TAMPEREEN TEKNILLINEN YLIOPISTO

**Pate Blomqvist**

# LAAJENNETTAVAN LÄÄKETIETEELLISEN MITTALAITESIMULAATTORIN TOTEUTUS

Diplomityö

Tarkastaja: Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston
kokouksessa 05.10.2011

# TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
**BLOMQVIST, PATE**: Laajennettavan lääketieteellisen mittalaitesimulaattorin toteutus
Diplomityö, 58 sivua
Tammikuu 2012
Pääaine: Ohjelmistotuotanto
Tarkastajat: Professori Tommi Mikkonen
Avainsanat: fysiologinen, lääketieteellinen, mittaus, protokolla, simulaattori, laajennettavuus, arkkitehtuuri

Terveyskeskuksissa ja sairaaloissa on jo pitkään käytetty hoitotyön tukena fysiologisia mittauksia tekeviä potilasvalvontamonitoreja. Monitorit sisältävät nykyään suuren määrän ominaisuuksia, jotka on usein toteutettu modulaarisesti siten, että kukin moduuli pitää sisällään joukon fysiologisia mittalaitteita. Aikaisemmin moduulit on kytketty monitoreihin erilaisilla protokollilla, jotka ovat yleisesti olleet yhteensopimattomia keskenään.

Tilanteen korjaamiseksi monitorien valmistaja on kehittänyt uuden fysiologisen protokollan mittausten välittämiseen. Uusi protokolla hyödyntää XML- ja binääritietovirtoja mittausten viemiseen perille tehokkaasti, mutta silti riittävän yksityiskohtaisuuden säilyttäen. Fysiologiset mittaukset ovat luonteeltaan arkaluontoisia, ja virheet mittaustulosten välittämisessä ovat pahimmillaan hengenvaarallisia. Tämän vuoksi uuden protokollan testaamiseen kehitetään testausympäristö, missä fysiologisen datan lähteenä toimii tässä diplomityössä kehitettävä simulaattori.

Kehitettävän simulaattorin on oltava helposti laajennettava, ja sen täytyy tukea erilaisia fysiologisia parametreja erilaisista käskylähteistä tulevilla komennoilla. Varsinainen protokollan toiminnallisuus tulee valmistajan kehittämällä protokollakirjastolta, jonka lisäksi simulaattoriin toteutetaan toiminnallisuus käyttäjältä tulevien käskyjen käsittelyyn ja parsintaan. Toteutusvaiheessa simulaattorin tulee tukea mittausdatan lähettämistä normaalin lähiverkkoyhteyden yli, ja samoin simulaattorin käskytys testausympäristössä on tarkoitus tehdä lähiverkon kautta tulevien käskyjen avulla.

Toteutuksessa käytettiin ketteriä menetelmiä, ja lopputuloksena syntynyt simulaattori on käytössä monitoriin tulevan protokollapinon toteutuksessa. Simulaattorista havaittiin olevan korvaamatonta apua kehitystyössä, sillä tässä vaiheessa protokollaa käyttäviä mittalaitemoduuleja ei ole vielä olemassa. Simulaattorin arkkitehtuurista tuli vaadituilta osiltaan helposti laajennettava, ja simulaattori tulee siirtymään testausympäristön osaksi, kunhan monitorin protokollatoteutus etenee testausvaiheeseen. Valmistaja on kokonaisuudessaan tyytyväinen projektin tuloksiin ja on esittänyt myös toiveen simulaattorin käyttämisestä protokollakirjaston käyttöesimerkkinä.

# ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
**BLOMQVIST, PATE**: Implementing an extendable medical instrument simulator
Master of Science Thesis, 58 pages
January 2012
Major: Software Engineering
Examiner: Professor Tommi Mikkonen
Keywords: physiological, medical, measurement, protocol, simulator, expandability, architecture

For a while now, hospitals and medical centers have utilized patient monitors as a supportive tool for clinical work. Today's monitors have many features, many of which are implemented modularly as multiple physiological measurement instruments embedded into a single module. Currently the modules are usually connected to the monitor using a variety of protocols, usually incompatible with each other.

To overcome incompatibility issues, a manufacturer of monitors has developed a new protocol for passing physiological measurements from module to monitor. The new protocol uses XML and binary data flows to present data efficiently, but still preserving enough detail in the measurements. Physiological data is sensitive by nature, and any error in data delivery has a potential to be fatal. To minimize the risk for errors, a comprehensive testing environment is required before the protocol can be utilized.

In this thesis, a protocol simulator is developed for testing and verifying the new protocol. The simulator must be easily expandable, and it must support multitude of physiological measurements and take commands from several command sources. Actual implementation for the new protocol is received from a protocol library being developed by the manufacturer. In addition to this, functionality for receiving and parsing commands from the user was built into the simulator. The simulator must support measurement data output over normal local area network (LAN), and the test environment must be able to send commands to the simulator over LAN.

The simulator was implemented using agile methods, and the finished simulator is currently used in the development of the protocol stack for the monitor. The simulator was found to be invaluable help when developing the monitor side, as there are currently no actual measurement modules capable of outputting the new protocol. Architecture of the simulator is easily expandable from the required areas, and the simulator will be part of the testing environment once the monitor side protocol stack enters testing. Manufacturer was pleased with the results, and has even shown interest in using the simulator as a usage example for the protocol library.

# ALKUSANAT

Tämä työ on tehty Tampereen teknillisen yliopiston Ohjelmistotekniikan laitokselle. Työssä esitettävän simulaattorin toteutus aloitettiin kesäkuussa 2011 ja varsinainen kirjoitus syyskuussa 2011. Lopullinen viimeistely diplomityön kirjalliseen asuun tehtiin joulukuussa 2011.

Työ tehtiin osana asiakasprojektia Vincit Oy:ssä, joka on Tampereen Hervannassa toimiva, noin 60 henkeä työllistävä ohjelmistotalo. Vincit on perustettu vuonna 2007, ja itse olen ollut yrityksen palveluksessa vuodesta 2009 lähtien. Yrityksen puolelta diplomityön etenemistä ohjasi DI Pekka Virtanen, ja yliopiston puolella työn tarkastajana ja ohjaajana toimi professori Tommi Mikkonen. Lämpimät kiitokset kummallekin, minkä lisäksi haluan kiittää tässä nimettömäksi jäävää asiakkaan edustajaa työn sisällön tarkastamisesta.

Suureksi avuksi diplomityön kirjoittamisessa ovat olleet Petteri Laukkanen, Leena Blomqvist, Olli-Pekka Valtonen sekä puolisoni Maija Pihlaja, joiden kaikkien antamat kommentit tutkielman sisällöstä ja sen ulkoasusta ovat auttaneet merkittävästi. Lisäksi erikoiskiitos Maijalle, joka on jaksanut kuunnella ainaista ”tänään pitää vääntää dippaa” -ruljanssia kiitettävästi läpi koko syksyn. :)

Ilman teitä työ ei olisi siinä pisteessä, missä se tällä hetkellä on. Kiitos!

