OULUN AMMATTIKORKEAKOULU

Jarkko Säkkinen

KOMPONENTTIEN VIRRANMITTAUKSEN AUTOMATISOINNIN MAHDOLLISTAMINEN

KOMPONENTTIEN VIRRANMITTAUKSEN AUTOMATISOINNIN MAHDOLLISTAMINEN

Jarkko Säkkinen
Opinnäytetyö
Kevät 2019
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä: Jarkko Säkkinen
Opinnäytetyön nimi: Komponenttien virranmittauksen automatisoinnin mahdollistaminen
Työn ohjaaja: Timo Vainio
Työn valmistumislukukausi ja -vuosi: Kevät 2019 Sivumäärä: 42

Opinnäytetyö tehtiin Polar Electro Oy:lle. Työn tavoitteena oli vähentää laitteiden virranmittauksiin kuluvaa aikaa. Mittauksiin kuluvaa aikaa voitaisiin vähentää automatisoimalla virranmittaukset. Ongelmaa on moneen kertaan lähdetty Polarilla ratkaisemaan, ja aiempien projektien materiaalia ja laitteistoa käytettiin tässäkin projektissa.

Kokonaisvirran mittauksen automatisointi oli jo toiminnassa, mutta usein halutaan myös tietää mikä komponentti kuluttaa virtaa ja kuinka paljon. Komponentti virranmittauksen automatisointia lähdettiin ensin ratkaisemaan määrittelemällä, mikä on työn toivottu lopputulos. Eli miten ja mihin järjestelmää tulisi pystyä käyttämään. Sitten lähdettiin selvittämään, mitä mittalaitteita on saatavilla aiemmista projekteista ja miten kyseessä olevilla mittalaitteilla voidaan mitata virtaa, ja mitoittamaan vastuksia virranmittaukseen.

Seuraavassa vaiheessa lähdettiin tekemään ohjelmaa, jota voidaan testikoodista ohjata ja käskyttää mittaamaan virtaa. Projektiin oli myös suunniteltu kolmas vaihe, mittalaitteiden kalibrointi, mutta aiempi vaihe kasvoi paljon laajemmaksi kuin aluksi luultiin, joten kalibrointivaihe jätettiin tehtäväksi myöhempään projektiin.

Tuloksena on C#-kielellä kirjoitettu ohjelma, jolla voidaan mitata virtaa National Instrumentsin datankeräyslaitteita käyttäen. Ohjelma on Windows-palvelu, jota voidaan ohjata web-rajapintaa käyttäen.

Avainsanat: C#, Topshelf, National Instruments, Windows-palvelu, Web API, virranmittaus

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

Author: Jarkko Säkkinen
Title of thesis: Multichannel current consumption system to enable automation
Supervisor: Timo Vainio
Term and year when the thesis was submitted: Summer 2019 Number of pages: 42

The thesis was done for Polar Electro Oy. The goal of this project was to reduce the amount of time spent doing manual current measurements, which takes a lot of time and money. The solution was to try and automate current measurements.

First phase of the project was to search material and devices left over from similar projects done at Polar, and then to research how current is measured with these devices. This phase also includes defining the system that will be created for measuring current and how this system can be ordered to measure.

The next phase was to create a system that can be used to measure current using National Instruments data acquisition devices. The system also had to have a web API, so the system could be used from test code, which can be running on a separate machine.

There was also a third phase planned, which was calibrating the measurement devices to be used. Unfortunately, the second phase ended up being larger than expected so this calibration phase was left out and to be done in a separate project.

The result of this project is a program written in C# that can be used to measure current from multiple components simultaneously using one or more National instruments devices. The program is a Windows service that can be controlled using a web API.

Keywords: C#, Topshelf, National Insturments, Windows service, Web API, Current measurement

TERMIT JA LYHENTEET

API	Application programming interface eli rajapinta, jonka avulla esim. ohjelmaa voidaan ohjata toisesta ohjelmasta.
DAQ	Data acquisition eli datan keräys, mutta usein lyhennettä käytetään, kun puhutaan DAQ-laitteista eli datankeräyslaitteista.
Framework	Ohjelmistokehys, jonka päälle ohjelmia voidaan tehdä
Library	Kirjasto eli kokoelma luokkia, joita voidaan käyttää ohjelman tekoon
NI	National Instruments, mittalaitteita valmistava yritys

SISÄLLYS

1 JOHDANTO

Virranmittaus on tärkeää laitekehityksessä, koska usein halutaan tietää esimerkiksi, kuinka kauan laitetta voidaan käyttää yhtäjaksoisesti. Kun ominaisuuksia lisätään, halutaan tietää, kuinka nämä uudet ominaisuudet vaikuttavat akun kestoon. On myös tärkeää tietää, kun huomataan kokonaisvirrankulutuksen kasvaneen, minkä komponentin virrankulutus on kasvanut. Näin voidaan etsiä virheitä ja yleisesti parantaa akun kestoa. Virranmittausta tehdään muun muassa yllämainituista syistä koko ajan, mutta se vie paljon aikaa ja sitoo ainakin yhden henkilön siihen työhön, joten se maksaa myös rahaa. Tätä ajan käyttöä voidaan vähentää automatisoimalla virranmittaus. Automatisoimalla voidaan myös mahdollistaa virrankulutusdatan käyttöä useampaan tarkoitukseen.

Opinnäytetyö tehtiin Polar Electro Oy:lle, joka valmistaa mm. aktiivisuusrannekkeita ja sykemittareita. Polarilla kokonaisvirran mittaus on automatisoitu, mutta komponenttien virranmittausta tehdään vielä käsin. Tämän opinnäytetyön tavoitteena oli mahdollistaa komponenttien virranmittauksen automatisointi, tekemällä palvelu tai jokin muu ohjelma, jota voidaan ohjata ohjelmallisesti tekemään mittauksia testiautomaatiojärjestelmästä. Palvelun ohjelmallinen ohjaus mahdollistaa palvelun käytön mm. testiautomaatiojärjestelmästä, jossa kerättyä virranmittausdataa voidaan käyttää moneen tarkoitukseen.

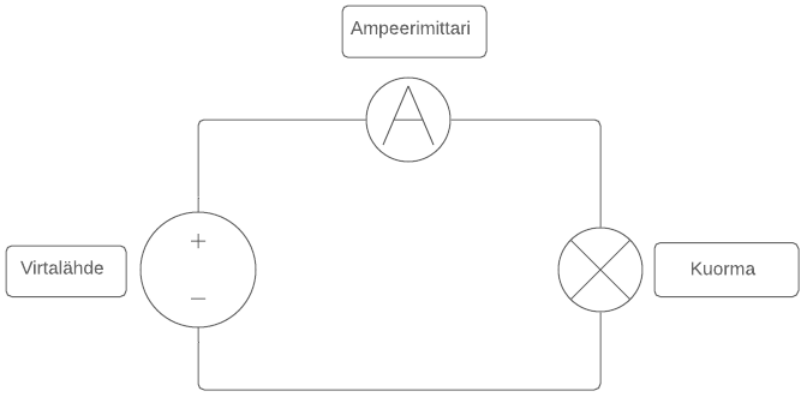
Teoriaosiossa käsitellään mm. virranmittausten tekoa DAQ-laitteilla, vastuksien mitoitusta ja rajapintojen tekoa. Tässä osiossa käsitellään myös virranmittausjärjestelmässä käytettäviä tekniikoita ja kirjastoja. Toteutusosiossa käydään läpi, miten järjestelmää voidaan käyttää, miten asiat on tehty ja mitä ongelmia ja niihin tehtyjä ratkaisuja työn aikana esiintyi.

2 VIRRANMITTAUS JA VASTUSTEN MITOITUS

Sähkövirta voidaan määritellä seuraavalla tavalla:

> *"Sähkövirralla (engl. current) tarkoitetaan sähkövirran (atomin elektronien tai aukkojen) siirtymistä johtavassa materiaalissa atomista toiseen. Sähkövirta syntyy, kun syöttöjännite vaikuttaa suljettuun virtapiiriin. Erityisen hyvin virta kulkee kuparissa ja muissa johdemateriaaleissa. Virran tunnus on I ja perusyksikkö on ampeeri (A)." (1, s. 13.)*

Sähkövirtaa voidaan mitata ampeerimittareille, joka kytketään sarjaan kytkennän kanssa (kuva 1).

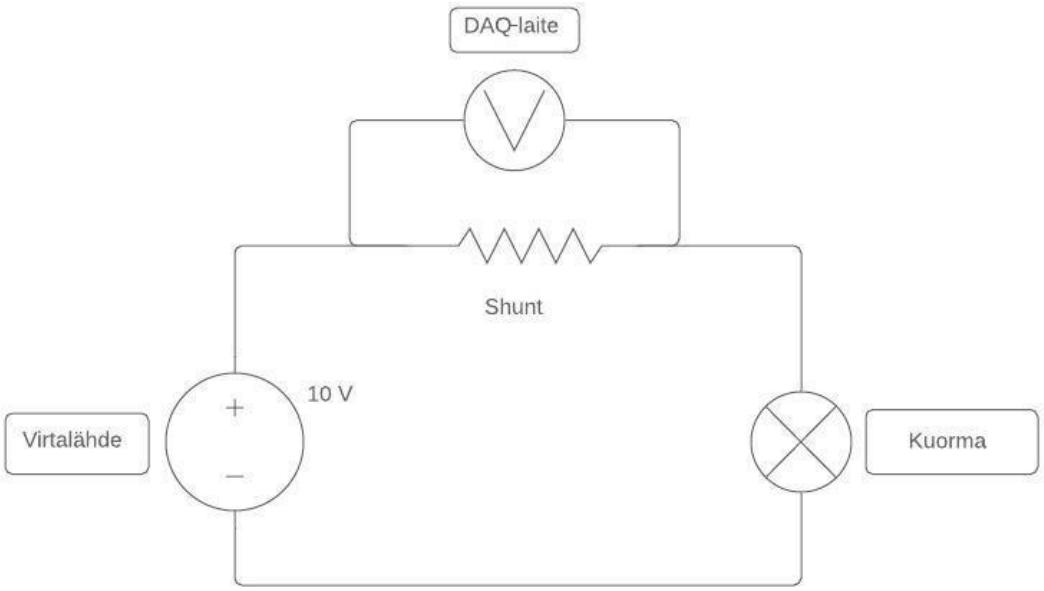


KUVA 1. Ampeerimittarin sarjaankytkentä

2.1 Virranmittaus DAQ-laitteilla

Data acquisition (lyh. DAQ) eli datankeräyslaitteilla virranmittaus tapahtuu usein mittaamalla vastuksessa tapahtuvaa jännitehäviötä. Kun tiedetään tarkkaan, mikä on vastuksen resistanssi ja mitkä ovat minimi- ja maksimivirrat piirissä, voidaan tämä jännitehäviö muuttaa ampeereiksi eli sähkövirran suureeksi.

