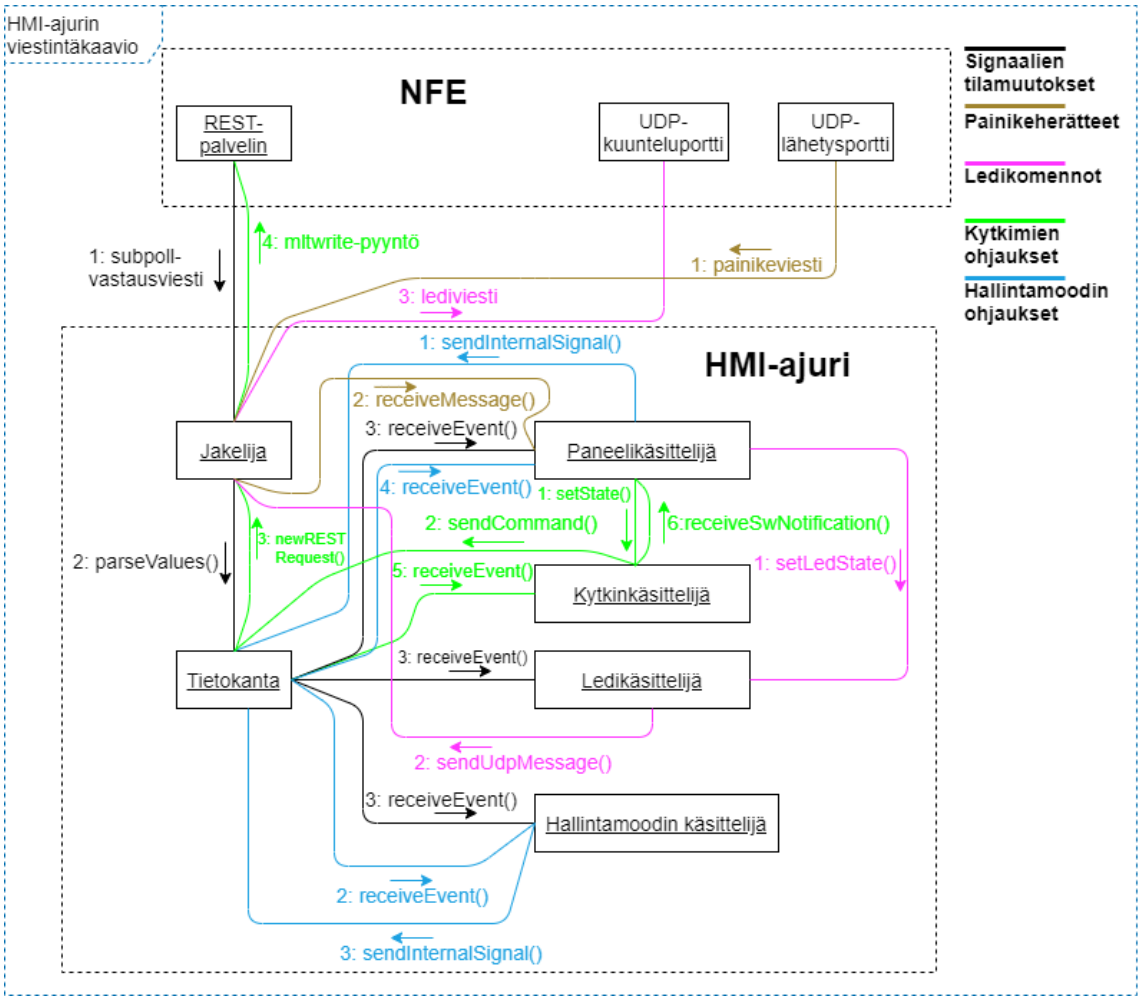
Edellä kuvatut signaaliobjektit on tarkoitettu ulkopuolelta HMI-ajurille tulevan datan käsittelyyn. Toiseen suuntaan liikkuvan datan, eli ajurin lähettämien komentojen käsittelyä varten ajurissa on lähes vastaavalla tavalla, mutta käänteiseen suuntaan, toimiva komentoluokka (*command*). Se sisältää tilaajien listan sijaan julkaisijoiden listan (*publishers*) ja siihen kirjoitetaan arvoja komentokirjoittimella (*command writer*). Signaalien tapaan on siis mahdollista, että tiettyä komentoa voi käyttää useampi käsittelijä.

5.7 Jakelija

Jakelija (*dispatcher*) on komponentti, joka ylläpitää ajurin viestiyhteyksiä muuhun järjestelmään. Se sisältää koodin, joka ylläpitää Libcurl-kirjaston avulla HTTP-yhteyksiä NFE:n REST-rajapintaan sekä funktiot ledi- ja summerikomentojen lähettämiseen ja painikeviestien vastaanottamiseen. Jokaisen viestejä vastaanottavan käsittelijän on konstruktorissaan kutsuttava jakelijan *registerCommAddr()*-funktiota, jolloin jakelija tallentaa käsittelijän IEC-osoitteen (*common address*) ja osoittimen käsittelijäobjektiin omaan sisäiseen hajautustauluunsa. Viestin saapuessa jakelija jäsentelee viestin otsikkodatasta vastaanottavan laiteyksikön osoitteen, tekee sen perusteella haun hajautustauluun ja osuman löytäessään kutsuu osoitetta vastaavan käsittelijän receiveMessage() -funktiota antaen sille parametrina viestin sisältämän datan.

5.8 HMI-ajurin viestintä

HMI-ajurilla on kaksi tapaa lähettää ja vastaanottaa informaatiota toimintaympäristössään: IEC 60870-5-101 -tiedonsiirtostandardin mukaan muotoillut, Ethernet-väylällä liikkuvat datapaketit sekä NFE:n ylläpitämä, HTTP-liikenteeseen perustuva, REST-rajapinta, joka on kuvattu tarkemmin luvussa 4.



Kuva 9. HMI-ajurin viestintäkaavio.