---

Pate Blomqvist
Tampereella 12.12.2011

# SISÄLLYS

1. Johdanto . . . . . 1
2. Protokolla fysiologisiin mittauksiin . . . . . 3
2.1 Fysiologiset mittaukset ja tulosten esittäminen . . . . . 3
2.2 Protokollan kuvaus . . . . . 4
2.2.1 Yleiskuvaus . . . . . 4
2.2.2 Istunto ja datayhteydet . . . . . 4
2.2.3 Datasäiliöt ja mittausten lähettäminen . . . . . 6
2.2.4 XML ja XML-binary . . . . . 7
2.3 Testausympäristö . . . . . 8
2.3.1 Simulaattori . . . . . 9
2.3.2 Monitori . . . . . 9
2.3.3 RobotFramework . . . . . 10
2.3.4 Protokollamuunnin . . . . . 10
2.4 Yhteenveto . . . . . 10
3. Protokollakirjasto . . . . . 12
3.1 Yleiskuvaus . . . . . 12
3.2 Core-kerros . . . . . 13
3.3 Transport-kerros . . . . . 14
3.3.1 IpServer . . . . . 15
3.3.2 IpConnection . . . . . 16
3.4 Marshalling-kerros . . . . . 16
3.4.1 Dataluokat . . . . . 17
3.4.2 Binärityypit . . . . . 18
3.5 Interaction-kerros . . . . . 18
3.5.1 SessionManagerDevice . . . . . 20
3.5.2 DeviceConnectionHandler . . . . . 21
3.6 Rakenteellisia yksityiskohtia . . . . . 21
3.6.1 Boost . . . . . 21
3.6.2 Enumeraatioiden kapselointi . . . . . 22
4. Simulaattorin suunnitteluperusteet . . . . . 24
4.1 Vaatimukset . . . . . 24
4.1.1 Toimintaympäristö . . . . . 24
4.1.2 Toiminnalliset vaatimukset . . . . . 25
4.1.3 Reunaehdot . . . . . 25
4.1.4 Käyttöoletukset . . . . . 26
4.2 Suunnitteluperiaatteet ja käytännöt . . . . . 26
4.3 Ketterät menetelmät . . . . . 27

5. Simulaattorin toteutus . . . . . 28
5.1 Tilamalli . . . . . 28
5.2 Yleisarkkitehtuuri . . . . . 29
5.3 Käskysilmukka . . . . . 30
5.3.1 Toiminta . . . . . 30
5.3.2 Käskylähteet . . . . . 32
5.3.3 Käskykäsittelijät . . . . . 33
5.3.4 Laajentaminen . . . . . 34
5.4 Parametrien käsittely . . . . . 35
5.4.1 Toiminta . . . . . 35
5.4.2 Blokkien alustus . . . . . 36
5.4.3 Mittausarvojen asettaminen . . . . . 38
5.4.4 Laajentaminen . . . . . 39
5.5 Kirjastorajapinta . . . . . 42
5.5.1 Toiminta . . . . . 42
5.5.2 Ajastuksen hallinta . . . . . 44
5.5.3 Laajentaminen . . . . . 47
5.6 Aputoiminnot . . . . . 47
6. Valmiin simulaattorin arviointi . . . . . 49
6.1 Projektin onnistuminen . . . . . 49
6.2 Prosessin arviointi . . . . . 50
6.3 Vaatimusten toteutuminen . . . . . 51
6.3.1 Toimintaympäristö . . . . . 51
6.3.2 Toiminnalliset vaatimukset . . . . . 51
6.3.3 Reunaehdot . . . . . 52
6.3.4 Käyttöoletukset . . . . . 53
6.4 Jatkokehitys . . . . . 54
7. Päätelmät . . . . . 55
Lähteet . . . . . 57

# 1. JOHDANTO

Terveyskeskuksissa ja sairaaloissa on jo pitkään käytetty potilasvalvontamonitoreja hoitotyön tukena. Monitorit pitävät nykyisin sisällään suuren määrän ominaisuuksia, jolloin niiden on myös tuettava suurta joukkoa erilaisia fysiologisia mittalaitteita. Fysiologisilla mittauksilla tarkoitetaan elimistön toimintaa seuraavia ja toimintojen muutoksista raportoivia mittauksia. Usein monitorit ovat modulaarisia siten, että kukin moduuli pitää sisällään joukon mittalaitteita erilaisille fysiologisille parametreille. Tällaisia parametreja ovat esimerkiksi elektrokardiografia (EKG), verenpainemittaukset sekä erilaisiin hengityskaasuihin liittyvät mittaukset.

Aikaisemmin moduulit on kytketty valmistajan monitoreihin käyttäen lukuisia toisistaan poikkeavia protokollia. Tästä seuraa suuri määrä protokollaspesifistä koodia monitorissa, jonka lisäksi myös liittimet ja muu laitteisto joudutaan suunnittelemaan jossain määrin protokollan ehdoilla. Ongelman poistamiseksi valmistaja on kehittänyt moduulin ja monitorin väliseen kommunikaatioon uuden yleiskäyttöisen protokollan, jonka on tarkoitus korvata aiemmin käytössä olleet laitteistoriippuvaiset ratkaisut. Uusi kehitetty protokolla mahdollistaa yleisesti tunnettujen fysiologisten mittaustulosten lähettämisen esimerkiksi normaalin lähiverkon yli. Protokolla käyttää mittausten kuvaamisen ilmaisuvoimaista XML-kieltä (*Extensible Markup Language*), jonka jälkeen varsinainen mittausdata lähetetään kuvaukseen perustuvassa binäärimuodossa vastaanottajalle. Näin saavutetaan riittävä kapasiteetti sekä mittausten kuvaamiselle että mittausdatan lähettämiselle rajallisen siirtokaistan yli.

Fysiologiset mittaukset ovat luonteeltaan arkaluontoisia, ja virheet mittaustulosten välittämisessä ovat pahimmillaan hengenvaarallisia. Tästä johtuen kaikki mittaukseen osallistuvat laitteet ja protokollat tulee testata ja varmentaa huolellisesti ennen kuin ne voidaan hyväksyä kliinisen työn apuvälineiksi. Protokollan kehitysvaiheessa varsinaisia mittausmoduuleita ei ole vielä käytössä, mistä johtuen uuden protokollan testausympäristön keskeiseksi komponentiksi kehitetään tässä diplomityössä protokollaa tukeva simulaattori. Simulaattori rakennetaan käyttäen valmistajan kehitteillä olevaa protokollakirjastoa, joka toteuttaa uuden protokollan keskeiset rakenteet C++-kielellä. Protokollakirjasto tulee käyttöön myös varsinaisen potilasvalvontamonitorin protokollapinossa, ja simulaattorin toteutus tukee samalla protokollakirjaston kehitystä.

Simulaattorin tarvitsee tukea aluksi vain suppeaa osaa kaikista fysiologisista mittauksista, mutta simulaattorin toteutuksessa on otettava huomioon sen laajennettavuus tule-

vaisuudessa kattamaan yhä suuremman joukon protokollakirjaston tukemia mittauksia. Simulaattorin keskeinen suunnitteluperuste on siis laajennuspisteiden tekeminen mahdollisuuksien mukaan yksinkertaisiksi siten, että ne eivät rajoita liikaa tulevien laajennusosien toimintaa. Laajennuspisteiden ympäristöä lukuun ottamatta simulaattorin toteutuksessa pyritään mahdollisimman yksinkertaisiin ratkaisuihin, sillä yksinkertainen toteutus on helpompi varmentaa oikein toimivaksi. Simulaattorin toteutuksessa tullaan käyttämään ketteriä ohjelmistomenetelmiä, jonka seurauksena välidokumenttien määrä pienenee ja huomio voidaan keskittää aina kulloinkin eniten huomiota kaipaavaan osa-alueeseen.

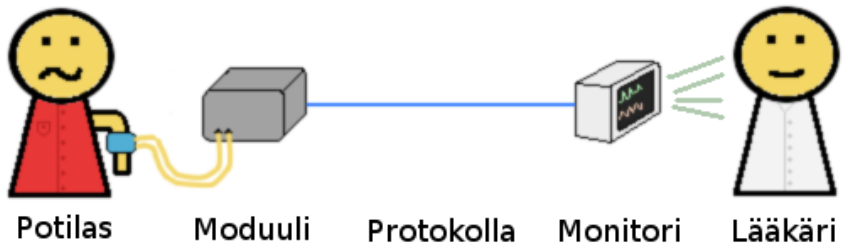
Luvussa 2 käydään läpi yleisesti fysiologisiin mittauksiin liittyviä rajoitteita, minkä jälkeen käsitellään valmistajan uusi protokolla sekä protokollan testaukseen kehitetty testausympäristö. Luvussa 3 kuvataan uuden protokollan toteuttava protokollakirjasto, ja siinä keskitytään protokollakirjaston yleiskuvaukseen, sen eri kerroksiin sekä toteutettavan simulaattorin kannalta oleellisiin rakenteellisiin yksityiskohtiin. Luvussa 4 käsitellään monitorin valmistajan simulaattorille asettamat vaatimukset sekä käydään läpi niitä suunnitteluperiaatteita ja käytäntöjä, joita simulaattorin toteutuksessa noudatettiin. Luvussa 5 käydään läpi simulaattorin tilamalli ja yleisarkkitehtuuri, minkä jälkeen paneudutaan käskysilmukan, parametrien käsittelyn sekä protokollakirjastorajapinnan yksityiskohtiin. Luvussa 6 pohditaan simulaattorin sekä sen toteutusprosessin onnistumista, verrataan toteutuneita ominaisuuksia aiemmin esitettyihin vaatimuksiin ja pohditaan simulaattorin jatkokehitystä. Luvussa 7 esitetään yleiset tulokset ja tehdään päätelmiä simulaattorin toteutuksesta ja tulevaisuudesta.