Kun vastuksen avulla mitataan virtaa, on huomioitava vastuksen resistanssin vaikutus olemassa olevaan kytkentään. Jos vastus on liian suuri, se voi häiritä laitteen toimintaa. On myös huomioitava vastuksen koon vaikutus mittauksen resoluutioon. Jos vastus on liian pieni, on jännitehäviö myös pienempi, eli mittauksen resoluutio on huonompi. Kun virtaa mitataan vastuksen avulla, asetetaan vastus sarjaan kytkennän kanssa ja mittalaite rinnan vastuksen kanssa. (2.) (Kuva 2.)



KUVA 2. DAQ-laitteen (tai muun jännitemittarin) kytkentä vastuksella virtaa mitattaessa

2.1.1 Jännitehäviö virraksi

Datankeräyslaitteet hoitavat jännitehäviön (engl. voltage drop) muutoksen virraksi, mutta on silti hyödyllistä tietää, miten se käytännössä tapahtuu. Tässä luvussa käydään läpi, miten jännitehäviöstä voidaan laskea virta. (3.)

Jos mitataan esimerkiksi 0–0,4 A:n aluetta 0,5 Ω:n vastuksella, voidaan jännitehäviö maksimivirralla vastuksessa laskea kaavalla: $0{,}5\ \Omega * 0{,}4\ \mathrm{A} = 0{,}2\ \mathrm{V}$. Eli jännitehäviöt ovat 0–200 mV:n alueella. Esimerkiksi jos mitataan 50 mV:n jännitehäviö, se voidaan jakaa vastuksen arvolla ja saada virta: $\frac{0{,}05\ \mathrm{V}}{0{,}5\ \Omega} = 0{,}1\ \mathrm{A}$.

2.1.2 Mittauksen resoluutio

Tässä luvussa tarkastellaan vastuksen koon vaikutusta jännitehäviön ja mittauksen resoluutioon. (4.)

Ensimmäisenä lasketaan mittalaitteen resoluutio eli analogi-digitaalimuuntimen (engl. analog-to-digital converter. Lyh. ADC) pienin huomattava muutos arvossa. Esimerkiksi NI 9238 -mittalaitteen

ADC-resoluutioksi on ilmoitettu 24 bittiä ja mittausalueeksi ±500 mV. Tästä voidaan laskea pienin huomattava muutos arvossa: $\frac{\pm 500}{2^{24}} = \frac{1}{2^{24}} = 59{,}6nV = \sim 60nV$.

Mitattavan alueen halutaan myös olevan mahdollisimman lähellä mittalaitteen mittausaluetta parhaan resoluution saamiseksi. Esimerkiksi jos mitattava alue on ±100 mV ja mittalaitteen mittaus alue on ±500 mV, on vain 1/5 mittausalueesta käytössä. Mittapisteiden määrä on siten paljon pienempi. Jos käytetään liian pientä vastusta, on mitattava alue pienempi ja mitattavan alueen resoluutio huonompi. Jotkut mittalaitteet tukevat mittausalueen vaihtoa eli resoluution muutosta. Esimerkiksi jos mittalaite tukee ±1 V:n, ±5 V:n ja ±10 V:n alueita, voidaan käyttää eri mittausalueita parantamaan resoluutiota mittauksille.

2.2 Vastusten mitoitus

Kun mitoitetaan vastusta virranmittauksiin, on yritettävä minimoida kaikki mukaan laskematon resistanssi, esimerkiksi mahdollisista johdoista koituva resistanssi ja vastuksen tarkkuuden eli toleranssin tulisi olla hyvä. Suurempia virtoja mitattaessa tärkeitä arvoja ovat myös lämmön ja tehon kesto, koska vastuksen arvo muuttuu lämpötilan muuttuessa. Kun lämmön kesto ylitetään, vahingoittuu vastuksen resistanssi pysyvästi, jolloin kaikki tulevat mittaustulokset ovat väärässä. Tässä projektissa mitattavat virrat olivat sen verran pieniä, että nämä arvot eivät tulleet vastaan. (5.)

3 NI, DAQ-LAITTEET JA NIIDEN KÄYTTÖ OHJELMALLISESTI

National Instruments (lyh. NI) valmistaa paljon erilaisia mittalaitteita. Tässä luvussa käsitellään muutamia tärkeitä asioita mittalaitteista ja niiden käytöstä ohjelmallisesti.

DAQ on lyhenne sanoista data acqusition eli datan keräys. Mm. NI valmistaa laitteita tähän tarkoitukseen ja niitä kutsutaan DAQ-laitteiksi tai yleisesti vain DAQ:eiksi. (Kuva 3 ja 4.)

> *”DAQ-laite toimii liitäntänä tietokoneen ja ulkomaailman ilmiöitä mittaavien antureiden välillä. Päätehtävänä on muuntaa sisään tulevat analogiset signaalit digitaaliseen muotoon tietokoneen sovellusohjelmia varten.” (6.)*

KUVA 3. NI CDAQ-kotelot (engl. chassis) joihin voidaan liittää DAQ-moduuleja, vasemmalla kuvassa on cDAQ-9179 ja oikealla cDAQ-9191 (7.)

KUVA 4. NI 9238 DAQ-moduuli (engl. module) (8.)

3.1 NI-DAQmx-ajuri

Jotta NI:n DAQ-laitteita voitaisiin käyttää ja hallita tietokoneella, tarvitaan ajurit (eng. drivers) laitteille. Ajurin nimi on NI-DAQmx ja se on saatavissa NI:n sivuilta Windows- ja Linux-käyttöjärjestelmille. (9.)

Ajurin lisäksi kehitykseen tarvitaan kirjasto tai rajapinta, jolla voidaan käyttää ja ohjata ajuria eli ohjata laitteita ajurin kautta tehdyssä ohjelmassa eli ohjelmallisesti. NI tarjoaa näitä kirjastoja muutamille eri ohjelmointikielille. (10.)

3.2 Ohjelmointikielen valinta

Kirjastolla tarkoitetaan kielitukea tai rajapintaa, jonka avulla NI-DAQmx-ajuria voidaan ohjata ohjelmallisesti eli jostain ohjelmasta. Näitä kirjastoja on saatavilla muutamille eri ohjelmointikielille käyttöjärjestelmästä riippuen. Kaikki luetellut kirjastot ovat NI:n tekemiä.

- Windowsille saatavilla olevia kirjastoja:
    - National Instruments LabVIEW
    - ANSI C

    - Microsoft Visual C++
    - Microsoft Visual C# .NET
    - Microsoft Visual Basic .Net
    - Microsot Visual Basic 6.0
    - Python (11.)
- Linuxille saatavilla olevia kirjastoja:
    - National Instruments LabVIEW
    - ANSI C. (10.)

Monet edellä mainituista kirjastoista ovat wrapper-rajapintoja. Wrapper-rajapinta on rajapinta C-rajapintaan, joka ohjaa ajuria. NI on kertonut mahdollisesti lopettavansa tuen näihin rajapintoihin ja tämän ohjelmaa ei lähdetty tekemään näillä kielillä. (12.)

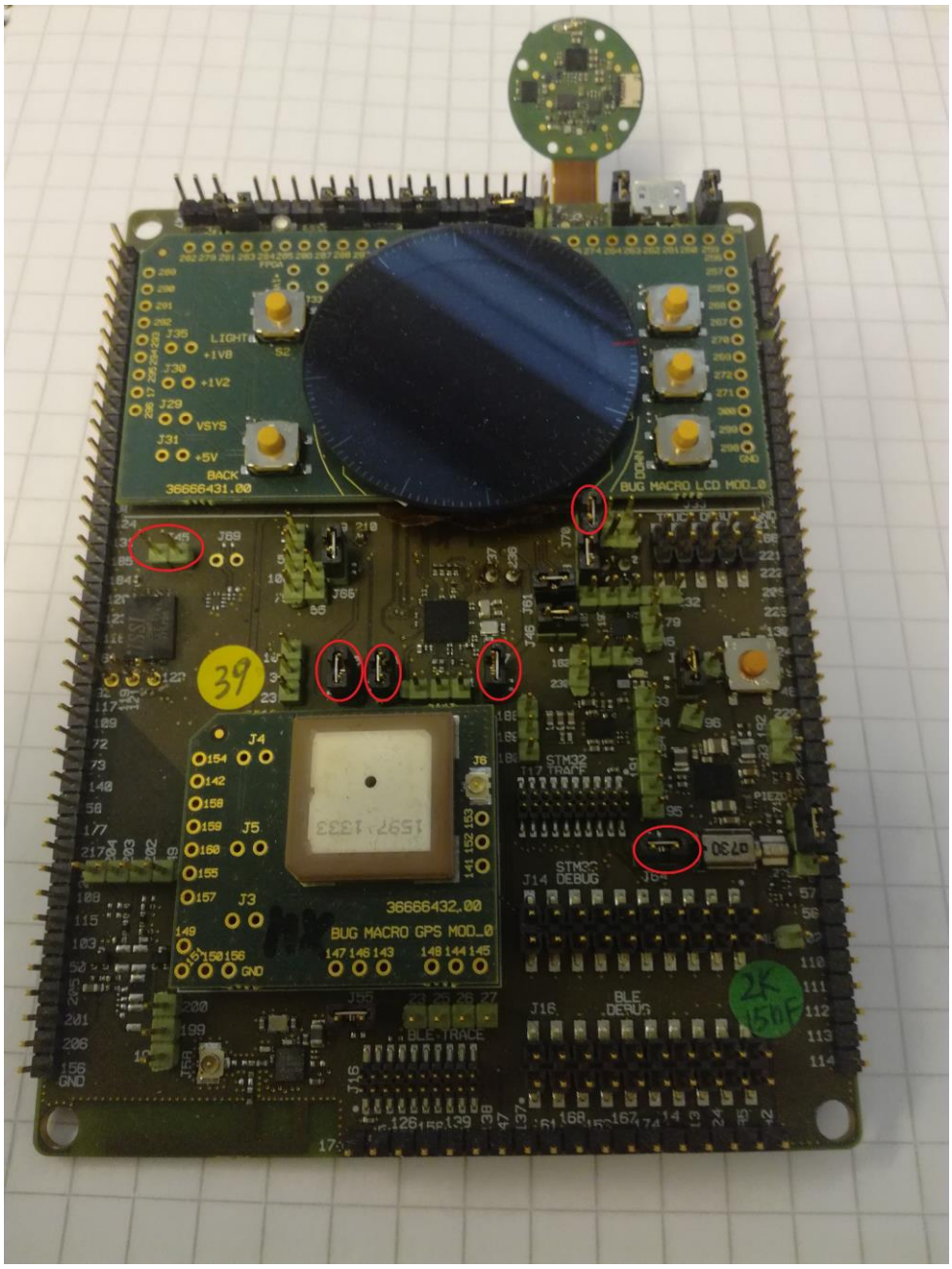
Löytyi myös yksi Java-rajapinta NI-DAQmx-ajurille, mutta se ei ollut NI:n tekemä ja se vaikutti muutenkin keskeneräiseltä. Monet tutkitut rajapinnat olivat todella huonosti dokumentoidut, esim. C# .NET-rajapinnan dokumentoinnista puuttui sanoja lauseista ym.