Netcon 200 -järjestelmään kuuluvien laiteyksiköiden välinen kommunikaatio on toteutettu IEC 60870-5-101 -standardin määrittelemällä sovelluskerroksen rakenteella, kuitenkin niin, että fyysisen- ja siirtoyhteyskerroksen toiminnallisuus on toteutettu Ethernetin avulla. Netcon 200 -laiteketjun jokaisen laitteen emolevy sisältää Ethernet-kytkimen, joka ohjaa kunkin Ethernet-kehyksen sen otsikossa (*header*) määriteltyyn MAC-osoitteeseen (*Media Access Control*). Ethernet-kehykseen upotetaan varsinainen IEC-protokollan mukainen PDU (*Protocol Data Unit*). NFE:n sisäänrakennettu IEC 60870-5-101 -protokolla-ajuri toimii välikappaleena tämän paljaan Ethernet-liikenteen ja gateway-kortin Linux-käyttäjäavaruudessa toimivien sovellusten välillä. Se ylläpitää tietokoneen paikallisessa IP-osoitteessa (*localhost*) kahta UDP-porttia (*User Datagram Protocol*). Toinen niistä on kuunneltava portti, johon sovellukset voivat lähettää viestejä. NFE riisuu viestien UDP-kehyksen sisältä datasisällön eli IEC-standardiformaatissa olevan binäärimuotoisen

datan, etsii sisäisestä taulukostaan viestissä kuvattua laiteyksikön kohdeosoitetta (*Common Address*) vastaavan MAC-osoitteen, upottaa viestin Ethernet-kehykseen ja lähettää sen kyseiseen MAC-osoitteeseen. Vastakkaiseen suuntaan laiteyksiköiltä NFE:lle saapuvat viestit upotetaan UDP-datagrammiin ja lähetetään toiseen edellä mainituista porteista, jota kuuntelemalla sovellukset voivat vastaanottaa niitä. Laiteyksiköille menevien ja niiltä saapuvien viestien on kuljettava NFE:n kautta, koska NFE toimii IEC 60870-5-101 -väylän isäntänä (*master*), joten ainoastaan se pystyy kommunikoimaan suoraan renkimoduulien kanssa.

HMI-ajuri käyttää IEC-viestistandardia paneelien indikaattoriledien ohjaukseen. Se myös vastaanottaa painonappitapahtumiin liittyvät herätteet IEC-viesteinä. Viestit koostuvat binääridatasta, jonka 4 ensimmäistä oktettia muodostavat datayksikkötunnisteen (*Data Unit Identifier*), jota seuraa valinnainen määrä informaatio-objekteja (*Information Object*). Datayksikkötunnisteesta käyvät ilmi viestin tyyppi, informaatio-objektien lukumäärä viestissä, viestin syy sekä laiteyksikön osoite (*Common Address*). Taulukossa 2 on kuvattu tyyppiä 0x8B (*HMI control message*) oleva viesti, joka sisältää ainoana informaatio-objektinaan yhden ledikomennon. Komento on osoitettu laiteyksikölle osoitteessa 0x11, ja sen sisällä ledille, joka sijaitsee osoitteessa 0x1234AA. Ledin asetusarvoksi on annettu 0x1, joka järjestelmän spesifikaatiossa tarkoittaa käskyä kytkeä ledi päälle. QOS-kentän arvoksi on merkitty 0x0 (*execute*), jolloin komento toteutetaan välittömästi.

Taulukko 2. Esimerkki ledikomennon sisältävästä IEC-standardin mukaisesta viestistä.

1	0	0	0	1	0	1	1	Tyyppitunniste (*type identification*): 0x8B	DATA UNIT IDENTIFIER
0	0	0	0	0	0	0	1	Informaatio-objektien lukumäärä (*variable structure qualifier*): 0x1	
0	0	0	0	0	1	1	0	Syy (*cause of transmission*): 0x6	
0	0	0	1	0	0	0	1	Laiteyksikön osoite (*Common Address*): 0x11	
1	0	1	0	1	0	1	0	Informaatio-objektin osoite (vähiten merkitsevä oktetti): 0xAA	INFORMATION OBJECT
0	0	1	1	0	1	0	0	Informaatio-objektin osoite (keskimmäinen oktetti): 0x34	
0	0	0	1	0	0	1	0	Informaatio-objektin osoite (eniten merkitsevä oktetti): 0x12	
0	0	0	0	0	0	0	1	Asetusarvo, vähiten merkitsevä oktetti: 0x1	
0	0	0	0	0	0	0	0	Asetusarvo, eniten merkitsevä oktetti: 0x0	
0	0	0	0	0	0	0	0	Laatu (*qualifier of set-point command, QOS*): 0x0	

5.9 Käytetyt kirjastot

Tässä alaluvussa käsitellään HMI-ajurin käyttämiä Libevent-, Libcurl- ja Libxml2-ohjelmakirjastoja. Kirjastoista Libcurl ja Libxml2 on linkitetty HMI-ajuriin dynaamisesti, eli ne eivät ole kiinteä osa HMI-ajurin ohjelmabinääriä. Ne ovat samanaikaisesti myös muiden järjestelmän sovellusten käytettävissä. Libevent taas on ainakin kirjoitushetkellä linkitetty staattisesti HMI-ajuriin, koska muut Netcon 200 -järjestelmän sovellukset eivät tarvitse sitä, eikä sitä siksi ole asennettu järjestelmään jaettuna kirjastona.

5.9.1 Libevent

HMI-ajuri toimii tapahtumapohjaisesti. Tämä tarkoittaa sitä, että normaalitilassa, kun ulkoisia syötteitä ei ole, ohjelma tekee aktiivisesti hyvin vähän tai ei mitään. Ajuri reagoi muutamaan eri herätetyyppiin: painonappien tilamuutoksista informoiviin IEC-standardin mukaisiin viesteihin, omiin HTTP-pyyntöihinsä saapuviin vastauksiin, jotka indikoivat muutoksia signaalien tiloissa, ajurin eri komponenttien toisilleen lähettämiin sisäisiin signaaleihin sekä erityyppisten ajastimien laukeamiseen.

Toteutuksessa on käytetty apuna Libevent-nimistä C-kirjastoa (tyyliteltynä *libevent*), joka tarjoaa valmiin infrastruktuurin tapahtumien käsittelylle. Libevent on melko hyvin dokumentoitu kirjasto, ja sen referensseinä mainittakoon esimerkiksi Chromium-selain, joka käyttää kirjastoa Linux- ja Mac-versioissaan sekä tmux (terminal multiplexer) -sovellus.

Libevent toimii käytännössä siten, että kirjaston rajapintaa käyttäen luodaan *event*-tyyppinen tapahtumaobjekti. Objektiin liitetään tietty tiedostokahva, tapahtumakäsittelijäfunktio (*callback function*) joka suoritetaan tapahtuman yhteydessä sekä tieto siitä, millaiset tapahtumat tiedostokahvassa laukaisevat tapahtumakäsittelijän (luku/kirjoitus/ajastimen laukeaminen). Libeventin kanssa on käytettävä asynkronisia yhteyskahvoja (*non-blocking sockets*), joiden seuraamiseen se käyttää kohdekoneen käyttöjärjestelmästä riippuen eri järjestelmäkutsuja, kuten *poll*, *epoll* tai *kqueue*, toimien tavallaan abstraktiokerroksena niiden päällä.