# 2. PROTOKOLLA FYSIOLOGISIIN MITTAUKSIIN

Kehitteillä olevan protokollan kuvaukseen lähdetään kommentoimalla fysiologisten mittausten luonnetta sekä jo olemassa olevia protokollia. Tämän jälkeen kuvataan kehitteillä oleva protokolla pääpiirteissään, minkä jälkeen käydään läpi protokollan testaamiseen rakennettava automaatiotestiympäristö.

## 2.1 Fysiologiset mittaukset ja tulosten esittäminen

Fysiologisten mittausten tekemiseen ja esittämiseen on olemassa eri valmistajilta lukuisia ratkaisuja. Usein näissä käytetään useita monitorityyppejä, joihin voidaan kytkeä suuri määrä erilaisia moduuleja. Eri monitorien ja moduulien väliseen kommunikointiin on täytynyt sopia fysiologisen mittausdatan välittämiseen soveltuvia protokollia, joista monet ovat kunkin valmistajan omistamia ja tarkasti suojelemia. Tyypillinen fysiologisen datan mittaustilanne on esitetty kuvassa 2.1.




Kuva 2.1: Tyypillinen mittaustilanne. Moduuli ja monitori ovat yleensä samassa yksikössä.

Aikaisemmin protokollat saattoivat olla hyvin laitteistoriippuvaisia, mutta tilanne on muuttunut merkittävästi. Nykyään sulautetuissa järjestelmissä käytetään usein esimerkiksi johonkin Linux-jakeluun pohjautuvia laitteistoja, jolloin käytössä on joukko valmiita verkkoprotokollia. Valmiit verkkoprotokollat mahdollistavat kommunikaation käyttäen olemassa olevia liittimiä, mikä yksinkertaistaa laitteita ja tekee niistä edullisempia valmistaa. Valmiit protokollat mahdollistavat datan siirron myös langattomasti, mutta tässä yhteydessä on pidettävä mielessä mitattavan datan arkaluontoisuus sekä datavirtojen sekaantumisesta aiheutuvat riskit. Yksi erityinen huolenaihe langattomissa järjestelmissä on käytössä olevien taajuuskaistojen ruuhkautuminen, jonka on jo nyt havaittu aiheuttavan ongelmia muun muassa lääketieteellisissä laitteissa [1].

Uuden protokollan toteutuksessa on datan luonteesta johtuen päästävä erittäin pitkälle

vietyyn turvallisuuteen ja toimintavarmuuteen. Tätä varten protokollan toteuttavat kirjastot sekä protokollan varsinaiset käyttökohteet on testattava huolellisesti. Varsinaisia mittauksia tekevät moduulit ovat usein kalliita, joten testauksessa yksi vaihtoehto on rakentaa simulaattori korvaamaan tämä osuus ketjusta. Simulaattoria on usein myös helpompi ohjata ohjelmallisesti, jolloin myös varsinaisten testitapausten kirjoittaminen on helpompaa.

## 2.2 Protokollan kuvaus

Tässä kohdassa kuvataan kehitteillä olevan protokollan rakenne. Yleiskuvauksen jälkeen esitellään protokollan istunnot sekä datayhteydet. Tämän jälkeen siirrytään alaspäin datan esitystasolle sekä käydään läpi XML- ja XML-binary -esitystapojen käyttö protokollassa.

### 2.2.1 Yleiskuvaus

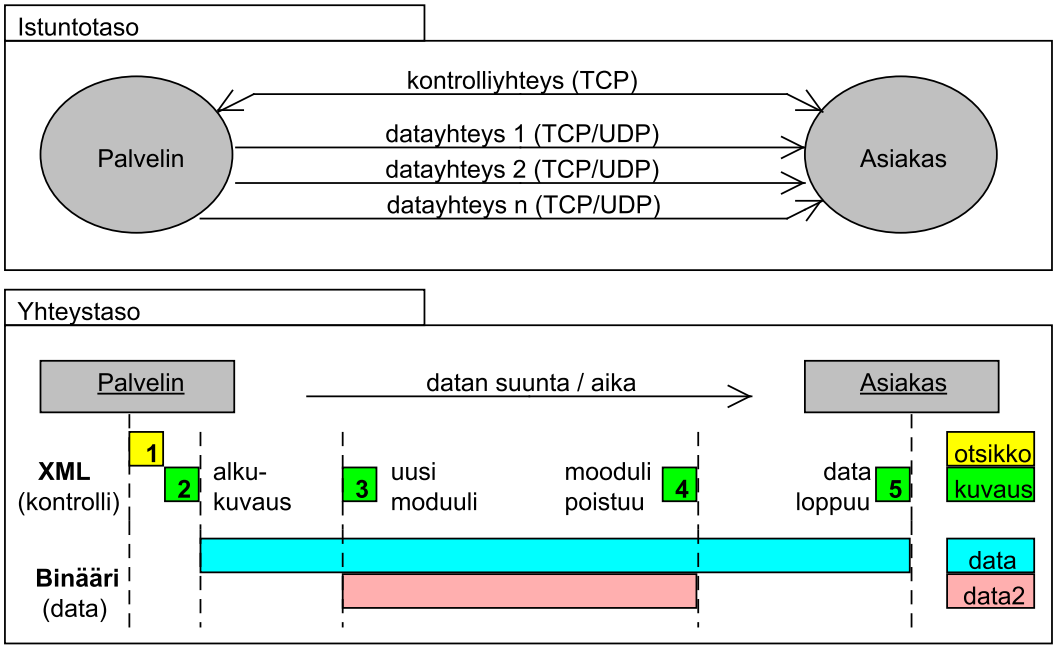
Kehitteillä oleva protokolla on tarkoitettu uuden sukupolven siirtoprotokollaksi reaaliaikaisen fysiologisen mittausdatan välittämiseen sairaaloissa ja muissa hoitolaitoksissa. Protokolla tukee myös erilaisten hälytysten ja tapahtumien välittämistä laitteelta toiselle, ja sen on tarkoitus korvata valmistajan nykyään käyttämät, keskenään epäyhteensopivat protokollat yhdellä toimivalla ratkaisulla. Kirjoitushetkellä protokollaa ei ole julkaistu, ja sen kehitys on vielä osittain työn alla.

Protokolla on suunniteltu skaalautumaan helposti pienistä mukana kannettavista sensoreista aina yleiskäyttöisiin monitorointilaitteisiin asti. Kukin laite voi päivittää dynaamisesti XML-kuvaustaan, jolloin voidaan helposti tukea kaikkia niitä laitekonfiguraatioita, joita esimerkiksi sairaaloissa tarvitaan. Protokolla mahdollistaa kaiken datan lähettämisen ulospäin erilaisina tietovirtoina, joista kuuntelijan on helppo suodattaa ja hakea haluamansa tieto tehokkaasti.

Tietosuoja on tärkeä osa käytännöllisesti katsoen kaikissa mahdollisissa käyttötapauksissa. Protokolla on suunniteltu rakentumaan *Transmission Control Protocol* (TCP,[2])- ja *User Datagram Protocol* (UDP,[3])-protokollien päälle, minkä lisäksi lähetetty tieto voidaan vielä erikseen salata. Tämä mahdollistaa protokollan käyttämisen myös monissa nopeasti yleistyvissä käyttötapauksissa kuten kotihoidossa sekä liikkuvissa hoitoyksiköissä.

### 2.2.2 Istunto ja datayhteydet

Istunto on *asiakkaan* ja *palvelimen* välinen käsite, johon kuuluu joukko erilaisia data- ja kontrolliyhteyksiä (kuva 2.2). Mikä tahansa mittausdataa tuottava laite (moduuli/simulaattori) toimii protokollassa *palvelimena*, joka lähettää mittausdataa siihen yhteydessä oleviin asiakkaisiin (monitori). Istunto luodaan yhden potilaan mittausdatan lähettämiseen palvelimelta asiakkaalle.