Lopulta valittiin C# .NET -rajapinta, koska se vaikutti parhaalta vaihtoehdolta, joka täytti kaikki ohjelman muut tarpeet. Lisäksi Linuxille oli heikosti saatavilla kirjastoja, eli oli pakko tehdä Windows-ohjelma muutenkin. Windows-ohjelma voidaan tehdä Windows-palveluksi, joka voidaan esim. asettaa palvelu käynnistymään itsestään. Siten ei tarvitse huolehtia ohjelman käynnistämistä erikseen.

LabVIEW olisi varmasti antanut parhaimmat mahdollisuudet mittalaitteiden käyttöön ja varmaan myös parhaan dokumentaation, mutta koska ohjelmassa tulee olla myös web-rajapinta, jonka kautta mittalaitteita voidaan ohjata, ei LabVIEW välttämättä olisi mahdollistanut tätä.

3.3 Mitattava laite

Makro (engl. macro board) on testilaite, jota käytetään rannelaitteiden kehityksessä. Siinä on samoja komponentteja kuin itse rannelaitteessa. Makrossa on kiinni monia mittauspisteitä, joita halutaan mitata. Tämän takia tarvitaan myös mittalaite, jolla pystytään mittaamaan montaa kanavaa samanaikaisesti. (Kuva 5.)



KUVA 5. Mitattava laite, punaisella ympyröitynä muutama mittapiste, joita halutaan mitata

4 VIRRANMITTAUSJÄRJESTELMÄ

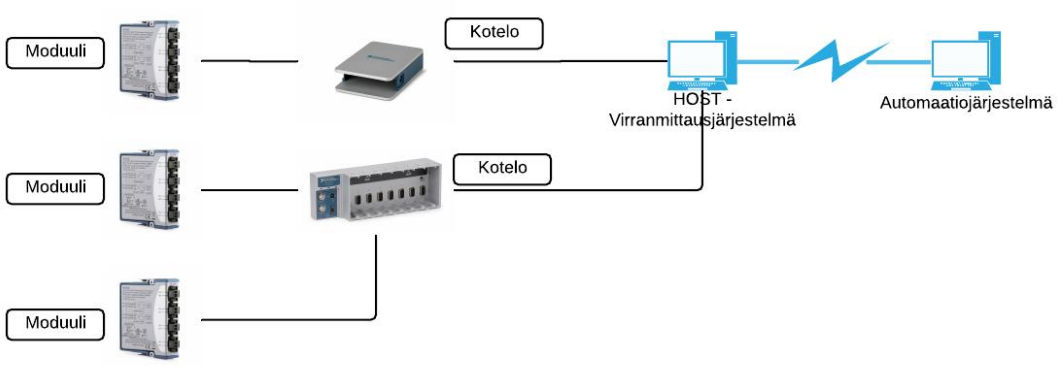
Tässä luvussa esitellään virranmittausjärjestelmä, jota lähdetään tekemään ja käydään läpi määrityksiä, jotka on hyvä huomioida ennen työn alkua. Esimerkiksi tuli selvittää miten ohjelmaa käytetään, koska ei haluttu tehdä ohjelmaa, joka pyörii paikallisesti jollain koneella ja joka täytyy käydä manuaalisesti käynnistämässä. Tämä tarkoittaa, että ohjelmassa täytyy olla jokin rajapinta, jonka avulla mittauksia voidaan käynnistää ja hallita ohjelmallisesti. Tämä rajapinta on määrittävä tekijä mittausten automatisoinnissa.

Tuli myös käydä hieman läpi, minkälaisia käskyjä ohjelmalle lähetetään, koska ei haluta, että käyttäjän tarvitsee tietää, mikä mittalaite on missäkin mittapisteessä kiinni. Tähän tarvitsee luoda konfiguraatiot, joissa listataan, mikä mittapiste on missä mittalaiteessa ja mittalaitteen kanavassa kiinni.

4.1 Automatisoitu virranmittausjärjestelmä

Virranmittauksia ei käytännössä tehdä automaattisesti tässä työssä, vaan tämä työ mahdollistaa virranmittausten teon testiautomaatiojärjestelmästä, jossa tehdään muita automaattisia tehtäviä. Tämä testiautomaatiojärjestelmä on virranmittausjärjestelmän käyttäjä (engl. client), eli kaiken palautetun datan ym. ei tarvitse olla ihmisen luettavassa formaatissa. Testiautomaatiojärjestelmä käsittelee palautetun datan haluamallaan tavalla. Testiautomaatiojärjestelmään voidaan luoda testitapauksia tai tehtäviä, joita voidaan suorittaa esimerkiksi joka yö tai kun tehdään muutoksia.

Testiautomaatiojärjestelmä on toiminnassa samalla koneella tai jollain toisella kuin virranmittausjärjestelmä. Testiautomaatiojärjestelmä lähettää käskyjä virranmittausjärjestelmän-rajapintaan, joka hallitsee DAQmx-ajuria. Tämä ajuri ohjaa kaikkia mittalaitteita, jotka on yhdistetty koneeseen. Mittalaitteita voidaan yhdistää koneeseen monella eri tavalla, esimerkiksi USB:n kautta tai Wi-Fi:n kautta, riippuen käytetystä NI DAQ -kotelosta (kuva 3). Virranmittausjärjestelmässä voi olla monta NI DAQ -koteloa. Yhdessä kotelossa voi olla monta moduulia tai vain yksi. Moduuleissa voi myös olla eroja, esimerkiksi mittauskanavien määrässä. On myös mittalaitteita, joissa ei ole moduulia ollenkaan. (Kuva 6.)



KUVA 6. Suunniteltu virranmittausjärjestelmä

4.2 Windows-palvelu

Virranmittausohjelmasta tehdään Windows-palvelu (engl. service), koska ollaan tekemässä ohjelmaa vain Windows-käyttöjärjestelmälle. Windows-palvelu on ohjelma, joka voidaan asentaa Windowsille ja se toimii taustalla eli tausta prosessissa. Palvelu voidaan asettaa käynnistymään, kun Windows käynnistyy tai jonkun muun tapahtuman yhteydessä. Palvelun hallintaan voidaan käyttää Windowsin Service Control Manageria. (13.)

4.3 Mittalaitteiden ja mitattavien laitteiden konfiguraatiot

Konfiguraatiota tuli hieman käydä läpi ennen ohjelman tekoa, koska mittausohjelman käyttäjän ei haluta tarvitsevan tietää mitään käytettävistä mittalaitteista tai siitä, missä mittalaitteessa mikäkin mittapiste on kiinni. Näin mittausohjelmalle voidaan sanoa esimerkiksi, ”mittaa GPS-komponentin virtaa 10 minuuttia”. Ohjelma tarvitsee tiedon siitä, missä mittalaitteessa on mikäkin mitattava laite ja mittapiste. Näissä konfiguraatioissa kerrotaan ohjelmalle myös mittapisteen vastuksen arvo ja maksimi- ja minimivirrat.

5 RAJAPINTA

Ohjelmaan tuli tehdä jonkinlainen rajapinta (engl. API, Application programming interface), jonka avulla mittauksia, dataa ym. voidaan hallita toisesta ohjelmasta eli testiautomaatiojärjestelmästä. Web-rajapinta on ehkä paras vaihtoehto, koska se mahdollistaa myös palvelun käytön muista koneista ja ohjelmista helposti. Tässä luvussa käydään läpi, miten web-rajapintoja voidaan tehdä C#-kielellä, miten niitä voidaan käyttää ja miten ne toimivat.

5.1 OWIN ja Katana

OWIN eli Open Web Interface for .NET on standardi .NET-verkkopalvelimien ja web-sovellusten välillä. Sen tavoitteena on irrottaa palvelin ja sovellus toisistaan. OWIN ei itsessään ole mikään paketti tai kirjasto, minkä avulla rajapinta voitaisiin tehdä, vaan se on malli tai määritelmä siitä, miten se tulisi tehdä. (14.)

Katana on kokoelma Microsoftin komponentteja, jotka toteuttavat OWIN-standardia. Katana-ohjelmistokehystä käyttäen voidaan tehdä ja ylläpitää (engl. host) web-ohjelmia omissa prosesseissaan. (15.)

5.2 Web-rajapinnan luonti

Web-rajapintojen luonti on todella helppoa Katana-kirjaston kanssa. Tässä osiossa esitellään, miten voidaan luoda yksinkertainen rajapinta ja käyttää sitä omassa prosessissa. Virranmittausjärjestelmän rajapinnat ovat toteutettu tällä tavalla. (16.)

Ensimmäisenä on haettava paketit, joita käytetään luomaan ja ylläpitämään rajapinta. Nämä paketit ovat saatavissa mm. NuGet-paketinhallintaohjelmasta. Seuraavaksi on luotava luokka, jota käytetään konfiguroimaan rajapinta. Tämä luokka tulee toteuttaa OWIN-standardin mukaisesti.

```
1.  public class Startup
2.  {
3.      public void Configuration(IAppBuilder appBuilder)
4.      {
5.          // Configure Web API for self-host.
6.          HttpConfiguration config = new HttpConfiguration();
```

```
7.          config.Routes.MapHttpRoute(
8.              name: "DefaultApi",
9.              routeTemplate: "api/{controller}/{id}",
10.             defaults: new { id = RouteParameter.Optional }
11.         );
12.
13.         appBuilder.UseWebApi(config);
14.     }
15. }
```

Sitten lisätään Katana-verkkopalvelin toteutus ja annetaan sille konfiguraatioluokka ja osoite, mitä halutaan käyttää ohjelmalla.

```
1.  static void Main(string[] args)
2.  {
3.      using (WebApp.Start<Startup>(url: "http://localhost:9000/"));
4.  }
```

Nyt projekti on kykenevä ylläpitämään rajapintoja omassa prosessissaan, eikä tarvitse käyttää muita raskaampia tapoja. Projektiin voidaan lisätä rajapintoja helposti. Uuden rajapinnan projektiin voi lisätä luomalla luokan, joka perii (engl. extends) ApiController-luokan.

```
1.  public class ValuesController : ApiController
2.  {
3.      // GET api/values
4.      public IEnumerable<string> Get()
5.      {
6.          return new string[] { "value1", "value2" };
7.      }
8.  }
```

Luotua rajapintaa voidaan nyt kutsua esimerkiksi Postman-ohjelmalla seuraavasti: GET http://localhost:9000/api/values. Tämä kysely ohjautuu ValuesController-luokan Get()-metodiin, joka palauttaa kaksi merkkijonoa value1 ja value2.

5.3 Rajapinta ja parametrit

Rajapinnan metodille voidaan määritellä parametrejä monilla eri tavoin. Tässä osiossa esitellään muutamia erilaisia tapoja lisätä parametrejä rajapinnan metodeille. Seuraavat esimerkit toimivat aiemmin esitellyssä ValuesController-luokassa ja niitä voidaan myös yhdistellä keskenään.