Libeventin ohjelmointirajapinta on kohtuullisen intuitiivinen ja helppokäyttöinen. Keskeisenä komponenttina toimii niin sanottu tapahtumakanta (*event base*) eli tietorakenne, joka pitää kirjaa siihen lisätyistä tapahtumaobjekteista ja seuraa niihin liittyviä tiedostokahvoja. Tapahtumakannan luonti tapahtuu *event_base_new()*-funktiolla, joka palauttaa osoittimen heap-muistiin luotuun objektiin:

```
struct event_base *evbase = event_base_new();
```

Tämän jälkeen tapahtumakantaan liitetään tapahtumia. Tämä tehdään HMI-ajurin tapauksessa käyttäen *event_assign*()-funktiota. Esimerkkikoodista 8 nähdään, että funktio ottaa parametreina osoittimet alustamattomaan tapahtumaobjektiin (*example_event*) ja juuri luotuun tapahtumakantaan (*evbase*)**,** tiedostokahvan numeron (*sock*), kokonaisluvun, joka koostuu erilaisista tapahtumaobjektiin liittyvistä lippubiteistä (*EVREAD* ja *EV_PERSIST*), funktio-osoittimen tapahtumakäsittelijänä toimivaan callback-funktioon (*example_cb*) sekä void-tyyppisen osoittimen *arg*, jolla ohjelmoija voi tuoda tapahtumakäsittelijään vapaamuotoisen parametrin.

```
#include “event2/event.h”
#include <iostream>
#include <string>

void example_cb(evutil_socket_t fd, short what, void *arg)
{
      /*Tämä funktio suoritetaan, kun tapahtuma laukeaa*/
      std::string str((char*) arg);
      std::cout << str << std::endl;
}

struct event_base *evbase = event_base_new();
struct event example_event;
evutil_socket_t sock = 0; /*Stdin-syöte*/

event_assign(&example_event, &evbase, sock, EV_READ | EV_PERSIST, example_cb,
(void*) “Tapahtuma laukesi”);
```

Esimerkkikoodi 8. Libeventin perustoiminnallisuus.

Esimerkissä kuvattujen funktiokutsujen jälkeen tapahtuma on alustettu (*initialized*). Tässä vaiheessa tiedostokahvan tilassa tapahtuvat muutokset eivät vielä kuitenkaan laukaise tapahtumia. Tätä varten tapahtuma on vietävä odottava-tilaan (*pending*) kutsumalla *event_add()*-funktiota, joka ottaa parametrina osoittimen tapahtumaobjektiin, ja vapaaehtoisena toisena parametrina *timeval*-tyyppisen aikarajan, jonka kuluttua

tapahtuma viimeistään laukeaa, *EV_TIMEOUT*-bitillä varustettuna. Aikarajan voi kuitenkin ohittaa asettamalla parametrin arvoksi *NULL*:

```
event_add(&example_event, NULL);
```

Tapahtumien alustamisen ja lisäämisen jälkeen Libeventille on ilmoitettava, että se voi alkaa seuraamaan tapahtumakantaan listattujen tapahtumien tiedostokahvoja. Tämä tapahtuu kutsumalla *event_base_dispatch()*-funktiota, joka ottaa parametrina osoittimen tapahtumakantaan:

```
event_base_dispatch(&evbase);
```

Yllä mainittu funktiokutsu aloittaa tapahtumasilmukan, jossa tapahtumien lauetessa niihin liittyviä callback-funktioita kutsutaan, ja jos mitään tapahtumia ei tule, ohjelma ei tee käytännössä mitään. Funktiosta palataan vasta, kun jonkin tapahtumakäsittelijän sisältä kutsutaan esimerkiksi *event_base_loopbreak*()-funktiota.

5.9.2 Libcurl

NFE:n REST-rajapintaa käytetään HTTP-protokollan avulla, jolle tarvitaan HMI-ajurissa tuki. Tähän tarkoitukseen käytetään cURL-ohjelmistoprojektiin kuuluvaa Libcurl-kirjastoa (tyyliteltynä *libcurl*). Libcurl on laajasti käytössä oleva, monia eri protokollia tukeva ja kohtuullisen kattavasti dokumentoitu URL-tiedonsiirtokirjasto, joka tarjoaa tarvittavat funktiot ja tietorakenteet esimerkiksi HTTP-yhteyksien hallintaan.

Libcurlin toiminnallisuus on jaoteltu niin sanottuihin *easy*- ja *multi*-rajapintoihin. Easy-rajapintaa voidaan ajatella kirjaston peruskomponenttina, ja se tarjoaa keinot yksittäisen HTTP-yhteyden käsittelyyn. Yksittäistä HTTP-pyyntöä edustaa *CURL*-tyyppinen objekti, jota Libcurlin terminologiassa kutsutaan yhteyskahvaksi (*easy_handle*). Yhteyskahvan luominen tapahtuu *curl_easy_init()*-funktiolla.

Kahvan luomisen jälkeen se on konfiguroitava asettamalla sille optioita *curl_easy_setopt()*-funktiolla. Ainoa pakollinen optio on kohde-URL:n määrittelevä *CURLOPT_URL,* mutta yleensä on tarkoituksenmukaista määritellä myös esimerkiksi osoitin callback-

funktioon *(CURLOPT_WRITEFUNCTION),* jota käytetään saapuvan datan tallennukseen, sekä POST-pyyntöä käytettäessä myös POST-kentän liitedata optiolla *CURLOPT_POSTFIELDS*.

Kun kaikki tarvittavat optiot on asetettu, yhteys avataan funktiolla *curl_easy_perform()*.

```
#include <string>
#include <curl/curl.h>

std::string msg_buf;

/*Tallennusfunktion tulee noudattaa tätä prototyyppiä*/
size_t write_cb(char *data, size_t size, size_t nmemb, void *userdata) {
    msgbuf += std::string(data);
    return size;
}


CURL *handle = curl_easy_init();
if(!handle) {
    exit(-1); /*Jos paluuarvo on NULL, jotain meni vikaan*/
}

curl_easy_setopt(handle, CURLOPT_URL, "http://localhost:1234/mypostservice");
curl_easy_setopt(handle, CURLOPT_WRITEFUNCTION, write_cb);
curl_easy_setopt(handle, CURLOPT_POSTFIELDS, "Tärkeitä parametrejä");

CURLcode res = curl_easy_perform(handle);
if(res != CURLE_OK) {
    exit(-2); /*Funktio palautti virhekoodin*/

}
```

Esimerkkikoodi 9. Libcurlin easy-rajapinnan perustoiminnallisuus

Easy-rajapinta on synkroninen, eli siihen liittyviä funktioita kutsuttaessa ohjelma jää odottamaan, kunnes funktio palautuu. Tämän takia se soveltuu sellaisenaan heikosti HMI-ajuriin, jonka tarvitsee usein hallinnoida useita HTTP-yhteyksiä samanaikaisesti. Siksi Libcurlin multi-rajapinta soveltuu ajurin käyttöön paremmin.

Multi-rajapinta sisältää nimensä mukaisesti työkalut usean samanaikaisen yhteyden ylläpitämiseen, ja siihen kuuluu olennaisena osana *CURLM*-objekti eli moniyhteyskahva (*multi handle*). Moniyhteyskahva on tietorakenne, johon käyttäjä voi tiettyjä funktioita käyttäen lisätä useita HTTP-yhteyksiä easy handle -kahvojen muodossa, käynnistää niiden tiedonsiirron sekä tiedustella kunkin yhteyden tilaa nähdäkseen, onko sen

tiedonsiirto jo valmistunut. Moniyhteyskahvan avulla ohjelmoijan on helpompi käsitellä useaa samanaikaista yhteyttä ilman tarvetta luoda ohjelmaan useampia säikeitä (*threads*).