Kuva 2.2: Istunto ja datayhteydet

Istuntotasolla protokollaan kuuluu yksi *kontrolliyhteys*, joka toimii istunnon luomisen aikana tehtävän kättelyn viestien välittäjänä. Kontrolliyhteyden kautta tehdään myös pyynnöt uusien *datayhteyksien* käynnistämiseksi sekä varsinaisten tietovirtojen aloittamiseksi. Datayhteyksiä voi olla yhtä istuntoa kohden useita, ja ne voidaan usein luoda varmistamatonta kuljetuskerrosta käyttäen. Useat datayhteydet mahdollistavat esimerkiksi eri näytteenottotaajuuksilla tehtyjen mittausten lähettämisen eri kanavia pitkin. Kontrolliyhteys on koko istunnon kannalta keskeinen yhteys, joten se on syytä muodostaa varmistetun kuljetuskerroksen päälle. Yhteydet voidaan rakentaa esimerkiksi TCP- ja UDP-protokollien päälle, jolloin TCP on varmistettu ja UDP varmistamaton yhteys.

Kontrolliyhteys toimii edellä kuvatun lisäksi varmistetun luonteensa takia XML-kuvausten lähettämisyhteytenä. Datayhteyksissä lähetetään varsinainen mittausdata, joka on binäärikoodattu kontrolliyhteyden yli lähetettyjen XML-kuvausten mukaisesti. Istunnon kulku yhden datayhteyden suhteen on listattu alla.

1. **Otsikko:** Viesti kertoo asiakkaalle joukon koko datayhteyden ajan samana pysyviä tietoja. Tällaisia ovat esimerkiksi aloitusajankohta ja istunnon tunniste.

2. **Kuvaus:** Ensimmäinen kuvausviesti kertoo, missä formaatissa dataa tullaan lähettämään datayhteyden yli. Kuvaukseen kuuluu lähetettävien parametrien tiedot, parametreihin kuuluvien mittausten yksiköt sekä binäärikoodauksessa käytettävät yksiköt. Kun asiakas on vastaanottanut viestin, se pyytää kontrollikanavaa käyttäen palvelinta aloittamaan binääridatavirran lähetyksen.

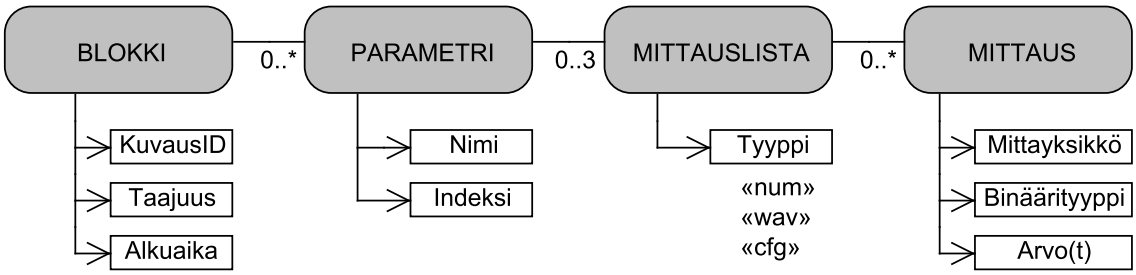
3. **Kuvaus:** Palvelimeen kytkettiin uusi moduuli, jolloin palvelin lähettää päivitetyn kuvausviestin asiakkaalle. Päivitetyssä kuvauksessa uuden moduulin parametrit on lisätty olemassa olevaan kuvaukseen. Tässä vaiheessa aloitetaan uuden moduulin datan lähettäminen saman binäärivirran yli.

4. **Kuvaus:** Jälkikäteen kytketty moduuli poistetaan palvelimesta, jolloin palvelin lähettää ensimmäistä kuvausta vastaavan XML-kuvauksen asiakkaalle. Palvelin jatkaa pelkän alkuperäisen binääridatavirran lähettämistä.

5. **Kuvaus:** Palvelin on sammuttamassa istuntoa, jolloin se lähettää asiakkaalle tiedon yhteyden sammuttamisesta ja sulkee kontrolliyhteyden. Kontrolliyhteyden sulkeutuessa molemmat päät sulkevat kaikki yhteyteen liitetyt datayhteydet, ja istunto suljetaan.

### 2.2.3 Datasäiliöt ja mittausten lähettäminen

Kehitteillä oleva protokolla tarjoaa käyttäjälle tietyt rakenteet datan jäsentämiseksi. Varsinainen datan sijoittelu ja yhteyksiin jakaminen on jätetty käyttäjän tehtäväksi. Protokollan käsitteelliset datasäiliöt on esitetty kuvassa 2.3.




Kuva 2.3: Kuvaus protokollan datasäiliöistä

Korkeimman tason yksikkö protokollan datan siirrossa on *blokki*. Blokki kuvaa jotain fysiologisten parametrien kokonaisuutta, jonka käyttäjä on määrittänyt lähetettäväksi yhdellä kertaa. Kullakin blokilla on sille käsin määritetty *KuvausID*, joka lähetetään kuvausviestin sekä jokaisen binääriviestin yhteydessä asiakkaalle. *KuvausID* määrää kuuntelijalle istuntokohtaisesti, mihin blokkiin kulloinkin vastaanotettu data liittyy. Blokilla on lisäksi tallessa lähetyksen *aloitusaika* sekä *lähetystaajuus*, jotka kerrotaan asiakkaalle kuvausviestillä ennen datavirran aloittamista. Binääriviestiin lisätään laskuri, joka kertoo kuinka mones blokki kulloinkin on lähetetty. Näillä tiedoilla asiakkaan on mahdollista laskea blokille todellinen aikaikkuna siinäkin tapauksessa, että jokin välillä lähetetyistä binääriviesteistä jää saapumatta.

Käyttäjä saa sijoittaa kuhunkin blokkiin haluamansa määrän *parametreja*, jotka kuvaavat jotakin protokollan määrittämää fysiologista mittauskokonaisuutta. Esimerkiksi elektrokardiografiaan (EKG) liittyvät mittaukset on yhdistetty yhden parametrin alle. Parametrilla on *nimen* lisäksi *indeksi*, joka mahdollistaa saman parametrin lähettämisen useaan kertaan saman blokin sisälle. Parametrien sisältämät mittaukset varmennetaan parametrin nimeä vasten käyttämällä XML Schemaa [4].

Parametrin sisällä mittaukset jaetaan *mittauslistoihin*, jolloin kukin parametri voi sisältää lukuisia *mittauksia*. Olennaista tässä on erotella esimerkiksi numeroarvot sekä erilaiset aaltokäyrät omiin blokkeihinsa, sillä niiden päivitystaajuudet poikkeavat huomattavasti toisistaan. Erilaisia mittauslistoja on kolmea päätyyppiä:

- **Num:** Numeromittaukset. Nämä ovat esimerkiksi sekunnin välein päivittyviä mittaustuloksia, joita tyypillisesti näytetään käyttäjälle puhtaasti numeromuodossa.
- **Wav:** Aaltokäyrät. Nämä päivittyvät lukuisia kertoja sekunnissa ja tyypillisesti ne esitetään graafisena käyränä. Aaltokäyrien tapauksessa yhdessä binääriviestissä lähetetään aina koko jakson mittaustulokset kerralla.
- **Cfg:** Mittausasetukset. Näiden avulla esitetään joitakin sellaisia arvoja, joita ei saa sijoitettua numeromittauksiin järkevästi. Tyypillisesti päivitystaajuus on huomattavasti numeromittauksia pienempi.