5.3.1 id kyselyn lopussa

Aiemman luvun Startup-luokassa on määritelty routeTemplate: api/{controller}/{id}. Tämä tarkoittaa, että kyselyt (engl. request), joiden perässä on jotain, reititetään datatyypin mukaan rajapinnan

metodiin, joka määrittää parametrin id samalla datatyypillä. Esimerkiksi alla olevaan metodiin reititetään kaikki POST-kyselyt, joiden kyselyjonon (engl. query string) perässä on joku numero.

```
1.  public string Post(int id)
2.  {
3.      return "value" + id;
4.  }
```

Kyselyjono, jonka perässä on id, näyttäisi tältä: http://localhost:9000/api/values/5. Tämä kysely palauttaisi merkkijonon: value5. Tämä sama tapa toimii, myös muilla datatyypeillä, esimerkiksi String, Guid jne.

5.3.2 Parametri kyselyjonosta

Alla olevaan metodiin reititetään kaikki kyselyt, joissa on joku merkkijono eli string-parametri kyselyjonon lopussa.

```
1.  public String Post([FromUri]string value)
2.  {
3.      return value;
4.  }
```

Metodiin voidaan lähettää kysely parametrin kanssa seuraavalla kyselyllä: http://localhost:9000/api/values?value=string. Tämä kysely palauttaisi merkkijonon: string.

5.3.3 Parametri kyselyn rungosta

Voidaan myös määrittää, että parametri tulee kyselyn rungosta (engl. body). Kaikki kyselyt, joiden rungossa on jokin merkkijono, reititettäisiin tähän metodiin.

```
1.  public String Post([FromBody]string value)
2.  {
3.      return value;
4.  }
```

Esimerkiksi seuraava kysely palauttaisi string-merkkijonon:

```
curl --header "Content-Type: text/plain"
 --request POST
 --data "string"
 http://localhost:9000/api/values
```

5.3.4 Malli

Kun kyselyyn halutaan useita parametrejä, voidaan apuna käyttää mallia (engl. model). Ensin on luotava luokka, johon määritetään parametrejä.

```
1.  public class ValuesModel
2.  {
3.      public String Name;
4.      public int Value;
5.  }
```

Nyt kaikki kyselyt, joilla on runko, reititetään tähän metodiin ja kyselyn rungossa ollut data löytyy mallista.

```
1.  public ValuesModel Post(ValuesModel model)
2.  {
3.      return new ValuesModel(model.Name, model.Value);
4.  }
```

Esimerkiksi seuraava kysely reititettäisiin ylläolevaan metodiin.

```
curl --header "Content-Type: application/json"
 --request POST
 --data '{"Name":"string","Value":16}'
 http://localhost:9000/api/values
```

Malleja voidaan myös käyttää palauttamaan dataa, esimerkiksi yllä oleva kysely palauttaa:

```
{"Name":"string", "Value":16}
```

5.4 Reititys

Reitityksellä (engl. routing) tarkoitetaan käskyjen ohjausta haluttuihin metodeihin, toimintoihin tai resursseihin. Tässä luvussa esitellään kaksi tapaa, miten kyselyjä reititetään ja miten niitä voidaan käyttää.

5.4.1 Convention based routing

Ylläolevissa esimerkeissä reititys tapahtui Startup-luokassa määritetyn routeTemplate mukaan eli routeTemplate: api/{controller}/{id}. Tämä määrittää, että kyselyt reititetään ohjaimeen (engl. controller) ja mahdollisen id:n perusteella.
Tähän kaavaan voidaan myös lisätä toiminto (engl. action) eli joku metodi seuraavasti: api/{controller}/{action}/{id}. Kaavaan voidaan myös lisätä oletusarvoja ja rajoituksia parametreille. (17.)

5.4.2 Attribute routing

Attribute routing mahdollistaa reitityksen hallinnan ohjain-tasolla, kun convention based routing vaikuttaa jokaiseen rajapintaan ohjelmassa. Molemmat reititystavat toimivat myös yhdessä. (18.)

Attribute routing voidaan sallia lisäämällä projektin Startup-luokkaan seuraava käsky:

```
1.  config.MapHttpAttributeRoutes();
```

Nyt reititystä voidaan hallita lisäämällä C#-attribuutteja ohjaimelle ja sen metodeille.

```
1.  [RoutePrefix("api/valuesAPI")]
2.  public class ValuesController : ApiController {
3.  .
4.  .
5.  .
6.      [Route("Time")]
7.      [HttpGet]
8.      public DateTime Time()
9.      {
10.         return DateTime.Now;
11.     }
12. .
13. .
14. .
15. }
```

Nyt seuraava kysely reitittyisi tähän ohjaimeen ja metodiin: GET http://localhost:9000/api/valuesAPI/Time. Oletusarvoja ja rajoituksia voidaan myös lisätä metodi- ja ohjain-tasolla.

5.5 HTTP-tilakoodit

Kun mittausohjelman rajapintaan lähetetään kutsuja tai käskyjä, sen täytyy palauttaa HTTP-tilakoodeja (engl. status codes). Kun käsky onnistuu, palautetaan tilakoodi 200, ja kun epäonnistuu tai tapahtuu jokin virhe, palautetaan tilanteesta riippuen eri tilakoodi. Esimerkiksi jos jokin pyydetty resurssi puuttuu, tulee palauttaa 404-tilakoodi, eli ”Not found”.

Tilakoodit ovat tärkeä osa web-rajapintaa, koska niiden avulla palautetaan käyttäjälle tietoa, siitä mitä ohjelmassa tapahtuu. Esimerkiksi jos tilakoodia 200 ei palautettaisi, ei käyttäjä voi tietää, onko haluttu toiminto tapahtunut vai ei.

6 KÄYTETTYJÄ SUUNNITTELUMALLEJA

Suunnittelumallit (engl. design patterns) ovat ratkaisuja ohjelmistokehityksessä useasti esiintyviin ongelmiin. Tässä luvussa esitellään kaksi suunnittelumallia, joita käytettiin projektissa. Molemmista suunnittelumalleista näytetään myös esimerkki toteutus. (19.)

6.1 Singleton-suunnittelumalli

Singleton-suunnittelumallissa jollain luokalla voi olla vain yksi olio tai ilmentymä (engl. instance). Tähän olioon tulee päästä helposti käsiksi luomalla sille jokin tapa, jolla se voidaan hakea globaalisti, esimerkiksi staattinen property tai metodi, joka palauttaa olion referenssin. Singleton-suunnittelumallin etuja ovat mm. sen toteutusnopeus ja joustavuus. (19.)

Singletoneja voidaan luoda monilla eri tavoilla. Tässä on esitetty yksi tapa, jolla singleton voidaan tehdä C#-kielessä.

```
1.  public sealed class MeasurementControl
2.  {
3.      static readonly MeasurementControl _instance = new MeasurementControl();
4.      public static MeasurementControl Instance
5.      {
6.          get
7.          {
8.              return _instance;
9.          }
10.     }
11.     private MeasurementControl()
12.     {
13.         // Initialize
14.     }
15. }
```

Singleton-luokassa on käytetty sealed-avainsanaa. Sen avulla estetään luokan periminen. Luokan-rakentaja (engl. constructor) on yksityinen (engl. private), jotta sitä ei voida kutsua luokan ulkopuolelta. _instance-muuttuja on myös yksityinen, eli sitäkään ei voida hakea luokan ulkopuolelta. Sillä on readonly-avainsana, eli se voidaan vain lukea. Instance propertyä käytetään, kun halutaan päästä käsiksi singleton-luokkaan. Tämä property on staattinen, eli se kuuluu luokalle tai tyypille eikä tietylle oliolle tai instanssille ja palauttaa _instance-muuttujan referenssin. (20.)
Staattista propertyä voidaan kutsua seuraavasti saamaan referenssi singleton-luokalle:

```
1.  MeasurementControl measurementControl = MeasurementControl.Instance;
```

6.2 Rakentaja-suunnittelumalli

Rakentaja-suunnittelumalli (engl. builder design pattern) on monimutkaisempi kuin luvussa 6.1 esitelty singleton-suunnittelumalli. Rakentaja-suunnittelumallia käytetään rakentamaan (engl. construct) monimutkaisia olioita ja usein myös eri versioita rakennettavasta tyypistä. Tämän suunnittelumallin etuja ovat mm. sen monikäyttöisyys ja parempi olioiden luonnin hallinta. (19.)

Samoin kuin singletonien kanssa rakentaja-suunnittelumallin voi toteuttaa monella eri tavalla. Tässä luvussa esitellään projektissa käytetty toteutus. Rakentaja-suunnittelumallissa on ohjaaja, joka kutsuu tarvittavia metodeja rakentajissa ja palauttaa luodun olion lopuksi.

```
1.  public class BuildDirector
2.  {
3.      public MeasurementConfiguration Build(MeasurementBuilder builder)
4.      {
5.          builder.createChannels();
6.          return builder.Build();
7.      }
8.  }
```

Rakentajia voi olla useita ja ne jakavat yhteisiä ominaisuuksia. Esimerkiksi tässä projektissa kaikkiin mittauksiin määritellään näytteenottokello samalla tavalla. Tämän takia kaikki rakentajat perivät abstraktin (engl. abstract) rakentajan, joka luo tehtävät, määrittää kellot ym. Tässä abstraktissa luokassa on määritetty metodit, jotka perivän luokan on toteutettava.

```
1.  public abstract class MeasurementBuilder
2.  {
3.      protected MeasurementConfiguration configuration;
4.      public abstract void createChannels();
5.      public MeasurementConfiguration Build()
6.      {
7.          // Build configuration
8.          return configuration;
9.      }
10. }
```

Rakentajat toteuttavat perityn luokan abstraktin metodin omalla toteutuksellaan.

```
1.  public class CurrentMeasurementBuilder : MeasurementBuilder
2.  {
3.      public override void createChannels()
4.      {
5.          // Create current measurement channels
6.      }
7.  }
```

Esimerkiksi tässä projektissa kanavat luodaan erilaisille mittauksille eri tavoin. Vaikka projekti on tarkoitettu virranmittauksiin, rakentaja-suunnittelumalli mahdollistaa helposti erilaisten mittausten, esimerkiksi lämmön, lisäämisen.

```
1.  public class TemperatureMeasurementBuilder : MeasurementBuilder
2.  {
3.      public override void createChannels()
4.      {
5.          // Create temperature measurement channels
6.      }
7.  }
```

Rakentajia voidaan käyttää seuraavalla tavalla:

```
1.  BuildDirector director = new BuildDirector();
2.  MeasurementBuilder builder = new CurrentMeasurementBuilder();
3.  director.Build(builder);
```

7 DATAN TALLENNUS

Ollaan tekemässä virranmittaus eli datankeräysohjelmaa, jossa kerätyn datan tallennus on tärkeä vaihe ohjelmaa. Datan tallennukseen otettiin käyttöön SQLite-tietokanta. SQLite on C-kielellä tehty kirjasto, joka toteuttaa tietokantajärjestelmän. Tämä tietokantajärjestelmä on itsenäinen, se toimii kaikissa käyttöjärjestelmissä eikä siinä ole useita riippuvaisuuksia muihin ohjelmiin. Koko SQLite-kirjaston saa yhdessä tiedostossa ja se on helposti rakennettavissa. SQLite on myös serveritön, eli ei tarvita erillistä serveriprosessia ja datan luku ja kirjoitus tapahtuu suoraan levyllä oleviin tietokantatiedostoihin. SQLite ei myöskään vaadi erillistä asennusta, käynnistystä ym. SQLiten saa toimimaan monilla eri ohjelmointikielillä, mm. C/C++, C#, Java, Python jne. (21.)