Vielä merkittävämpi tekijä multi-rajapinnassa HMI-ajurin kannalta on, että se mahdollistaa tapahtumapohjaisen arkkitehtuurin yhteistyössä ohjelmoijan haluaman tapahtumakirjaston, tässä tapauksessa Libeventin kanssa. Tapahtumapohjaista mallia käytettäessä ohjelmoija asettaa kokoamalleen moniyhteyskahvalle optioina kaksi callback-funktiota. CURLMOPT_SOCKETFUNCTION-funktio on Libcurlin tapa raportoida sovellukselle moniyhteyskahvan yhteyksiin liittyvistä tiedostokahvoista. CURLMOPT_TIMERFUNCTION-funktio on puolestaan kirjaston keino asettaa yhteyskahvoihin liittyviä ajastimia. Ohjelmoijan tulee itse määritellä molemmat näistä funktioista.

Käytettäessä Libcurlia tapahtumapohjaisessa sovelluksessa sovelluksen tulee itse seurata kirjaston käyttämiä tiedostokahvoja valitsemallaan tavalla ja reagoida niiden tilassa tapahtuviin muutoksiin kutsumalla *curl_multi_socket_action()*-funktiota ja antamalla sille parametreina osoitin moniyhteyskahvaan *multi_handle*, muutoksen kohteena olevan tiedostokahvan numero *sockfd* sekä lippubittimuuttuja *ev_bitmask,* joka kertoo tapahtuman tyypin eli sen, onko ko. tiedostokahva valmis lukemista tai kirjoittamista varten vai onko siihen liittyvä aikaraja umpeutunut. Lisäksi neljäntenä parametrina annetaan osoitin kokonaislukuun *running_handles*, johon Libcurl funktiosta poistuessaan tallentaa niiden yhteyskahvojen määrän, joissa tiedonsiirto on vielä kesken.

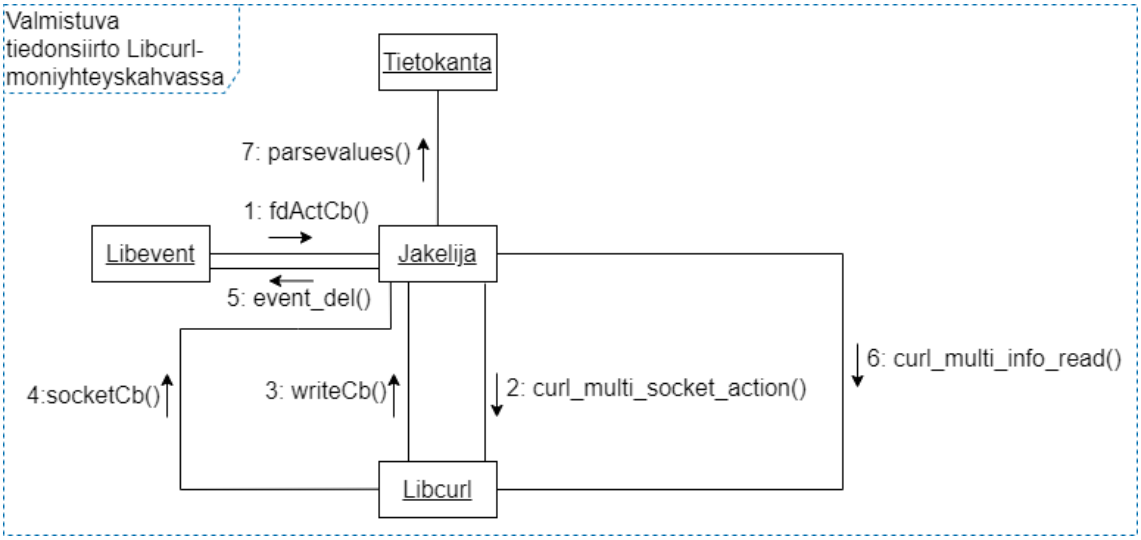
```
CURLMcode curl_multi_socket_action(CURLM * multi_handle, curl_socket_t sockfd,
int ev_bitmask, int *running_handles);
```

Esimerkkikoodi 10. Curl_multi_socket_action() -funktion signatuuri.

Yhteistyö Libeventin kanssa toimii pääpiirteissään siten, että kutsuttaessa *curl_multi_socket_action()* -funktiota Libcurl tekee tilanteesta riippuen yhden, useamman tai ei yhtään kutsua CURLMOPT_SOCKETFUNCTION -optiolla määritellylle callback-funktiolle *socketCb()*. Kyseinen funktio on HMI-ajurin tapauksessa määritelty niin, että se joko luo parametrina annettua tiedostokahvaa koskevan Libevent-tapahtuman *event_assign()*-funktiolla ja siirtää sen pending-tilaan *event_add()*-funktiota kutsumalla,

tai poistaa tiedostokahvaan liittyvän tapahtuman Libeventin kirjanpidosta mikäli Libcurl ilmoittaa, ettei enää käytä sitä.

Kun Libevent huomaa, että tiedostokahva muuttuu lukukelpoiseksi, se kutsuu tiedostokahvatapahtumille määriteltyä callback-funktiota *fdActCb(),* joka puolestaan kutsuu *curl_multi_socket_action()*-funktiota informoidakseen Libcurlia muutoksesta, jolloin kirjasto suorittaa muutoksen edellyttämät luku- tai kirjoitustoimenpiteet. Aina dataa vastaanottaessaan Libcurl kutsuu tallennusta varten määriteltyä callback-funktiota *writeCb()*, joka kirjoittaa datan kyseiselle yhteyskahvalle omistettuun datapuskuriin. Jos HMI-ajurin jakelijaluokka toteaa *curl_multi_socket_action()*-kutsun palautuessa, että *running_handles*-muuttujan arvo on pienentynyt, se kutsuu *curl_multi_info_read()*-funktiota selvittääkseen, minkä yhteyden tiedonsiirto on valmistunut. Tämän jälkeen se voi välittää palvelimen vastauksen tietokantaluokalle kutsumalla sen *parseValues()*-funktiota. Kuvan 10 kommunikaatiodiagrammi havainnollistaa komponenttien välistä vuorovaikutusta edellä kuvatussa tilanteessa, jossa tiedostokahvaan saapuu sisääntulevaa dataa ja Libcurl toteaa yhteyden tiedonsiirron valmistuneen.



Kuva 10. Valmistuva tiedonsiirto Libcurl-moniyhteyskahvassa. UML-kommunikaatiodiagrammi.