Varsinainen mittaustieto talletetaan mittaus kerrallaan mittauslistoihin. Kullekin mittaukselle määritetään UCUM-yksiköissä määritetty *mittayksikkö* [5]. Tämän lisäksi kullekin mittaukselle määritellään *binäärityyppi*, johon sisältyy tieto binääriesityksen pituudesta. Tietoa hyödyntäen binääriviestin hyötykuorma voidaan rakentaa niin, että vastaanottaja pystyy tulkitsemaan sen kuvausviestin perusteella. Binärityypeissä käytetään XML Schemaan määriteltyjä tietotyyppejä [6]. Mittaukseen liittyy lisäksi vielä *varsinaiset mittaustulokset*. Nämä lähetetään asiakkaalle binäärikoodattuina oikeaa binärityyppiä käyttäen. Sekä numeromittauksissa että mittausasetuksissa arvoja on mittausta kohden yksi. Aaltokäyrämittauksissa yhdessä binääriviestissä lähetetään riittävästi mittaustuloksia täyttämään kahden blokin välinen aikaväli.

### 2.2.4 XML ja XML-binary

Saavuttaakseen sille asetetut tehokkuusvaatimukset protokolla hyödyntää mittausdatan siirrossa XML-formaattia sekä erikseen tähän tarkoitukseen määritettyä XML-binary-formaattia[1]. XML-formaatti tarjoaa hyvät laajentamismahdollisuudet, jonka lisäksi esitystavan apuna on suuri joukko sekä suljetun että avoimen lähdekoodin kehitystyökaluja. Lisäksi XML-formaatissa oleva data voidaan varmentaa tehokkaasti käyttäen XML

[1] Eri asia kuin W3C Working Groupin määrittämä samanniminen formaatti [7].

Schemaa [4]. XML-formaattia käytetään protokollassa pääasiassa vähän kaistanleveyttä vaativiin, epäsäännöllisesti lähetettäviin viesteihin. Tällaisia viestejä ovat esimerkiksi potilastietoja sisältävät viestit, yhteyden muodostuessa tehtävä kättely sekä lähetettävien blokkien kuvaukset.

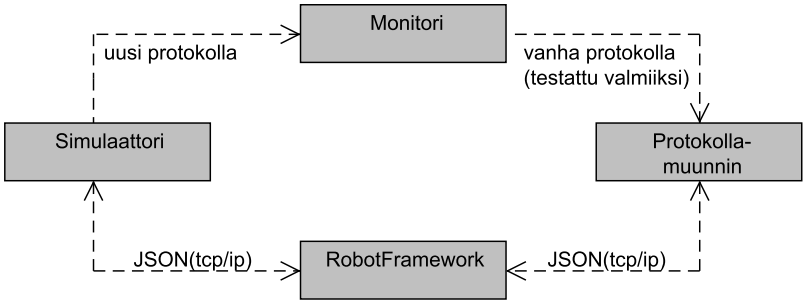
XML-binary-formaattia käytetään pääasiassa nopeasti muuttuvan aaltokäyrä- ja numeroarvodatan välittämiseen. Binäärinen esitysmuoto on oleellinen sovelluksissa, joissa hyötykuorman suuruus ja siirtokaistan tehokas käyttö ovat tärkeitä. Tästä on hyötyä varsinkin akkukäyttöisissä laitteissa tai tiedonsiirtokapasiteetin ollessa rajoitettu.

Binääriesityksessä hyödynnetään fysiologisen mittausdatan luonnetta, jonka mittaustuloksissa muuttuvat yleensä ainoastaan mittausten arvot mittayksiköiden ja mitattavien suureiden pysyessä samoina. Tällöin binääridatavirta on mahdollista tulkita, kunhan tulevan datan rakenne on yhteyttä muodostaessa välitetty jollakin tavalla. Datan rakenne kuvataan käyttämällä luotettavan yhteyden – esimerkiksi TCP – yli lähetettävää XML-viestiä, jossa määritellään yksiselitteisesti tulevan datavirran sisältämät mittaukset. Näistä tiedoista vastaanottaja pystyy rakentamaan alkuperäisen datan saamansa binääridatavirran perusteella.

Binäärisen esityksen etuna on lähetettävän datan pieni koko. Tekstimuodossa lähetetty data voi viedä kymmeniä kertoja enemmän tilaa kuin binääriesityksessä lähetetty. Koska kuvausviestien formaatti on normaali XML, tiedot esimerkiksi mittayksiköistä voidaan tallentaa suurella tarkkuudella tämän vaikuttamatta kaistan kulutukseen. Lisäksi menetelmä hyödyttää datayhteyden molempia päitä, koska binääridataa ei tarvitse muuttaa tekstimuotoon kummassakaan päässä. Tämä parantaa käsittelynopeutta varsinkin reaalilukujen tapauksessa moninkertaiseksi. [8]

## 2.3 Testausympäristö

Uuden protokollan ja protokollakirjaston testaamiseksi rakennetaan testausympäristö, joka koostuu keskenään verkkoyhteyksien yli keskustelevista ohjelmista. Testausympäristö on esitetty kuvassa 2.4.




Kuva 2.4: Yleiskuva testiympäristöstä

Testitapaukset kirjoitetaan *RobotFrameworkille*, joka ohjaa simulaattoria TCP-yhteyden yli lähettämällä sille komentoja *JSON*-formaatissa (*Java Script Object Notation*). *Simulaattori* vastaanottaa komennot ja lähettää ne edelleen monitorille myöhemmin kuvattavaa protokollakirjastoa käyttäen. *Monitori* hyödyntää samaa protokollakirjastoa datan vastaanottamiseen ja lähettää datan edelleen verkkoon käyttäen valmistajan aikaisempaa, jo testattua dataprotokollaa. *Protokollamuunnin* kuuntelee vanhaa protokollaverkkoa ja välittää datan takaisin RobotFrameworkin sitä kysyessä.

### 2.3.1 Simulaattori

Tässä diplomityössä rakennettu mittalaitesimulaattori rakennettiin toimimaan edellä kuvatun testausympäristön osana. Ennen testausympäristön kaikkien komponenttien valmistumista simulaattorin tehtävä on auttaa monitorin uuden verkkopinon toteuttamista, sillä oikea mittausmoduuli on kallis eikä sitä ole vielä tässä vaiheessa saatavilla.

Simulaattori ottaa TCP-yhteyden yli vastaan JSON-muotoisia komentoja RobotFrameworkilta ja muodostaa protokollakirjastoa hyödyntäen protokollayhteyden monitorin kanssa. Yhteyden saatuaan simulaattori lähettää monitorille tietyin väliajoin RobotFrameworkin määräämää mittausdataa. Simulaattori lähettää samaa dataa niin kauan, että se saa uudet mittausarvot RobotFrameworkilta. Simulaattori raportoi annettujen komentojen onnistumisesta RobotFrameworkille saman TCP-yhteyden yli, josta komennot tulivat.

### 2.3.2 Monitori

Monitori on testausketjun todellinen kohde, sillä koko testiympäristön rakentamisen tarkoituksena on mahdollistaa monitoriin toteutettavan protokollakerroksen testaaminen. Oleellisilta osiltaan monitori on Gentoo Linux-jakelun päälle pystytetty modulaarinen C++-ohjelma, joka käyttää laitteiston tarjoamia rajapintoja mittausdatan keräämiseen. Monitori lähettää myös mittausdataa eteenpäin valmistajan vanhaan tietoverkkoon, joka on käytännössä TCP/IP:n päälle rakennettu, usean monitorin mittausdatan keskitettyä seurantaa varten tehty protokolla.

Monitoriin toteutetaan uusi verkkopino käyttäen siinä valmiina olevia modulaarisia rajapintoja sekä uuden protokollan C++-protokollakirjastoa. Verkkopino tulee toimimaan TCP- ja UDP-yhteyksien päällä, joita luodaan ja ohjataan protokollakirjaston sisältä. Protokollakirjasto itsessään instantioidaan verkkomittauksiin erikoistuneen rajapinnan taakse, jolloin se saadaan yhdistettyä monitorin mittausdatan lähteitä valitsevaan komponenttiin. Kun toteutettu verkkopino valitaan lähteeksi, se luo uuden instanssin protokollakirjastosta ja jää odottamaan datan tuottajan (simulaattori) yhteydenottoa.