7.1 Tietokannan luonti

SQLite-tietokanta eli tiedosto, johon voidaan tallentaa dataa, voidaan luoda ohjelman ajon yhteydessä seuraavasti:

```
1.  SQLiteConnection.CreateFile("ResultsDatabase.sqlite");
```

Kun tietokanta on luotu, voidaan siihen luoda taulut SQL-kyselykielellä.

```
1.  SQLiteCommand command = new SQLiteCommand("CREATE TABLE Table-
    Name (id INT, value INT)", databaseConnection);
2.  command.ExecuteNonQuery();
```

Tietokanta-yhteys voidaan avata komennolla:

```
1.  SQLiteConnection connection = new SQLiteConnection("Data Source=ResultsData-
    base.sqlite; Version=3;");
```

Vain yhden tietokanta-yhteyden käyttö ja sen jakaminen kaikkien operaatioiden kesken parantaa suorituskykyä. Sen takia projektissa tietokanta-yhteys avataan singleton-luokassa, josta yhteys voidaan hakea.

Yhteyden avauksen yhteydessä voidaan yhteyteen myös lisätä Pragma-lauseita. Näillä lauseilla voidaan muuttaa tietokannan toimintaa. Esimerkiksi PRAGMA temp_store = MEMORY voi parantaa suorituskykyä, koska käytetään nopeampaa muistia väliaikaisten tiedostojen tallennukseen.

7.2 Datan kirjoitus

Datan tallennus tietokantaan tapahtuu käyttämällä SQL-kyselykieltä, esimerkiksi:

```
1.  SQLiteCommand command = new SQLiteCommand("insert into Table-
    Name (id, value) val-ues (1, 3)", databaseConnection);
2.  command.ExecuteNonQuery();
```

Kun dataa kirjoitetaan tietokantaan yllä olevan esimerkin mukaan, jokainen insert-kysely aloittaa uuden transaktion (engl. transactions) ja kirjoittaa datan tietokantaan eli levylle. Kun ollaan kirjoittamassa satoja tai tuhansia arvoja sekunnissa tietokantaan, ei pystytä tekemään tätä tarpeeksi nopeasti ja ohjelman toiminta hidastuu tai jumiutuu. Transaktioita voidaan aloittaa manuaalisesti ja lisätä siihen useita insert-kyselyjä. Kun transaktio lopetetaan, kirjoitetaan kaikkien insertien datat tietokantaan. Esimerkiksi:

```
1.  using (SQLiteTransaction mytransaction = connection.BeginTransaction())
2.  {
3.      using (SQLiteCommand insertSQL = new SQLiteCommand(connection))
4.      {
5.          for (int i = 0; i < 100; i++)
6.          {
7.              insertSQL.CommandText = "insert into TableName (id, val-ue) val-
    ues (?,?)";
8.              insertSQL.Parameters.AddWithValue("id", id);
9.              insertSQL.Parameters.AddWithValue("value", i);
10.             insertSQL.ExecuteNonQuery();
11.         }
12.     }
13.     mytransaction.Commit();
14. }
```

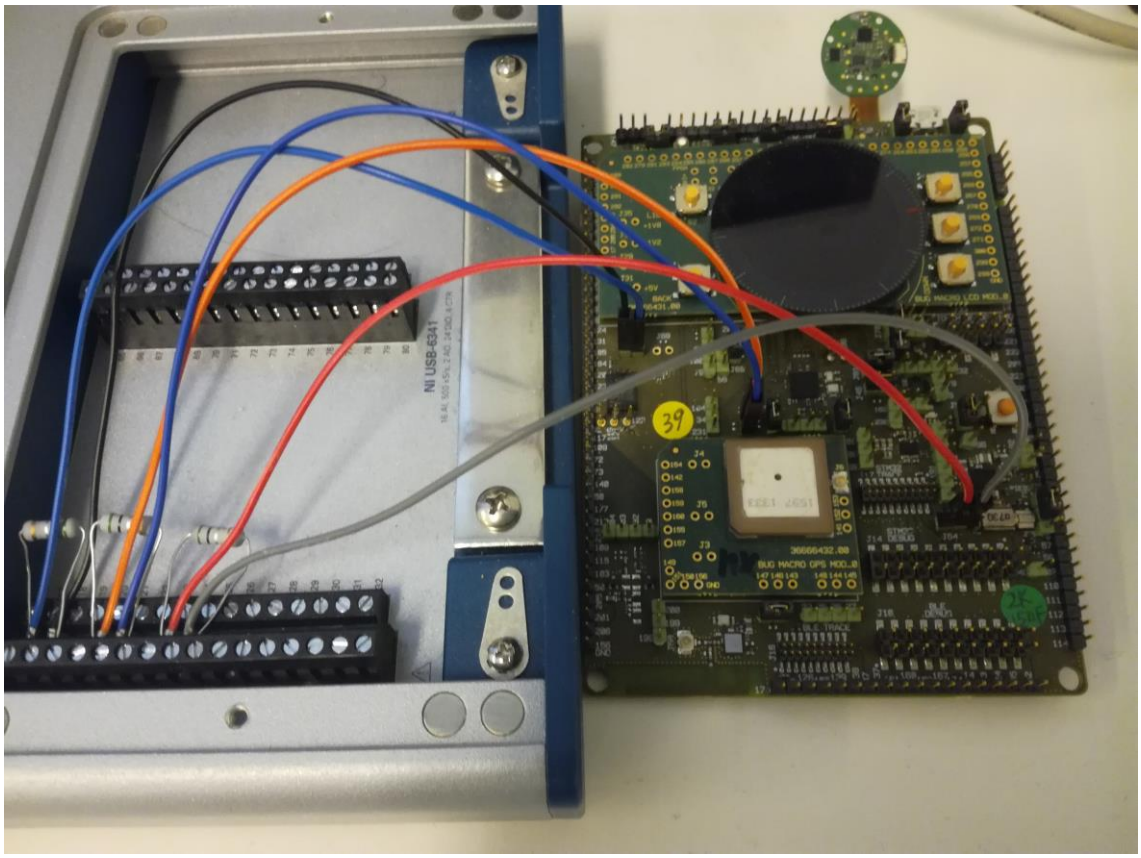
Koska tietokanta toimii tietokoneen levyllä ja kirjoitus tapahtuu suoraan tiedostoon, voidaan suorituskykyä parantaa myös käyttämällä nopeampaa kovalevyä, esim. SSD.

8 TOTEUTUS

Tässä osiossa käydään läpi kaikki tärkeimmät toteutuksen vaiheet, vastaan tulleet haasteet ja niiden ratkaisut.

8.1 Vastusten mitoitus ja kytkennän suunnittelu

Ensimmäisenä projektissa lähdettiin mitoittamaan vastuksia makrolle eli mitattavalle laitteelle, jotta voidaan käytännössä testata NI DAQ 9238 -mittalaitetta (kuva 4). Vastuksen kooksi valittiin 0,5 ohmia, koska se antaa riittävän tarkkuuden mittauksille eikä sen pitäisi häiritä mitattavan laitteen toimintaa. Tässä vaiheessa käytettiin valmiiksi saatavilla olevia vastuksia eli ei tarkkuusvastuksia. Vastuksien olisi hyvä olla tarkkuusvastuksia minimoimaan mukaan laskemattoman resistanssin määrää.



KUVA 7. USB-6341-mittalaite yhdistettynä kolmeen mittapisteeseen

8.2 Ohjelmien asennus ja versioiden valinta

Jotta mittalaitetta voidaan testata, sen käyttöön tarvitaan ajurit ja jokin ohjelma, jolla voidaan tehdä mittauksia. NI-DAQmx-ajurin sai ladattua National Instrumentsin sivuilta. Ajurin mukana tuli muutama sovellus mittalaitteiden testaukseen, kalibrointiin jne. Ajurin asennuksen yhteydessä pystyi myös asentamaan .NET-kirjaston, joka tarvitaan virranmittausjärjestelmän tekoon. Kirjaston mukana tuli myös esimerkkiohjelmia kirjaston käytöstä. NI-DAQmx-ajurin mukana asennettu kirjasto tukee vain .NET 4.5.1 -versiota ja en huomannut tarkistaa projektin alussa, onko .NET-ohjelmistokehys taaksepäin yhteensopiva. Tämän takia koko projekti on tehty .NET 4.5.1 -versiolla ja kaikki muut kirjastot käyttävät 4.5.1-yhteensopivaa versiota.

8.3 Mittalaitteen toiminnan testaaminen ja NI-DAQmx-C#-kirjaston käyttö

Kun vastukset oli saatu mitoitettua ja mitattava laite kytkettyä mittalaitteeseen, oli aika testata mittalaitteen toimintaa. NI-DAQmx-ajurin mukana tuli muutama ohjelma, joilla voidaan testata laitteita, mutta näillä ohjelmilla ei pystynyt tekemään mittauksia. Tässä vaiheessa haluttiin saada tehtyä yksinkertainen virranmittaus, jonka keskiarvoa voidaan verrata yleismittarilla saatuun keskiarvoon. Tämän mittauksen tekoon käytettiin SignalExpress-ohjelmaa. SignalExpress on NI:n tekemä ohjelma, jolla voidaan nopeasti kerätä dataa mittalaitteista. Tämän mittauksen tarkoituksena oli myös saada testattua monen kanavan samanaikaista mittausta ja varmistaa, että mittalaitetta voidaan käyttää virranmittausjärjestelmän tekoon. Kaikki toimi kuten pitikin, pystyttiin mittaamaan useaa kanavaa samaan aikaan ja saatu keskiarvo kokonaisvirran kulutuksesta oli samaa luokkaa kuin yleismittarilla saatu keskiarvo.

Seuraavaksi lähdettiin testaamaan C#-kirjastoa NI-DAQmx-ajurille. Tässäkin haluttiin tehdä kokonaisvirran mittaus muutaman minuutin ajalta ja verrata tuloksia yleismittarilla saatuun keskiarvoon. Tämä mittaus tehtiin käyttämällä NI:n tekemää C#-esimerkkiohjelmaa. NI:llä on useita esimerkki ohjelmia erikielisille kirjastoille. Ohjelmat näyttävät, miten mittalaitteita voidaan käyttää ohjelmallisesti eri mittausten (erilaisen datan keräyksen) tekemiseen, esimerkiksi virranmittaus, jännitemittaus ja lämmönmittaus. Käytetty esimerkkiohjelma oli yksinkertainen datan keruu ohjelma, joka piirsi kerättyä dataa käyräksi. Tätä esimerkkiohjelmaa lähdettiin muokkaamaan, jotta se voisi laskea keskiarvon tuloksista.

Kun esimerkkiohjelma oli saatu laskemaan keskiarvoa oikein ja tulokset vahvistettu olevan samaa luokkaa kuin yleismittarin keskiarvo, oli aika tehdä esimerkkiohjelma, jolla voidaan tehdä erilaisia mittauksia. Tämän ohjelman tarkoituksena oli opetella NI-DAQmx-C#-kirjaston käyttöä ja samalla aloittaa ensimmäisen komponentin teko virranmittausjärjestelmälle. Komponentilla tässä tarkoitetaan kokoelmaa luokkia, jotka voidaan kopioida lopulliseen ohjelmaan. Tällä ensimmäisellä tehdyllä komponentilla voidaan mitata virtaa, asettaa mittauksille tietty kesto ja rakentaa mittauksia.