5.9.3 Libxml2

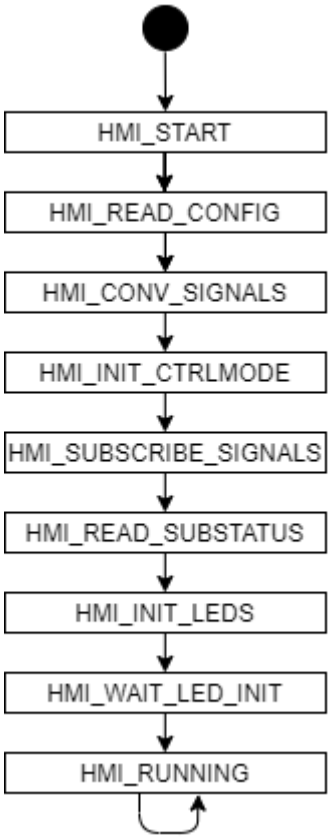
Libxml2 on alun perin Gnome-työpöytäympäristön kehittäjien laatima, XML-merkintäkielen jäsentelyyn tarkoitettu C-kielinen kirjasto. HMI-ajurin konfiguraatiotiedosto on XML-kielinen, ja Libxml2-kirjastoa on sovelluksessa käytetty sen sisältämän informaation jäsentelyyn. Kirjasto abstraktoi XML-kielen syntaksiin liittyvät yksityiskohdat helpommin hahmotettavan ohjelmointirajapinnan taakse. Esimerkkikoodin 11 testiohjelma lataa muistiin XML-tekstitiedoston ”*configuration.xml*” ja käy for-silmukassa läpi suoraan dokumentin juuritason alla olevat elementit. Jos ohjelma löytää elementin nimeltä *hostname*, se tulostaa elementin sisältämän tekstin näytölle.

```
xmlDoc *doc = xmlReadFile("configuration.xml", NULL, 0);
xmlNode *root = xmlDocGetRootElement(doc);

for (xmlNode *child = root->children; child; child = child->next) {
     if(xmlStrEqual(child->name, (const xmlChar *) "hostname")) {
          const char *elem =
          (const char*) xmlNodeListGetString(doc, child->xmlChildrenNode, 1);

          if(elem && strlen(elem) > 0) {
               printf("%s\n",  elem);
          }
     }
}
```

Esimerkkikoodi 11. Libxml2-kirjaston perustoiminnallisuus.

5.10 HMI-ajurin käynnistyminen

HMI-ajuri suorittaa main()-funktion sisällä ainoastaan välttämättömät alustustoimenpiteet (komentoriviparametrien jäsentely, tietorakenteiden luominen ja alustaminen). Tämän jälkeen se siirtyy päätason tilakoneeseen, joka määrittelee toimenpiteet, jotka ajurin on käynnistyttyään suoritettava siirtyäkseen normaaliin operointitilaan. Tilakone on käytännössä toteutettu globaalissa nimiavaruudessa sijaitsevana funktiona. Tilakonetta varten luodaan erityinen Libevent-ajastin, jolle annetaan parametrina tilakonefunktioon viittaava funktio-osoitin. Tämä mahdollistaa sen, että kun kulloisenkin tilan vaatimat toimenpiteet (ja mahdollinen tilasiirtymä) on suoritettu, voidaan myös aikatauluttaa tilakonefunktion seuraava kutsu.



Kuva 11. HMI-ajurin päätason tilakoneen tilasiirtymät

Kuvasta 11 nähdään ajurin käynnistyessään suorittamat toimenpiteet alkaen siitä, kun tilakonetta kutsutaan ensimmäisen kerran aina siihen asti, kun ajuri siirtyy tarvittavien alustustoimien jälkeen normaaliin operointitilaan. Tilakone on toiminnaltaan yksinkertaisin mahdollinen, koska jokaisesta tilasta on ainoastaan yksi mahdollinen siirtymä toiseen tilaan.

Tilassa HMI_READ_CONFIG ladataan keskusmuistiin HMI-ajurin XML-konfiguraatio joko komentoriviparametrina annetusta tiedostosta, tai ennalta määritellystä oletussijainnista, mikäli ajuri on käynnistetty ilman kyseistä parametria.

Konfiguraation pohjalta luodaan instanssit ajurin tarvitsemista luokista. Tällaisia ovat esimerkiksi jakelija-, tietokanta- ja hallintamoodin käsittelijäobjektit, tarvittavat paneelikäsittelijät, kytkinkojeistojen käsittelijät ja ledikäsittelijät riippuen siitä, millaisella konfiguraatiolla ajuri on käynnistetty.

HMI_CONV_SIGNALS-tilaan päästyään ohjelma suorittaa käännöksen konfiguraatiotiedostosta lukemistaan datapisteiden nimistä NFE-osoitteiksi käyttäen REST-rajapinnan *mltconv* -palvelua. HMI_INIT_CTRLMODE-tilassa ajuri tarkistaa NFE:n tietokannasta, onko aseman hallintamoodi asetettu. Jos sille ei ole asetettu arvoa, ajuri kirjoittaa tietokantaan oletusarvona etäkäyttömoodin.

Siirryttyään HMI_SUBSCRIBE_SIGNALS-tilaan ohjelma kutsuu *subreg*-palvelua antaen sille parametrina konvertointivaiheessa saadut NFE-osoitteet. NFE muodostaa tällöin pyydetyistä signaaleista tilausryhmän ja palauttaa ryhmän tunnisteen HMI-ajurille. HMI_READ_SUBSTATUS-tilassa ohjelma kutsuu substatus-palvelua antaen sille parametrina tilausryhmän tunnuksen. Palvelu palauttaa HMI-ajurille kaikkien tilausryhmän sisältämien signaalien senhetkiset arvot, joiden avulla ajuri alustaa omassa tietokannassaan olevat signaaliobjektit. Tämän jälkeen ryhmän tilaa seurataan ainoastaan subpoll-pyynnöillä, jotka palauttavat arvoja ainoastaan siinä tapauksessa, että yhden tai useamman seurannassa olevan signaalin tila on muuttunut.

HMI-paneelien ledien ohjaaminen on niin kriittinen osa sovelluksen toimintaa, että ledien alustamiselle on myös päätason tilakoneessa erillinen tila. Ledipaneelit saattavat ajurin käynnistyessä olla missä tahansa satunnaisissa tiloissa, joten osana alustustoimenpiteitään ajuri lukee ledien senhetkiset tilat NFE:stä, sammuttaa kaikki päällä olevat ledit ja siirtyy HMI_WAIT_LED_INIT-tilaan odottamaan REST-rajapinnan kuittausta siitä, että ledit todella ovat sammutetussa tilassa. Kun ledien ja muidenkin signaalien alkuarvot on kirjattu ajurin omiin tietorakenteisiin, ajuri siirtyy tilaan HMI_RUNNING, joka on ohjelman normaali operointitila.