### 2.3.3 RobotFramework

RobotFramework on Nokia Siemens Networksin tukema vapaan lähdekoodin ympäristö, joka mahdollistaa automaatiotestauksen testitapausten kirjoittamisen avainsanoihin perustuvan syntaksin avulla. RobotFramework on kirjoitettu Pythonilla, ja siihen voidaan luoda uusia moduuleja jonkin tietyn toiminnon toteuttamiseksi. Lisämoduulit liitetään osaksi RobotFrameworkin tuntemaa avainsanastoa. Testitapausten kirjoittaminen on näiden avainsanojen yhdistämistä korkeamman tason avainsanoiksi, jolloin varsinaiset testitapaukset ovat jo lähellä luettavan tekstin tasoa. [9]

Kuvatussa testiympäristössä RobotFramework toimii automaatiotestauksen suorittajana, jonka toimintaa tukemaan kirjoitetaan Pythonilla ajurit sekä simulaattorille että protokollamuuntimelle. RobotFramework ohjeistaa simulaattoria ottamaan yhteyttä haluttuun monitoriin ja lähettämään tälle testitapauksen vaatimat mittaukset kirjoitettujen testitapausten perusteella. Tämän jälkeen RobotFramework pyytää protokollamuunninta hakemaan vastaavat mittaukset vanhasta protokollaverkosta ja vertailee saatua dataa lähetettyyn dataan monitorin toiminnan varmistamiseksi.

### 2.3.4 Protokollamuunnin

Protokollamuunnin on viimeinen testausympäristöön tehtävä komponentti, koska sillä ei ole hyötykäyttöä varsinaisen testausympäristön ulkopuolella. Muunnin odottaa RobotFrameworkilta TCP-yhteyden yli tulevia JSON-ohjauskomentoja ja asettuu komennon saatuaan kuuntelemaan valmistajan vanhaa protokollaverkkoa. Kun vanhasta verkosta tulee dataa, protokollamuunnin muuntaa datan RobotFrameworkille sopivaan muotoon ja lähettää sen edelleen eteenpäin.

## 2.4 Yhteenveto

Fysiologisten mittausten välittämiseen potilasvalvontamonitorille on olemassa lukuisia ratkaisuja eri valmistajilta. Käyttötapausten suuri määrä vaatii sen, että kommunikoinnissa käytetty protokolla tukee mahdollisimman suurta määrää erilaisia toisilleen vieraita moduuleja. Tämä ei ole onnistunut vanhoilla protokollilla kovin hyvin, ja siksi valmistaja on kehittämässä uutta dataprotokollaa mittausdatan välittämiseen.

Protokolla itsessään määrittää XML Sceman avulla datan esitysmuodon sekä yhteystyypit, mutta jättää toteutusyksityiskohdat kunkin sovelluksen ratkaistavaksi. Valmistaja on helpottanut protokollan käyttöönottoa kehittämällä C++-kielisen protokollakirjaston, joka piilottaa sovelluksen toteuttajalta suuren osan protokollan monimutkaisuudesta.

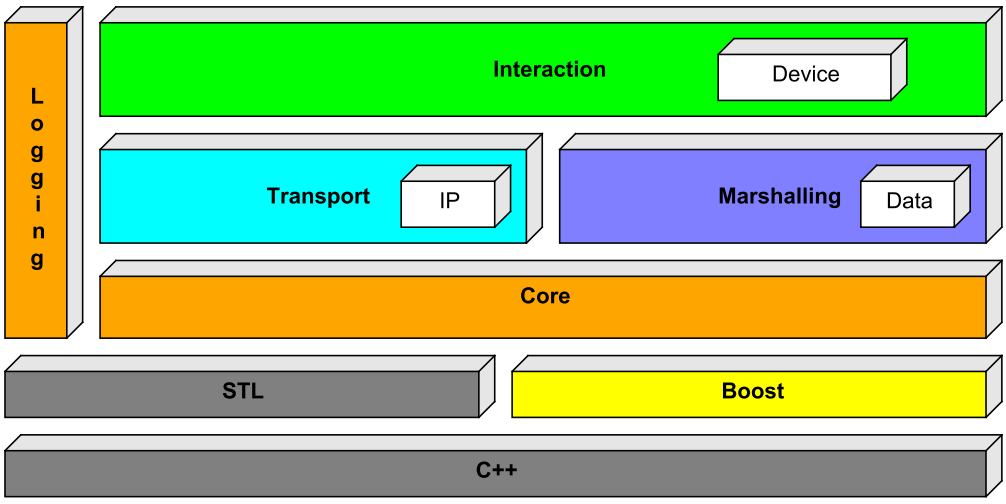
Fysiologisen datan luonteesta johtuen kaikki sovellukset on testattava huolellisesti. Varsinaiset mittalaitteet ovat kuitenkin kalliita ja lisäksi olisi suotavaa, että monitorisovellusta voitaisiin kehittää yhtä aikaa moduulisovelluksen kanssa. Testauksen helpottamiseksi lähdetään rakentamaan protokollasimulaattoria valmistajan tarjoaman protokollakirjaston pohjalta. Testauksen kohteeksi joutuu siis toteutettavan sovelluksen lisäksi myös protokollakirjasto.

# 3. PROTOKOLLAKIRJASTO

Simulaattoria varten ei lähdetty rakentamaan protokollatoteutusta itse, vaan tehtävässä käytettiin valmistajan kehitteillä olevaa protokollakirjastoa. Seuraavassa on esitetty protokollakirjaston kerrosarkkitehtuurin yleiskuvaus, jonka jälkeen käydään läpi kunkin kerroksen tärkeimmät velvollisuudet. Lopuksi käsitellään joitakin simulaattorin toteutuksen kannalta oleellisia rakenteellisia yksityiskohtia. Huomattavaa on, että tässä ei ole tarkoitus antaa lukijalle täydellistä kuvaa protokollakirjaston toiminnasta vaan keskittyä simulaattorin rakenteen kannalta olennaisiin yksityiskohtiin.

## 3.1 Yleiskuvaus

Kirjasto rakentuu kerrosarkkitehtuurin mukaisesti C++-ohjelmointikielen peruskirjastojen päälle siten, että toteutusta eri kerrosten sisällä voidaan vaihtaa modulaarisesti. Moduulien vaihtelu tapahtuu staattisesti käännösaikana, jolloin käytetyt moduulit valitaan *Interaction*-kerroksen luokkia instantioimalla C++-kielen *template*-rakennetta käyttäen. Protokollakirjaston kerrosarkkitehtuuri on esitetty kuvassa 3.1.




Kuva 3.1: Protokollakirjaston kerrosarkkitehtuuri

Protokollakirjaston läpikäynti tehdään alhaalta ylöspäin, perustoiminnoista kohti kirjaston käyttäjän näkemiä rajapintoja. Kirjasto on rakennettu C++-ohjelmointikielen oman STL-kirjaston (*Standard Template Library*, [10]) sekä erillisen *Boost*-laajennuskirjaston [11] päälle. Näiden päällä sijaitsee *Core*-kerros, joka tarjoaa joitakin perustason toiminnallisuuksia kirjaston ylemmille kerroksille. Tähän väliin kuuluu myöskin *Logging*-kerros,

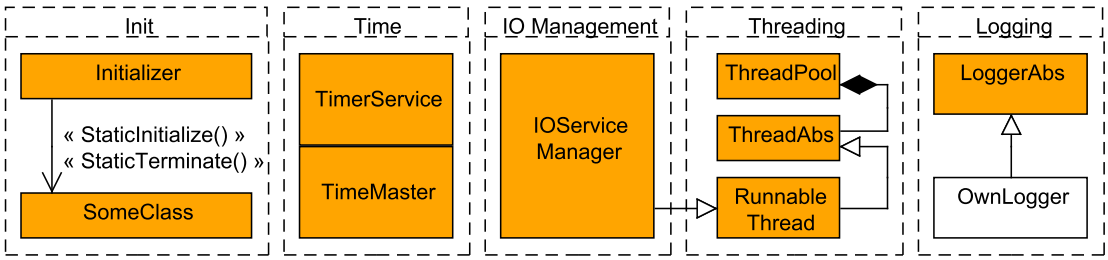
joka toimii protokollakirjaston kaikilla tasoilla keräten virheviestejä sekä tietoa kirjaston toiminnasta.