8.4 Mittauskomponentin toiminta

Tässä osiossa käydään hieman läpi mittauskomponentin toimintaa. Mittauskomponentissa on kolme tärkeää osiota, mittauksensuorittaja, mittaustehtävä ja sen rakentaja.

8.4.1 Mittaustehtävä

Mittaustehtävä on luokka, joka sisältää kaiken tiedon tehtävästä mittauksesta, eli mittauskohteen, vastusten arvot, minimi- ja maksimivirrat, mittauksen näytteenottonopeus jne. Kun se luodaan, se hakee itselleen uniikin ID:n, jota voidaan käyttää hallitsemaan mittausta web-rajapinnasta. Se myös sisältää yhden tai useamman DAQmx-tehtävän (engl. task). Task on NI-DAQmx-C#-kirjaston käyttämä luokka, johon mittauksen tarvitsemat laitteet, kanavat ja näytteenottokellon asetukset määritellään. Tätä DAQmx-tehtävää käytetään, kun halutaan luoda datanlukija tai datankirjoittaja mittalaitteille. Mittaustehtävä tulee varmistuttaa (engl verify), koska sen sisältämät DAQmx-tehtävät täytyy varmistuttaa, ennen kuin sitä voidaan käyttää. Samalla varmistetaan, että mittaukselle on asetettu kesto, koska mittauksilla halutaan olevan jokin timeout-aika, joka pysäyttää mittauksen itsestään. Mittauksella on oltava kesto, koska jos jostain syystä käyttäjä ei pääsisi pysäyttämään mittausta, olisi mittalaite aina varattuna. Tätä varten on myös tehty admin-rajapinta erikseen, jolla voidaan pysäyttää ja poistaa kaikki mittaukset. Jos tehtävän varmistus ei mene läpi, palautetaan virhe ja mittaus poistetaan.

8.4.2 Mittalaitteen ja näytteenottokellon määrittäminen DAQmx-tehtävälle

DAQmx-tehtävälle tulee määrittää laitteet ja kanavat, joita halutaan lukea tai joihin halutaan kirjoittaa. Tämä määrittely tehdään käyttämällä merkkijonoja, jotka vastaavat jotain mittalaitetta tietokoneella. Nämä merkkijonot näyttävät esimerkiksi tältä: cDAQ1/MOD1/AI1. cDAQ1 on laitteen tai kotelon nimi ja numero. MOD1 on laitteen moduuli, jota käytetään. Moduuleja voi olla yhdessä kotelossa useita tai ei yhtäkään. AI1 vastaa moduulin kanavaa, jota halutaan käyttää. Nämä merkkijonot luodaan laitteille siinä järjestyksessä, jossa ne on asennettu tietokoneeseen ja ovat nähtävissä mm. SignalExpress-ohjelmalla.

Yhdessä DAQmx-tehtävässä voi olla monta laitetta, mutta vain yksi näytteenottokello. Mm. tämän takia jokaiselle laitteelle luodaan oma DAQmx-tehtävä, koska saatetaan haluta tehdä mittauksia eri nopeuksilla, eri laitteilla. Kun DAQmx-tehtävään on kanavat ja kello määritetty, se tulee varmistuttaa (engl. verify). Tässä tarkistetaan, että kaikki parametrit ovat kunnossa laitteille. Jos vahvistus onnistuu, laitteet siirretään varattutilaan. Kun laite on varattu, sitä ei muut tehtävät voi käyttää ja heitetään virhe, jos joku toinen tehtävä yrittää lisätä varatun laitteen kanavia. Tämä tehtävä täytyy poistaa, ennen kuin laite vapautuu muiden tehtävien käyttöön.

8.4.3 Mittaustehtävänrakentaja

Mittaustehtävälle tuli tehdä rakentaja (engl. builder), koska mittaustehtävät tarvitsevat paljon tietoa ja tarvittava tieto voi laitteesta ja mittauksesta riippuen muuttua. Rakentaja on tehty rakentaja-suunnittelumallia (engl. builder design pattern) käyttäen ja on tämän ansiosta helposti muokattavissa. Siihen voidaan myös helposti lisätä erilaisia mittauksia. Rakentaja huolehtii mm. siitä, että jokaiselle laitteelle luodaan oma DAQmx-tehtävä ja määrittää kanavat ja näytteenottokellon tehtäville.

8.4.4 Mittauksensuorittaja

Mittauksille on tehty myös suorittaja, joka hoitaa mittauksen ajon, pysäytyksen ja poiston. Tätä suorittajaa voidaan kutsua rajapinnasta mittaustehtävän uniikilla ID:llä. Mittauksensuorittajalla on kello, joka pysäyttää mittauksen, kun määritetty mittausaika on mitattu. Suorittajalla on myös erillinen luokka, joka hoitaa datan lukemisen ja palauttaa suorittajalle ko. datan tietyin väliajoin. Suorittaja välittää datan sitten tietokannan puskuriin.

8.5 Palvelun teko

Kun virranmittauskomponentti oli valmis, lähdettiin tutkimaan parasta tapaa tehdä ohjelma. Kuten määrittelyvaiheessa mainittiinkin, koska ollaan muutenkin tekemässä Windows-ohjelma, voitaisiin ohjelmasta tehdä Windows-palvelu ja näin saada samalla ohjelman käynnistäminen ja sammuttaminen helpommaksi.

Tässä osiossa testattiin Windows-palvelun tekoa Visual Studio 2013 Windows service templatea käyttäen, mutta template-ohjelma vaikutti turhan monimutkaiselta. Internetistä löytyi Topshelf-ohjelmistokehys, jonka avulla voidaan tehdä Windows-palveluita todella helposti, ja se myös helpottaa palvelun kehitystä. Topshelfistä tuli asentaa versio 3.3.1, koska nykyinen versio (4.2.0) ei enää tue .NET 4.5.1 -versiota. Kun tämä saatiin asennettua ja tehtyä Windows-palvelu Topshelfillä testattiin, sen toimintaa ja varmistettiin, että sen saa mm. asennettua Windowsille. Seuraavaksi lisättiin edellisessä vaiheessa tehty komponentti tähän palveluun ja varmistettiin, että virranmittaus toimii asennetussa palvelussa. Topshelfin avulla Windows-palvelun asennus, poisto ja muu hallinta on myös paljon helpompaa.

8.6 Web-rajapinta Windows-palveluun

Tässä vaiheessa tutkittiin parasta tapaa saada web-rajapinta tehtyä Windows-palvelulle. Web-rajapinnan tekoon löytyvä template oli taas turhan monimutkainen ja sen toimimaan saanti Windows-palvelussa (engl. service) ei välttämättä olisi ollut edes mahdollista. Microsoftin dokumentaatiosta löytyi Katana-ohjelmistokehys, joka on Microsoftin toteutus OWIN:iin eli Open Web Interface for .Net. Katanan dokumentaatiosta ja GitHubista löytyi paljon esimerkkejä, miten sitä voidaan käyttää ja sen käyttöönotto oli todella yksinkertaista.

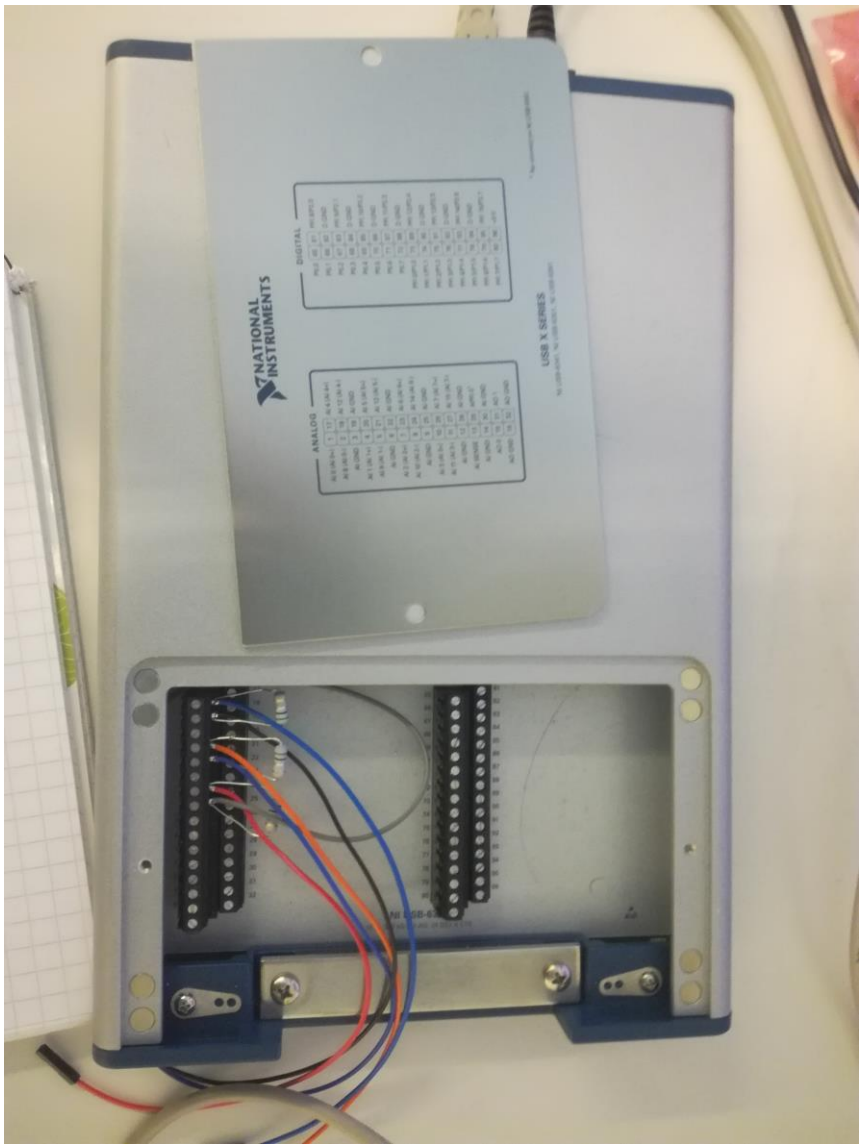
Katana on paketti, jonka voi lisätä projektiin. Kun sen käynnistykseen liittyvät luokat on lisätty projektiin, voidaan tehdä rajapintoja todella helposti. Uuden rajapinnan voi lisätä projektiin luomalla uusi luokka, joka perii (engl. extends) ApiController-luokan Katana-paketista. Kun tämä saatiin toimimaan esimerkkiohjelmassa ja opeteltua hieman, miten tähän voidaan lisätä erilaisia rajapintoja ja metodeja, oli aika lisätä tämä paketti lopulliseen projektiin.