5.11 Käytetyt ohjelmistot

HMI-ajurin kehityksessä on käytetty Eclipse-kehitysympäristöä tekstieditorina ja apuna lähdekoodihakemistojen hallinnassa. Kääntäjänä on käytetty GNU-projektin kehittämää GCC-ristikääntäjää, joka pystyy tuottamaan PowerPC-arkkitehtuurille tarkoitettuja ohjelmabinäärejä. Koontiautomaation työkaluiksi on valittu niin ikään GNU-projektin

kehittämä Autotools-kokoelma. Insinöörityössä esiintyvien UML-kaavioiden piirtämiseen on käytetty web-pohjaista draw.io-palvelua.

6 HMI-ajurin testaus ja profilointi

Kuten kaikki vähänkään laajemmat tietokoneohjelmat, myös HMI-ajuri altistuu kehityksensä aikana ohjelmointivirheille eli bugeille. Syntaksiin liittyvät virheet, jotka tuottavat kääntäjävirheitä ja -varoituksia, huomataan jo käännösvaiheessa ja voidaan kohtuullisen helposti korjata heti. On kuitenkin käytännössä väistämätöntä, että käännettyyn ohjelmaan jää joskus myös logiikkavirheitä, jotka ajon aikana ilmetessään voivat saada ohjelman kaatumaan tai toimimaan virheellisellä ja odottamattomalla tavalla. Logiikkavirheiden karsimiseksi HMI-ajuria on testattu kehityksen aikana paitsi manuaalisesti, myös automatisoiduin testein.

6.1 Manuaalinen testaus

Ohjelmiston manuaaliset testit suoritetaan aina, kun lähdekoodiin on tehty muutoksia. Niissä ohjelmistoa testataan kokonaisuutena, ja se käynnistetään komentoriviltä. Testaaja antaa järjestelmälle erilaisia syötteitä, kuten painelee HMI-paneelin painikkeita ja laukaisee WebGUI:n kautta simuloituja hälytyksiä. Jos HMI-ajuri ei reagoi annettuihin syötteisiin odotetulla tavalla tai kaatuu, mahdolliset logiikkavirheet voidaan löytää ja korjata. Testaaja voi myös antaa tarkoituksella virheellisiksi tiedettyjä syötteitä, jolloin voidaan arvioida sitä, miten ohjelmisto selviää virhetilanteista. Tarvittaessa apuna käytetään debuggeria, jolla ohjelman suoritus voidaan pysäyttää halutulle koodiriville ja esimerkiksi tarkastella muuttujien hetkellisiä arvoja. HMI-ajuri sisältää myös lokikirjoittajan, joka voidaan tarvittaessa määritellä tulostamaan ajurin tuottama lokisyöte näytölle tarkasteltavaksi.

6.2 Automaattiset yksikkötestit

Manuaalitestauksen lisäksi tarvitaan myös automaattitestejä, sillä jokaisen kuviteltavissa olevan rajatapauksen järjestelmällinen testaaminen manuaalisesti olisi erittäin raskasta ja aikaa vievää. Lisäksi ohjelmistosta halutaan testata erikseen myös pienempiä osia muusta tuotantokoodista erotettuna, jolloin virheiden paikantaminen on helpompaa. Tällöin puhutaan yksikkötestauksesta (*unit testing*). Yksikkötestauksessa testitapaukset kirjoitetaan erikseen jokaiselle testattavalle komponentille, kuten luokalle tai funktiolle. HMI-ajurin tapauksessa testattavat yksiköt ovat luokkia. Testejä on insinöörityön kirjoitushetkellä laadittu kaikille olemassa oleville paneelikäsittelijöille, lediajurille, hallintamoodin käsittelijälle ja tietokantaluokalle. Mahdollisuuksien mukaan testejä on tarkoitus kirjoittaa myös ainakin jakelijalle ja kytkinkäsittelijälle.

Yksikkötestien toteuttaminen on sitä helpompaa, mitä paremmin varsinainen tuotantokoodi soveltuu testattavaksi. Ideaalitilanteessa testattava koodi jakautuu selkeästi rajattuihin, itsenäisiin ja pieniin yksiköihin. Tällaisille yksiköille on helppo suunnitella testijoukko, joka kattaa mahdollisimman suuren osan mahdollisista rajatapauksista. Valtaosa HMI-ajurin luokista soveltuu yksikkötestattavaksi kohtuullisen hyvin, mutta testejä varten on silti jouduttu myös rakentamaan erillistä infrastruktuuria.

Suurin yksikkötestaukseen liittyvä haaste HMI-ajurin tapauksessa on, että monilla luokilla on riippuvuuksia muihin sovelluksen komponentteihin. Esimerkiksi paneelikäsittelijät kutsuvat muun muassa kytkinkäsittelijöiden, ledikäsittelijän ja tietokantaluokan funktioita. Jos testattava luokka kutsuu testien suorituksen aikana ulkopuolisia funktioita, eivät testit enää ole spesifejä testattavalle yksikölle. HMI-ajurin testeissä ongelma on ratkaistu korvaamalla testattavan olion ulkoiset riippuvuudet niin sanotuilla näköisversioilla (*test doubles*). Ne ovat luokkia, jotka periytyvät samasta abstraktista yläluokasta kuin niiden tuotantoversiot ja jakavat samat rajapinnat niiden kanssa, mutta funktioiden toteutukset ovat hyvin yksinkertaisia. Testattava luokka kutsuu abstraktin rajapinnan funktioita, joiden takana onkin näköisluokan toteutus, jota voidaan käyttää esimerkiksi testattavalta yksiköltä saatujen syötteiden tallentamiseen ja verifioimiseen yksikkötestin sisällä.

Testit on kirjoitettu Catch-nimisellä, C++-kieltä varten tarkoitetulla yksikkötestikehyksellä. Testit käännetään erilliseksi ohjelmabinääriksi, joka voidaan suorittaa kohdelaitteessa. Kun testibinääri ajetaan komentoriviltä, voidaan sille antaa erilaisia optioita, joilla valitaan esimerkiksi, ajetaanko kaikki testit vai ainoastaan tiettyjen yksiköiden testit. Kun kaikki valitut testit on suoritettu, tulostuu näytölle, mitkä niistä ovat onnistuneet ja mitkä epäonnistuneet.

6.3 Muistinkäytön analysointi

HMI-ajurin kehityksessä on pyritty noudattamaan ohjelmointityyliä, jossa tehdään mahdollisimman vähän dynaamista muistinvarausta. Tietyissä tilanteissa sitä ei kuitenkaan pysty välttämään. Kun dynaamisia varauksia tehdään, on mahdollisuuksien mukaan käytetty C++ -kielen standardikirjastoon kuuluvaa älykästä osoitintyyppiä *std::shared_ptr*, joka tuhoaa viitatun objektin ja vapauttaa sille varatun muistialueen automaattisesti, kun viimeinen objektiin viittaava älykäs osoitin tuhoutuu tai asetetaan osoittamaan johonkin toiseen objektiin. Eräissä tapauksissa on kuitenkin jouduttu luomaan objekteja heap-muistiin käyttämällä *new*-avainsanaa. Näissä tapauksissa objektien luominen on sijoitettu omistavan luokan konstruktoriin ja objektien tuhoaminen *delete*-avainsanalla sen destruktoriin. Näin voidaan varmistaa, että kaikki varatut resurssit myös vapautetaan. Tällaisesta ohjelmointityylistä käytetään olio-ohjelmoinnissa englanninkielistä termiä *RAII* (”*Resource Allocation Is Initialization”*).