*Core*-kerroksen päälle on rakennettu *Transport*- ja *Marshalling*-kerrokset, joiden tehtävä on modularisoida yhteysmedia ja järjestää lähetettävä data järkevällä tavalla. *Transport*-kerros huolehtii siirtokanavan alustamisesta sekä viestien lähettämiseen kuuluvasta toiminnallisuudesta. Kirjoitushetkellä *Transport*-kerrokselle on toteutettu ainoastaan IP-yhteyksiä tukeva moduuli. *Marshalling*-kerros sisältää fysiologisen datan jäsentämiseen liittyviä luokkia, joita protokollakirjaston käyttäjä voi hyödyntää lähetettävän datan järjestämiseen.

Protokollakirjaston instanssi alustetaan *Interaction*-kerroksella, jonka tehtävänä on toimia alempia kerroksia yhdistävänä tekijänä. *Interaction*-kerros hyödyntää *Transport*- kerrosta halutunlaisen yhteyden muodostamiseksi tavalla, joka mahdollistaa toiminnan tarpeen mukaan joko protokollan mukaisena asiakkaana tai palvelimena. Kerros toimii myös protokollayhteyden toisen pään ja oman *Marshalling*-kerroksen dataluokkien välisenä välittäjänä. *Interaction*-kerros koostuu moduuleista, jotka määräävät tarkemmin kahden kirjastopinon välisen kommunikaatiotavan. Tähän kuuluu esimerkiksi protokollan alustuksessa lähetettävien kättelyviestien sekä XML-kuvausviestien sisältö. Kirjoitushetkellä toteutettuna on ainoastaan kahden yksittäisen laitteen (*device*) välinen kommunikaatio.

## 3.2 Core-kerros

*Core*-kerroksen tarkoituksena on luoda yhtenäinen pohja ylempien kerrosten rakentamiselle sekä tarjota joitakin yleisiä tietorakenteita ja apuluokkia ylempien kerrosten käyttöön. Kerros sisältää toiminnallisuutta kirjaston staattisten osien alustamiseen, ajastimiin ja ajanhallintaan sekä I/O-liikenteen hallintaan liittyen. Tämän lisäksi *Core*-kerroksesta löytyvät luokat alustakohtaisten säikeiden toteuttamiseen sekä lokitietojen keräämiseen ja tallentamiseen. *Core*-kerroksen komponentit on esitetty kuvassa 3.2.




Kuva 3.2: *Core*-kerroksen osat kuvattuna vastuualueiden mukaan.

Protokollakirjasto on toteutettu niin, että useat sen komponentit tuovat reflektiomaisia piirteitä C++-kielen tarjoaman toiminnallisuuden päälle. Reflektiolla tarkoitetaan tässä sitä, että ohjelma pystyy ajon aikana tutkimaan omia tietorakenteitaan [12, s.193]. Esimerkiksi alikohdassa 3.6.2 esitetty enumeraatioiden kapselointi on tämänkaltainen omi-

naisuus. Tämän mahdollistamiseksi kirjaston staattiset osat on alustettava ennen kirjaston käyttöönottoa. Käytännössä tämä tarkoittaa *StaticInitialize*()-jäsenfunktioiden toteuttamista luokille siten, että *Core*-kerroksessa oleva *Initializer*-luokka pystyy alustamaan ne. Tämänkaltainen toiminnallisuus löytyy kaikista protokollakirjaston kerroksista, vaikka sitä ei ole jatkossa otettu mukaan kuviin ja tekstikuvakseen.

Vaikka kyseessä ei ole reaaliaikaprotokolla, protokollan toteutus tarvitsee kuitenkin joukon erilaisia ajastimia. Lisäksi on hyödyllistä tarjota protokollakirjaston käyttöön yhtenäinen tapa nykyisen päivämäärään sekä kellonajan esittämiseen. *Core*-kerroksen *TimerService*-luokka sisältää keinot kertatoimisten ja toistuvien ajastusten luomiseen. *TimeMaster*-luokka taas sisältää funktiot nykyisen ajankohdan muuntamiseen protokollan käyttämään merkkijonoesitykseen sopivaksi.

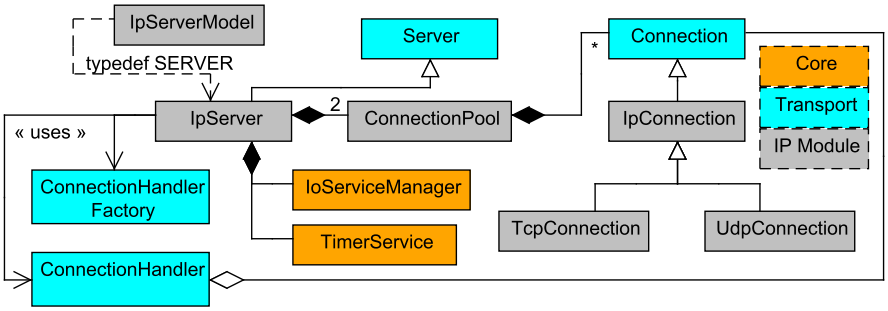
*Core*-kerroksesta löytyy myös *IOServiceManager*-luokka erilaisten I/O-toimintojen toteuttamiseen. Luokka toimii yleisenä rajapintana verkkoliikenteessä oleville kokonaisuuksille. Tällaisia ovat esimerkiksi soketti (*socket*), yhteyden hyväksyjä (*acceptor*) sekä erilliset käyttäjän määrittämät palvelut.

Protokollakirjastoa tullaan käyttämään lukuisissa eri käyttöjärjestelmissä, jolloin säikeiden toteutusta ei voida kirjoittaa suoraan kirjaston koodiin. Protokollakirjasto tarjoaa *ThreadAbs*-luokan, jonka avulla käyttäjä voi toteuttaa säikeet alustalle sopivalla tavalla. Erikseen on vielä määritelty *RunnableThread*-luokka, joka tarjoaa virtuaalisen *runService*()-funktiokutsun siltä varalta, että käyttäjän tarvitsee ajaa jotain omaa koodiaan erillisessä säikeessä. *Core*-kerros tarjoaa vielä erikseen määritellyn *ThreadPool*-luokan, jossa käyttäjä voi määrittää haluamansa määrän säikeitä protokollakirjaston käyttöön.

Joissakin tehottomissa alustoissa esimerkiksi tiedostoon kirjoitus on liian raskas operaatio, ja siksi myös lokitiedon tallennus pitää pystyä tekemään alustariippuvaisesti. Tätä varten *Core*-kerros tarjoaa *LoggerAbs*-rajapinnan, josta periyttämällä protokollakirjaston käyttäjä voi määrätä, mihin hän haluaa lokitiedon ohjata. Tämä mahdollistaa myös protokollakirjaston lokien liittämisen laajemman järjestelmän (esimerkiksi potilasvalvontamonitori) lokijärjestelmään.

## 3.3 Transport-kerros

*Transport*-kerroksen tehtäväksi määriteltiin yhteyksien luominen sekä viestien lähettämiseen kuuluva toiminnallisuus. Olennaista on, että *Transport*-kerrokselle voidaan toteuttaa useita moduuleja erilaisten viestikanavien tukemiseen. Tietyn moduulin valinta tehdään myöhemmin *Interaction*-kerroksella, missä malliksi valitaan *IpServerModel*-luokan mukainen moduuli. Jotta tulevaisuudessa voidaan varmistua riittävästä joustavuudesta erilaisten yhteystyyppien suhteen, yleisenä periaatteena on toiminnallisuuden pitäminen moduulien puolella yhteisten kantaluokkien sijaan. Yleiskuva *Transport*-kerroksen luokista on esitetty kuvassa 3.3.




Kuva 3.3: *Transport*-kerroksen yleisarkkitehtuuri IP-moduulin tapauksessa.

*Transport*-kerroksen ydin on Server-luokka, joka tarjoaa perustoiminnallisuuden erilaisten yhteyksien hallintaan. TCP/UDP-yhteyksiä käytettäessä valitaan moduuliksi *IpServerModel*, joka määrittää tyyppinimittämällä (*typedef*) *IpServer*-luokan toimimaan yhteyskäsittelijänä. *Transport*-kerroksen yleiseen arkkitehtuuriin kuuluu lisäksi *ConnectionHandler* ja *ConnectionHandlerFactory*-rajapinnat, joilla käytetyille yhteyksillä luodaan käsittelijät. Seuraavassa on käsitelty tarkemmin moduulikohtaisia *IpServer* ja *IpConnection*-luokkia.