8.7 Komponenttien kokoaminen

Nyt kun kaikki tärkeimmät osat projektista, eli palvelu Topshelfiä käyttäen, NI-DAQmx-ajurin käyttö palvelussa ja web-rajapinta palvelussa saatiin toimimaan yhdessä, oli aika koittaa saada mittaus aloitettua web-rajapinnan kautta. Tätä varten luotiin ensin yksinkertainen metodi rajapintaan, joka aloittaa mittauksia. Mittaus saatiin aloitettua, mutta tässä huomattiin ongelmia mittausten hallinnan kanssa.

Mittausten hallinta eli mittauksen pysäytys, poisto jne. web-rajapinnan kautta ei toiminut kuten alun perin lähdettiin tekemään. Tarkoituksena oli pitää referenssi mittauksensuorittajille listassa rajapinnan ohjaimessa. Ohjain sitten hakisi suorittajan referenssin listasta, mittauksen uniikkia ID:tä käyttäen ja tekisi halutun operaation suorittajalle. Edellä kuvailtu toiminta tapa ei toimi, koska rajapinnan ohjain luodaan aina uudestaan jokaisella käskyllä, eli referenssit eivät säily. Tämän toiminnan voisi muuttaa käyttäen mm. dependency injectiota, mutta tämä vaikutti turhan vaikealta saada toimimaan oikein. Tämä ongelma ratkaistiin luomalla mittauskomponentille oma singleton-luokka, jolla mittauksia voidaan hallita. Singleton-luokka luodaan vain kerran ja on aina olemassa ja se voidaan tarvittaessa hakea rajapinnassa. Tämä singleton-luokka pitää listassa kaikki olemassa olevat mittausten suorittajat ja vastaan ottaa käskyjä rajapinnasta. Tämä luokka on vastuussa mm. mittausten suorittajien luomisesta, aloituksesta, pysäytyksestä ja poistosta.

Tässä vaiheessa saatiin myös toinen mittalaite käyttöön ja jouduttiin kirjoittamaan osa virranmittauskomponenttia uudestaan, koska usean mittalaitteen käyttö samaan aikaan ei toiminut hyvin. Tämä mittalaite oli NI USB-6341. (Kuva 8.)



KUVA 8 USB-6341-mittalaite

8.8 Mittausongelmat

Datan lukemisen eli virranmittauksen kanssa oli paljon ongelmia, varsinkin kun kaikki data haluttiin tallentaa. Jos dataa luetaan liian nopeaa, ei datan kirjoitus tietokantaan tai tiedostoon pysy mukana. Tämä aiheuttaa mittauksen menemisen jumiin ja jossain vaiheessa mittalaitteen puskurin täyttymisen, joka aiheuttaa mittauksen kaatumisen. Näitä ongelmia tuli koko ajan vastaan ja näitä koitettiin ratkaista optimoimalla ohjelman toimintaa mm. kirjoittamalla data ensin puskuriin ja käyttämällä eri säikeitä eri tehtäville.

8.9 Konfiguraatiot

Konfiguraatiot tai asetukset vähentävät palvelun käyttäjän tarvitsemaa tietoa mittausten tekoon. Käyttäjä voi kertoa palvelulle mittauksen pituuden, näytteenottonopeuden ja mittauskohteet. Mittauksenrakentaja sitten hakee muut tarvittavat tiedot konfiguraatioista. Konfiguraatiot on tehty JSONia (JavaScript Object Notation) käyttäen, koska JSON-tiedostot voidaan suoraan lukea luokkiin ja saada helposti tarvittava tieto käyttöön.

```
1.  DAQConfig daqConfig;
2.  daqConfig = JsonConvert.DeserializeObject<DAQConfig>(jsonString);
```

Konfiguraatioiden avulla palvelun käyttäjä voi sanoa rajapinnalle esimerkiksi ”mittaa GPS komponentin virrankulutusta laitteesta, jonka ID on XXCCXXCC”. Mittauksenrakentaja hakee konfiguraatioista tämän laitteen tyypin. Laitteen tyyppiä voidaan sitten käyttää löytämään GPS-komponentin mittapisteen ID, minimi- ja maksimivirrat. Komponentin ID:tä ja laitteen ID:tä voidaan sitten käyttää löytämään mittalaite ja kanava, johon tämä komponentti on yhdistetty ja vastuksen koko tässä kanavassa.

Mittalaitteiden konfiguraatiota käytetään myös rakentamaan mittalaitteen nimeävä merkkijono, jota käytetään tunnistamaan mittalaitteet toisistaan. Tässä on myös huomioitava erilaiset laitteet, esimerkiksi joissain laitteissa on useita moduuleja ja toisissa ei ole moduulia ollenkaan, eli konfiguraation on pystyttävä rakentamaan merkkijono kaikenlaisille mittalaitteille.

Palvelussa on myös neljäs konfiguraatio, jossa määritellään muiden konfiguraatioiden nimet ja sijainnit. Tässä tiedostossa asetetaan myös tietokannan nimi, web-rajapinnan osoite ja portti. Konfiguraatioita ei voi muuttaa palvelun ollessa ajossa, vaan se pitää pysäyttää ensin.

8.10 Logging

Logitus on myös tärkeä osa sovellusta. Sen avulla voidaan tutkia esimerkiksi, onko mittauksissa tapahtunut virheitä ja mitä ohjelma on tehnyt virheen tapahtuessa. Logitus on toteutettu log4net-kirjastoa käyttäen. Logituksessa käytettiin rolling file -metodia, jossa määritetään logi-tiedostolle tietty koko. Kun määritetty tiedoston koko on saavutettu, luodaan uusi puhdas logi-tiedosto ja vanha tallennetaan. Tätä toistetaan, kunnes määritelty tiedostojen lukumäärä saavutetaan, jolloin vanhin

tallennettu logi-tiedosto poistetaan. Näin sovelluksen ei tarvitse huolehtia logitukseen kuluvasta levytilasta. Topshelf:in saa myös helposti toimimaan log4netin kanssa.

8.11 Web-rajapinta

Web-rajapintojen lisääminen projektiin on todella helppoa Katana-paketin kanssa. Tässä luvussa käydään läpi, minkälaisia rajapintoja palveluun on tehty.

8.11.1 Statuses-rajapinta

Ohjelmaan tarvitaan ohjelman tila -rajapinta (engl. statuses API). Tämän rajapinnan tarkoituksena on antaa käyttäjälle tieto siitä, onko palvelu päällä. Jos rajapinta ei anna vastausta, ei palvelu ole toiminnassa. Rajapinnassa on myös metodi, jonka avulla voidaan selvittää, onko joku tietty mittaus vielä menossa vai onko se pysähtynyt. Rajapinnan toinen metodi hakee kaikkien mittausten tilat ja myös kaikkien konfiguroitujen mittalaitteiden tilan. Tätä metodia voidaan myös käyttää tarkastelemaan, onko jokin mittalaite käytettävissä. Jos konfiguroitu mittalaite ei esiinny tämän metodin palauttamassa datassa, se ei ole NI-DAQmx-ajurin löydettävissä, eli se on pois päältä tai sitä ei ole konfiguroitu.

Mittalaitteiden tila kertoo, onko jokin mittalaite varattu vai ei, kun mittaus käynnistetään yhdellä mittalaitteen kanavalla, NI-DAQmx varaa koko mittalaitteen. Jos uusi mittaus koitetaan aloittaa tai koitetaan lisätä kanava ajossa olevaan mittaukseen, ajuri palauttaa virheen. Kun mittauksia aloitetaan, täytyy kaikki kanavat lisätä mittaukseen samaan aikaan, tämä täytyy ottaa huomioon rajapintoja tehdessä, mittauksienrakentajassa, datan tallennuksessa ja mittauksien ajossa.

8.11.2 Admin-rajapinta

Admin-rajapinta on tarkoitettu hallitsemaan kaikkia palvelun komponentteja. Esimerkiksi jos mittauksen uniikki ID on kadonnut, voidaan tällä rajapinnalla pysäyttää ja poistaa kaikki olemassa olevat mittaukset.

8.11.3 Measure-rajapinta

Measure-rajapinta on tarkoitettu hallitsemaan mittauksia. Tässä rajapinnassa on metodit mittausten luomiseen, aloittamiseen, pysäyttämiseen ja poistamiseen. Rajapintaa voidaan myös käyttää tarkistamaan, voidaanko jokin mittaus suorittaa. Mittausten luomiseen on kaksi metodia. Toinen luo ja aloittaa mittauksen ja palauttaa käyttäjälle mittauksen uniikin ID:n. Toinen metodi luo mittauksen ja varaa mittalaitteet, mutta ei aloita mittausta. Käyttäjä voi lähettää luodun mittauksen ID:n start nimiselle metodille, joka sitten aloittaa mittauksen.

Mittauksia voidaan hallita uniikilla ID:llä, joka palautetaan käyttäjälle mittauksen luonnin yhteydessä. Tätä ID:tä käytetään myös hakemaan mittauksentulokset, kun mittaus on pysäytetty. Tätä rajapintaa voidaan myös käyttää hakemaan menossa olevan mittauksen tiedot, eli mitä se on mittaamassa.

8.11.4 Measurements-rajapinta

Measurements-rajapinta on tarkoitettu avaamaan konfiguraatioita käyttäjälle. Käyttäjä voi käyttää tätä rajapintaa saamaan tiedon siitä, mitä laitteita palveluun on konfiguroitu, mitä komponentteja voidaan mitata jne.

8.11.5 Results-rajapinta

Results-rajapinta on tarkoitettu mittaustulosten käsittelyyn. Se sisältää metodit tulosten valmisteluun, niiden lataukseen ja poistamiseen. Tulokset pitää valmistella eli kirjoittaa tiedostoon, ennen kuin ne voidaan laittaa zip-tiedostoon. Kun tulokset on valmisteltu käyttäjä voi ladata ne rajapinnasta. Rajapinnassa on erillinen metodi tiedostojen valmisteluun, koska tämä käsky voi viedä kauan aikaa, riippuen kerätyn datan määrästä.

Dataa poistetaan käyttämällä samaa ID:tä, mitä sen valmisteluun ja lataukseen käytetään. Kun tämä ID annetaan DELETE-metodille, poistetaan kaikki valmistellut tulokset koneelta ja kaikki kentät tietokannasta, joissa tämä ID esiintyy. Tätä rajapintaa ei voi käyttää mittauksen ollessa päällä tietokannan suorituskyvyn takia.

8.12 Datan tallennus

Kun mittalaitteelta luetaan dataa, se kirjoitetaan ensin puskuriin, jolle on asetettu jokin numero arvo. Kun arvo saavutetaan, eli tämän arvon verran on kirjoitettu tuloksia, käynnistetään datan kirjoitus. Puskuria käytetään, koska nopeammilla datan luku nopeuksilla saattaa, esimerkiksi tiedostoon kirjoitus olla vielä kesken ja ohjelma jää jumiin, koska sen täytyy odottaa omaa vuoroa kirjoittaa dataa, eli ohjelma ei pysy mittaus nopeuksien mukana. Puskurin käyttö ei poista ongelmaa kokonaan, mutta paransi toiminnallisuutta hieman, koska dataa kirjoitetaan havemmin.