Muistivuodot Netcon 200 -järjestelmän tyyppisessä sulautetussa järjestelmässä tekisivät siitä epäluotettavan ja epävakaan, joten edellä kuvattujen käytäntöjen lisäksi ohjelmabinääriä on analysoitu Valgrind-kokoelmaan kuuluvan Memcheck-työkalun avulla, jolla löydettiinkin muutamia ohjelman alustusvaiheessa tapahtuvia muistivuotoja. Kyseiset muistivuodot johtuivat ohjelmassa käytetyn Libxml2-kirjaston väärästä käyttötavasta, ja ne pystyttiin korjaamaan melko helposti.

6.4 Profilointi

Sekä HMI-ajurin että koko järjestelmän resurssien käyttöä on testauksen aikana analysoitu *top*-ohjelmalla, joka näyttää järjestelmän hetkellisen CPU:n ja keskusmuistin käyttöasteet sekä yksittäisten käynnissä olevien prosessien osuudet niistä.

7 Työn tulokset

Insinöörityön viimeistelyn aikaan HMI-ajurin kehitys on jatkunut hieman yli vuoden verran, ja valtaosa tuotteen pilotointivaiheessa tarvittavasta toiminnallisuudesta on toteutettu. Ajurin toiminta on vahvasti riippuvaista järjestelmän muista osista kuten laitteistosta, muusta ohjelmistosta sekä firmware- eli laiteohjelmistokerroksesta, joita kaikkia kehitetään rinnakkain HMI-ajurin kanssa. Tähän asti valmiiksi kasattuja Netcon 200 -prototyyppiyksiköitä on myös riittänyt tuotekehitystiimin kesken jaettavaksi vain rajallinen määrä. Näistä syistä ohjelmiston kaikkia aspekteja ei ole päästy vielä testaamaan täysimääräisesti, vaan järjestelmän muita osia on pitänyt osittain simuloida ohjelmallisesti.

Tästä huolimatta ajuria on jo ehditty testaamaan useilla erilaisilla konfiguraatioilla, ja sen vakaus ja suorituskyky ovat testeissä olleet pääosin hyviä. Vasteajat eri tapahtumiin reagoimisessa ovat osoittautuneet hyväksyttävän lyhyiksi. Esimerkiksi kytkimiä ohjatessa aikaviive napinpainalluksesta sähköisen ohjaussignaalin aktivoitumiseen on sekunnin murto-osien luokkaa eli paneelia käyttävän ihmisen näkökulmasta havaittavissa, mutta ei häiritse käyttöä. Normaalitilanteessa, jossa HMI-ajurille ei anneta mitään syötteitä, sen CPU-käyttöaste on nolla prosenttia, koska ajuri toimii tapahtumapohjaisesti. Ajurin käynnistymisvaiheessa kuormitus nousee hetkellisesti noin 10 prosenttiin eikä sen ole koskaan huomattu ylittävän 15 prosentin lukemaa. Virtuaalista muistia HMI-ajuri ja kaikki sen käyttämät kirjastot varaavat yhteensä noin 19 megatavun verran. Käytännössä prosessi ja sen käyttämät dynaamisesti linkitetyt kirjastot sitovat hieman yli 5 megatavua fyysistä muistia, mikä järjestelmän 250 megatavun kokonaismuistin huomioiden tarkoittaa noin 2.3 %:n muistinkäyttöä. Suurin osa käytettävästä muistista varataan ajurin käynnistymisen yhteydessä, ja muistinkäyttö pysyy sen jälkeen tasaisena.

Analyysi on osoittanut, että ajurin käynnistymisvaihe on järjestelmää eniten kuormittava vaihe. Yksi merkittävä syy tähän vaikuttaa olevan päätason tilakoneen HMI_CONV_SIGNALS-tilassa lähetettävä mltconv-pyyntö. Yksinkertaisimmallakin laitekokoonpanolla ajurin tietokantaan kertyy vähintään muutamia kymmeniä signaaleja ja komentoja, jotka kaikki kootaan yhdeksi merkkijonoparametriksi ja lähetetään NFE:n REST-palvelimelle kerralla. Tämä ilmenee hetkellisenä piikkinä NFE:n CPU-käyttöasteessa, joka nousee usein jopa lähes sataan prosenttiin asti noin sekunnin ajaksi.

HMI-ajurin arkkitehtuuri mukautuu ainakin kirjoitushetkellä arvioitaessa kohtuullisen hyvin ohjelmistoon tehtäviin laajennuksiin ja parannuksiin. Jaetut abstraktit rajapinnat mahdollistavat esimerkiksi käsittelijätyyppisten luokkien käsittelyn polymorfisesti, mikä suoraviivaistaa sovelluksen toimintaa. Kapselointi on suurimmaksi osaksi onnistunut hyvin, ja ohjelmiston eri osia voidaan kehittää itsenäisinä kokonaisuuksina ilman merkittäviä muutoksia muihin komponentteihin. Poikkeuksiakin kuitenkin löytyy. Esimerkiksi tietokantaluokka on ajurin kehityksen aikana päässyt kasvamaan melko suureksi kokonaisuudeksi ja siihen on lisätty paljon toiminnallisuutta, joka ei varsinaisesti enää kuuluisi sille. Tämä heikentää osaltaan sovelluksen modulaarisuutta ja varsinkin vaikeuttaa tietokantaluokan yksikkötestaamista.

Ehkä merkittävin tekijä HMI-ajurin laajennettavuutta arvioidessa on, miten helposti siihen pystytään lisäämään tuki uudelle paneelityypille. Tämän arviointi jää valitettavasti pitkälti tämän insinöörityön ulkopuolelle, sillä työn valmistumishetkellä uusien laajennuspaneelien tuotekehitys ei ole vielä alkanut. Voidaan kuitenkin todeta, että esimerkiksi virta- ja jännitemittauksia sisältävien laajennusyksiköiden paneeleille, johon joukkoon myös VCCC-32 kuuluu, on sovelluksessa tuki PanelHandlerVCBase-yläluokan muodossa. Luokka on suunniteltu siten, että siitä pystytään periyttämään paneeli, joka sisältää minkä tahansa yhdistelmän 1–4 virta- tai jännitemittausmoduulista. Myös uuden GWDD-tyyppisen pääpaneelin luominen onnistunee PanelHandlerGWDDBase-yläluokan ansiosta kohtuullisen helposti.