Kehityksen aikana mittausten tuloksia kirjoitettiin suoraan tekstitiedostoon. Tämä tapa ei oikein toimi, kun palvelu otetaan käyttöön, koska käyttäjän ei haluta tarvitsevan tietää, missä tiedostossa minkäkin komponentin tulokset ovat ja tämä tapa on muutenkin todella epäkäytännöllinen. Datan tallennukseen otettiin käyttöön SQLite-tietokanta, mm. sen helpon käyttöönoton ja käytön takia. SQLite saadaan myös helposti toimimaan koneella, mihin palvelu asennetaan, koska se voidaan luoda käytön aikana (engl. runtime). Tämä tarkoittaa, että se tulee ohjelman mukana, eikä vaadi erillistä asennusta. Kun SQLite oli saatu toimimaan, huomattiin sen kanssa paljon suorituskyky ongelmia. Datan kirjoitus vei liian kauan ja aiheutti palvelun jumiin menon ja mittausten kaatumisen mittalaitteen puskurien täyttymisen takia.

Suorituskyky ongelmia koitettiin ratkaista ensin testaamalla erilaisia kirjoitus tapoja mm. käyttämällä transaktioneita ja bulk-insertejä, mutta mikään ei ratkaissut ongelmaa. Seuraavaksi koitettiin muuttaa SQLiten toimintaa tehokkaammaksi. Tämä tehtiin lisäämällä PRAGMA-lauseita tietokantayhteyden avauksen yhteydessä. Kun ongelmat jatkuivat, tutkittiin tallennukseen käytettyjä datatyyppejä. Kun mittalaitteelta saadaan dataa, on jokaisella arvolla aikaleima. Aikaleima tässä vaiheessa koetettiin tallentaa ihmisen luettavassa formaatissa, mikä vei paljon muistia ja hidasti kirjoitusta. NI:n C#-kirjaston tarjoama aikaleima on tarkempi kuin C#-kielen tarjoama. Siten siinä on myös enemmän dataa, joka tulee tallentaa. Aikaleima muutettiin tick-muotoon, eli yhdeksi suureksi numeroksi. Tämä toi suurimman parannuksen datan tallennuksen nopeuteen, mutta aikaleimasta menetettiin tarkkuutta tämän takia. Tämä tarkkuuden menetys voidaan korjata lisäämällä tietokantaan kenttä sekunnin murto-osille.

Tässä vaiheessa suorituskyky oli jo huomattavasti parempi, mutta ohjelma meni jossain vaiheessa jumiin, koska useat säikeet yrittivät kirjoittaa dataa samaan aikaan ja SQLite ei salli tätä. Tämä ratkaistiin muuttamalla puskurin toimintaa. Puskuri jaettiin kaikkien mittausten kesken, jotta datan

kirjoitus tapahtuu samasta säikeestä. Suorituskykyä voidaan myös parantaa, jos tietokanta yhteyksiä ei luoda useita. Tämän takia tietokanta-yhteys jaettiin kaikkien komponenttien kesken, tekemällä yhteydelle singleton-luokka, josta tietokanta-yhteys voidaan hakea.

Datan lukeminen tietokannasta vie myös kauan aikaa, mutta tätä ei ole vielä ehditty optimoida. Tämän takia datan lukeminen on estetty silloin kun mittaus on käynnissä. Dataa kirjoitetaan tietokannasta suoraan JSON-tiedostoon, jota myös kirjoitetaan samaan aikaan, mikä luultavasti on syy suorituskyky ongelmiin. Tässä vaiheessa tehdään myös aikaleimojen muutos takaisin luettavaan muotoon, mikä vie myös aikaa.

8.13 Kalibrointi

Projektiin oli myös suunniteltu kalibrointi vaihe, jossa olisi koitettu saada kaikki mittalaitteet antamaan samoja tuloksia ja yrittää saada myös mitattavat laitteet antamaan oikeita arvoja, koska niidenkin välillä on paljon eroja. Tähän vaiheeseen ei ikinä ehditty, mm. koska tietokannan kanssa tuli paljon ongelmia vastaan. NI:llä on ohjelma, joka tarjoaa apuja mittalaitteiden kalibrointiin.

9 TULOKSET JA POHDINTA

Projektin tavoitteena oli mahdollistaa komponenttien virranmittausten teko testiautomaatiojärjestelmästä ja näin mahdollistaa komponenttien virranmittauksen automatisointi. Tämä tarkoittaa, että virranmittausjärjestelmä tarvitsee mittalaitteen, joka kykenee mittaamaan useaa komponenttia samaan aikaan ja jota voidaan ohjata ohjelmallisesti. Jotta järjestelmää voidaan käyttää testiautomaatiojärjestelmästä, se tarvitsee rajapinnan, jolla mittauksia voidaan hallita.

Lopputuloksena projektissa oli järjestelmä, joka kykenee mittaamaan useaa mittalaitetta ja kanavaa samaan aikaan ja vastaanottamaan käskyjä muista ohjelmista ja laitteista. Virranmittausjärjestelmä on myös helposti laajennettavissa tekemään erilaisia mittauksia National Instrumentsin datankeräyslaitteilla.

Projekti oli todella laaja ja käytiin läpi monia osa alueita, kuten esimerkiksi vastusten mitoitus, rajapintojen teko, Windows-palvelut, National Instrumentsin laitteet ja niiden käyttö. Virranmittausjärjestelmässä saatiin kaikki toimimaan ja se on valmis käyttöönotettavaksi, mutta omasta mielestä jotkut osiot tuli huonosti tehtyä kiireen takia ja ne olisivat hyvä kirjoittaa uudestaan. Esimerkiksi konfiguraatioiden lukuun tehty luokka ja datan tallennukseen ja lukuun tehdyt osiot ovat aika epäselviä ja niissä on parantamisen varaa.

Projekti myöhästyi hieman aikataulusta. Esimerkiksi datan tallennuksen suorituskykyongelmien takia projektista jätettiin kalibrointivaihe pois. Opinnäytetyön dokumenttikin myöhästyi aikataulusta, koska olin arvioinut väärin, kuinka kauan kirjoittamiseen menee aikaa.

Järjestelmään jäi vielä tehtävää, esimerkiksi kalibrointi, tulosten haun optimointi ja aikaleimojen tarkkuuden parantaminen. Jos tuloksien hakua ei saada tarpeeksi nopeaksi, on rajapintaa laajennettava, jotta tuloksien valmistelua voidaan seurata. National Instrumentsin rajapinta DAQmx-ajurille on myös suuri ja siinä on ominaisuuksia, jotka voitaisiin vielä ottaa järjestelmässä käyttöön. Kun järjestelmä otetaan käyttöön, tulee rajapintoihin luultavasti myös lisättävää mm. enemmän virhekoodeja, erilaisille virhetilanteille, jotta testiautomaatio järjestelmä pystyy paremmin selvittämään, mistä virhe johtuu. Voisi myös olla hyödyllistä tutkia, voidaanko ohjelma nostaa (engl. upgrade) ylempään .NET versioon ja mitä hyötyä tästä olisi.

LÄHTEET

1. Volotinen , Vesa 1997, Analoginen elektroniikka, Komponentit ja peruskytkennät. Porvoo: WSOY.
2. Current Measurements: How-To Guide. 2017. National Instruments. Saatavissa: http://www.ni.com/tutorial/7114/en. Hakupäivä 22.4.2019.
3. Shunt resistor. Saatavissa: http://www.resistorguide.com/shunt-resistor. Hakupäivä 29.4.2019.
4. TechTip: Accuracy, Precision, Resolution, and Sensitivity. Measurement Computing. Saatavissa: https://www.mccdaq.com/TechTips/TechTip-1.aspx. Hakupäivä 29.4.2019.
5. Measurement of Current with a Voltage DAQ. 2018. National Instruments. Saatavissa: https://www.omega.com/en-us/resources/measurement-of-current-with-a-voltage-daq. Hakupäivä 22.4.2019.
6. Mitä on tiedonkeruu? National Instruments. Saatavissa: http://finland.ni.com/tiedonkeruu/mita-on. Hakupäivä 22.4.2019.
7. CompactDAQ Chassis. National Instruments. Saatavissa: http://www.ni.com/fi-fi/shop/select/compactdaq-chassis. Hakupäivä 22.4.2019.
8. NI-9238, C Series Voltage Input Module. National Instruments. Saatavissa: http://www.ni.com/fi-fi/support/model.ni-9238.html. Hakupäivä 22.4.2019.
9. NI-DAQmx. National Instruments. Saatavissa: http://www.ni.com/fi-fi/support/downloads/drivers/download.ni-daqmx.html#301173. Hakupäivä 22.4.2019.
10. Answers to Frequently Asked Questions about NI-DAQmx and Traditional NI-DAQ (Legacy). 2018. National Instruments. Saatavissa: http://www.ni.com/product-documentation/3021/en. Hakupäivä 22.4.2019.
11. A Python API for interacting with NI-DAQmx. 2017. National Instruments. Saatavissa: https://github.com/ni/nidaqmx-python. Hakupäivä 22.4.2019.
12. National Instruments .NET Support. 2019. National Instruments. Saatavissa: http://www.ni.com/product-documentation/14434/en. Hakupäivä 22.4.2019.
13. Stackify 2017. What are Windows Services? How Windows Services Work, Examples, Tutorials and More. Saatavissa: https://stackify.com/what-are-windows-services. Hakupäivä 27.4.2019.
14. manigandham 2014. OWIN, Open Web Interface for .NET. Saatavissa: http://owin.org. Hakupäivä 28.4.2019

15. Howard Dierking 2013. An Overview of Project Katana. Microsoft Docs. Saatavissa: https://docs.microsoft.com/en-us/aspnet/aspnet/overview/owin-and-katana/an-overview-of-project-katana. Hakupäivä 28.4.2019.
16. nschonni, gewarren, scottaddie, wadepickett, pgoldman5699, v-anpasi, tdykstra, Rick-Anderson & janmarques 2013. Use OWIN to Self-Host ASP.NET Web API. Microsoft Docs. Saatavissa: https://docs.microsoft.com/en-us/aspnet/web-api/overview/hosting-aspnet-web-api/use-owin-to-self-host-web-api. Hakupäivä 28.4.2019.
17. Ryan Nowak, Steve Smith & Rick Anderson 2019. Routing in ASP.NET Core. Microsoft Docs. Saatavissa: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-2.2. Hakupäivä 29.4.2019.
18. Mike Wasson 2014. Attribute Routing in ASP.NET Web API 2. Microsoft Docs. Saatavissa: https://docs.microsoft.com/en-us/aspnet/web-api/overview/web-api-routing-and-actions/attribute-routing-in-web-api-2. Hakupäivä 29.4.2019.
19. Design Patterns. Saatavissa: https://www.oodesign.com. Hakupäivä 22.4.2019.
20. mairaw, tompratt-AQ, guardrex, BillWagner, rpetrusha, nemrism, mjhoffman65, mikeblome, svick 2015. static (C# Reference). Microsoft Docs. Saatavissa: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static. Hakupäivä 10.5.2019.
21. About SQLite. sqlite.org. Saatavissa: https://www.sqlite.org/about.html. Hakupäivä 7.5.2019.