Kirjoitushetkellä HMI-ajurin lähdekoodi sisältää 5771 koodiriviä, kun sekä lähdekooditiedostot että header-tiedostot huomioidaan laskuissa. Luvussa ei ole huomioitu tyhjiä rivejä eikä kommenttirivejä. Tähän mennessä toteutettujen paneelikäsittelijäluokkien

perusteella voidaan arvioida, että uuden paneelityypin tukemiseen tarvittavan paneelikäsittelijän lisääminen onnistuu noin 300-500 koodirivillä.

8 Johtopäätökset

HMI-ajurin kehityksen ja testauksen aikana saatujen kokemusten perusteella voidaan sanoa, että ohjelmisto toimii ja täyttää sille asetetut vaatimukset. Tapahtumapohjaisella arkkitehtuurilla on saavutettu järjestelmä, joka käyttää suoritinresursseja ainoastaan, kun jokin HMI:hin liittyvä heräte vaatii käsittelyä. Myös muistinkäyttö pysyy kohtuullisena prosessia suoritettaessa. Yksi HMI-ajurin arkkitehtuurin merkittävistä eduista on, että ohjelmassa ei ole tarvinnut käyttää säikeitä, jotka tuovat mukanaan tiettyjä haasteita ja tekevät sovelluksesta monimutkaisemman.

Ajurissa on kuitenkin myös kehityskohteita. Tietyt luokat (kuten tietokantaluokka) ovat päässeet kehityksen myötä kasvamaan liian suuriksi kokonaisuuksiksi. Näitä luokkia kannattanee tulevaisuudessa jakaa useammaksi pienemmäksi luokaksi, joista jokainen toimii omalla selkeästi rajatulla vastuualueellaan. Tämä helpottaisi ohjelmiston jatkokehitystä ja yksikkötestien suunnittelua sekä selkeyttäisi lähdekoodia.

Sovelluksen resurssienkäyttö pysyy yleensä hyvin kohtuullisena, mutta varsinkin ajurin käynnistyessään NFE:ssä aiheuttamaa CPU-käyttöpiikkiä ja sen vaikutusta järjestelmän kokonaisuuteen on syytä seurata. Ongelma korostuu, jos HMI-ajurin prosessi luovuttaa toistuvasti alustusvaiheessa mltconv-kutsun suorittamisen jälkeen esimerkiksi puutteellisen NFE-tietokannan takia. Tällöin järjestelmän prosessimonitoriohjelma *pokemon* käynnistää ajurin välittömästi uudelleen, jolloin CPU-piikit toistuvat muutaman sekunnin sykleissä. Mikäli tämä osoittautuu merkittäväksi ongelmaksi järjestelmän vakauden kannalta, on etsittävä vaihtoehtoinen, vähemmän NFE:tä kuormittava tapa hakea signaalikäännökset palvelimelta.

Testiautomaatiossa on myös kehitettävää. Kirjoitushetkellä yksikkötestien kokoelma käännetään ristikääntäjällä PowerPC-arkkitehtuurille sopivaksi binääritiedostoksi, joka

on siirrettävä Netcon 200 -kohdelaitteelle ja ajettava sen komentokehoitteessa. Testien kääntäminen, siirto ja ajaminen vievät usein kaikkiaan yli minuutin aikaa ja vaativat manuaalista työtä. Siksi ne jäävät usein tekemättä nopeassa kehitys-testaussyklissä, jossa tuotantokoodin uudelleenkääntämistä tehdään tiheässä tahdissa. Testien koontiautomaatiota (*build automation*) voisi kehittää siten, että testibinääri käännettäisiinkin x86-arkkitehtuurille sopivaksi, jolloin se voitaisiin ajaa kääntäjäpalvelimen ympäristössä aina, kun tuotantokoodista käännetään uusi versio. Tällöin käännösvaiheessa nähtäisiin mahdollisten kääntäjän varoitusten lisäksi myös yksikkötestien testiraportti, eikä testibinääriä tarvitsisi siirtää kohdelaitteelle. Myös testien integroimista yrityksessä käytössä olevaan Jenkins-automaatiopalvelimeen kannattaa tutkia.

Lähteet

1. Elovaara, Jarmo & Haarla, Liisa. 2011. Sähköverkot I: Järjestelmätekniikka ja sähköverkon laskenta. Helsinki. Gaudeamus Helsinki University Press/Otatieto.
2. Sähkömarkkinalaki. 2013. Verkkoaineisto. Finlex. <https://finlex.fi/fi/laki/alkup/2013/20130588#Lidp446540224>. Luettu 4.9.2020.
3. Jakelujohdot ja muuntamot. 2020. Verkkoaineisto. Säteilyturvakeskus. <https://www.stuk.fi/aiheet/sahkonsiirto-ja-voimajohdot/jakelujohdot-ja-muuntamot >. Luettu 10.9.2020.
4. Sähköverkko tutuksi. 2017. Verkkoaineisto. Elenia. <https://www.elenia.fi/yritys/s%C3%A4hk%C3%B6verkko-tutuksi>. Luettu 10.9.2020.
5. Elovaara, Jarmo & Haarla, Liisa. 2011. Sähköverkot II: Verkon suunnittelu, järjestelmät ja laitteet. Helsinki. Gaudeamus Helsinki University Press/Otatieto.
6. IEC 60870-5-101 Ed. 2.0: Telecontrol equipment and systems – Part 5-101: Transmission protocols – Companion standard for basic telecontrol tasks. 2001. International Electrotechnical Commission.
7. Gourley, David; Totty, Brian; Sayer, Marjorie; Reddy, Sailu & Aggarwal, Anshu. 2002. HTTP – The Definitive Guide. O'Reilly Media, Inc.
8. Fielding, Thomas. 2000. Architectural Styles and the Design of Network-based Software Architectures. Väitöskirja. Verkkoaineisto. <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Luettu 13.11.2020.
9. The Java Language Environment. Verkkoaineisto. Oracle. <https://www.oracle.com/java/technologies/simple-familiar.html>. Luettu 19.9.2020.
10. Introduction to Object-Oriented Programming Concepts (OOP) and More. Verkkoaineisto. CodeProject. <https://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep>. Luettu 18.9.2020.
11. Scott, Michael Lee. 2009. Programming Language Pragmatics, 3rd edition. Burlington, MA. Morgan Kaufmann Publishers.
12. Meyer, Bertrand. 1997. Object-Oriented Software Construction, 2nd edition. Prentice Hall.

13. Schmuki, Ruedi; Wagner, Ferdinand; Wagner, Thomas & Wolstenholme, Peter. 2006. Modeling Software with Finite State Machines, A Practical Approach. Boca Raton, FL. Auerbach Publications.
14. Hopcroft, John; Motwani, Rajeev & Ullman, Jeffrey. 2001. Introduction to automata theory, languages and computation, 2nd edition. Addison Wesley.
15. OMG Unified Modeling Language Version 2.5.1. 2017. Object Management Group. Verkkoaineisto. <https://www.omg.org/spec/UML/2.5.1/PDF>. Luettu 12.11.2020.
16. NFE Monitor User Manual. 2004. Netcontrol Oy. Yrityksen sisäinen dokumentti.