### 3.3.1 IpServer

*IpServer*-luokan tarkoitus on mahdollistaa protokollakirjaston käyttö TCP/IP ja UDP/IP verkoissa. *Transport*-kerros ei erikseen määrää kuinka kukin moduuli yhteyksiään hallitsee, joten tähän tarkoitukseen on toteutettu erillinen *ConnectionPool*-luokka. *ConnectionPool*-luokan instansseja luodaan yksi kappale sekä sisään- että ulospäinsuuntautuneille yhteyksille. Erona yhteystyyppien välillä on niiden muodostumistapa: ulospäinsuuntautuneita yhteyksiä luodaan kutsumalla *IpServer*-luokan *openConnection*()-funktiota, sisäänpäinsuuntautuneita yhteyksiä syntyy puolestaan *listenConnection*()-funktiokutsun seurauksena. Huomattavaa tässä on, että TCP-yhteyden kuuntelijalla ja asiakkaalla ei ole tekemistä protokollan palvelimen ja asiakkaan kanssa, vaan ne voivat olla keskenään erilaiset. Esimerkiksi monitori odottaa yhteydenottoa moduulilta (joka on TCP-asiakas), mutta silti moduuli toimii protokollassa datan tuottajana (palvelin).

Yhteyksien luomiseen ja hallinnointiin *IpServer*-luokka käyttää *Transport*-kerroksen tarjoamia *ConnectionHandler* ja *ConnectionHandlerFactory*-luokkia. Uudet yhteydet lisätään haluttuun *ConnectionPooliin*, ja niille luodaan käsittelijät myöhemmin esiteltävän *InteractionModelin* määräämällä *ConnectionHandlerFactorystä* perityn luokan instanssilla. *ConnectionHandler*-luokalla on hallinnointia varten osoitin luotuun yhteyteen.

Varsinaisten yhteyksien muodostamiseen *IpServer* käyttää *Core*-kerroksesta löytyvää *IoServiceManager*-luokkaa. Sieltä löytyy valmiudet sokettien luomiseen ja sisääntulevien yhteyksien kuunteluun. Yhteyksiin kirjoitettavan ja niistä luettavan datan hallintaan *IpServer* käyttää *Core*-kerroksesta löytyviä *TimerService*-luokan tarjoamia ajastimia.

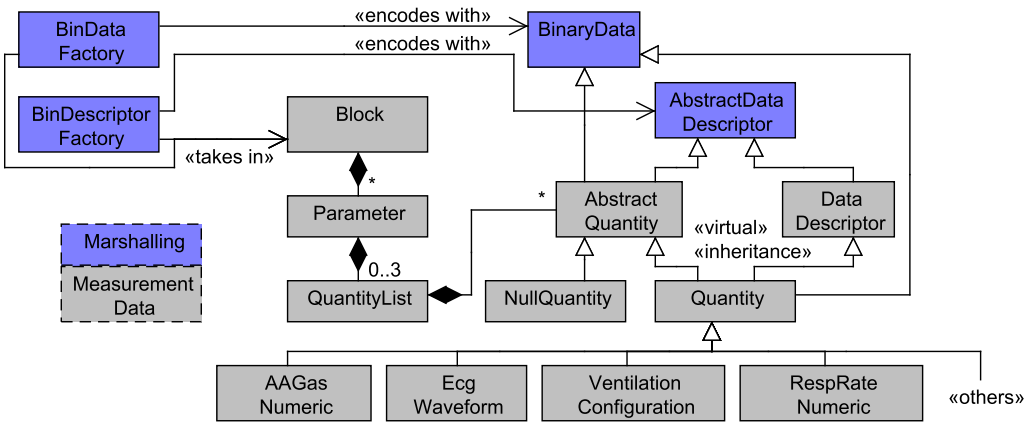
### 3.3.2 IpConnection

*IpServer*-luokan tavoin *IpConnection*-luokka ja siitä perityt luokat on erikoistettu toimimaan TCP/IP ja UDP/IP verkoissa. Erityisesti periytetyt luokat *TcpConnection* ja *UdpConnection* toteuttavat kullekin yhteystyypille ominaisen alustuksen. Alustus hoidetaan *IpServer*-luokan välittämällä *IOServiceManager*-luokan instanssilla.

Kunkin *Connection*-luokasta perityn luokan tehtävänä on tarjota toimituskanava jollekin protokollan kontrolli- tai datayhteydelle. *IpConnectionin* tapauksessa tämä tarkoittaa sitä, että jokaisella laitteella on kaikille protokollayhteyksille yhteinen IP-osoite. Protokollan sisäisten yhteyksien erottaminen tehdään käyttämällä IP-protokollaan kuuluvaa portin käsitettä. Käytännössä jokaiselle yhteydelle varataan portti järjestelmästä, minkä lisäksi yksi portti varataan paikalliseksi kuunteluportiksi, jota kautta laitteeseen voidaan avata uusia yhteyksiä.

## 3.4 Marshalling-kerros

*Marshalling*-kerroksen tehtävänä on jäsentää fysiologinen mittausdata protokollan ymmärtämään muotoon. Oleellisena osana tähän kuuluu varsinaisen datan jäsentävät luokat, ja sen lisäksi käytetään joitakin tehdasluokkia jäsennetyn datan muuttamiseksi XML-kuvaukseksi tai binääridataksi. Datan mallinnus vastaa aiemmin alikohdassa 2.2.3 kuvattua rakennetta. *Marshalling*-kerroksen yleisrakenne on esitetty kuvassa 3.4.




Kuva 3.4: *Marshalling*-kerroksen toiminnallisuus ja sen tarjoamat dataluokat.

Yleisesti *Marshalling*-kerros toimii protokollakirjaston osana tarjoten sille *BinDataFactory*- ja *BinDescriptorFactory*-luokat datankäsittelyn tueksi. Näiden luokkien vastuulla on erottaa annetusta blokista tilanteesta riippuen joko XML-kuvaukseen lisättävä metatieto tai vaihtoehtoisesti kuvauksen mukaan järjestetty mittausdata binääriesityksessä. Varsinaisten dataluokkien mittausdataa sisältävät *Quantity*-osiot peritään *Factory*-luokkien käyttämistä *BinaryData*- ja *AbstractDataDescriptor*-luokista.

### 3.4.1 Dataluokat

Protokollakirjaston dataluokat ovat protokollan alustamisen jälkeen käyttäjälle eniten näkyvä kirjaston osa. Nämä luokat vastaavat tarkasti aiemmin esitettyä protokollan datajärjestystä, ja näiden tehtävänä onkin toimia mittaustulosten tallennuspaikkana yhteyden molemmissa päissä. Yleisesti ajatellen voidaan sanoa, että jos protokollakirjasto toimii protokollassa datan tuottajana (palvelin), kirjastoa käyttävä ohjelmisto omistaa dataluokat. Jos se taas toimii datan vastaanottajana, takaisinkutsu saadaan vektorillisena (*std::vector*) *Block*-luokan instansseja aina uusien mittaustulosten saapuessa.

Uutta datayhteyttä luotaessa protokollakirjasto ottaa sille annetut *Block*-instanssit ja lähettää niiden sisältämän mittausdatan kuvauksen XML-muodossa vastaanottajalle. Vastaanottaja saa XML-kuvauksen ja koostaa siitä itselleen tyhjän *Block*-instanssin jokaista lähetettyä blokkia kohden. Tämän jälkeen datan tuottaja lähettää omistamistaan blokeista koottuja binääridatapaketteja, jotka vastaanottopää sijoittaa saamaansa blokkirunkoon täytteeksi.

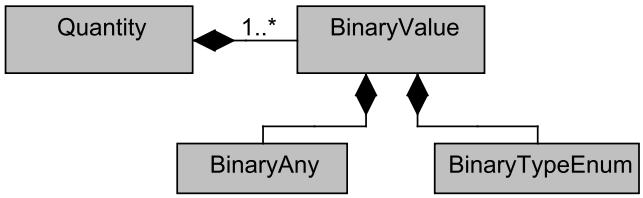
Ylimmän tason dataluokka on siis *Block*-luokka. *Block* voi pitää sisällään useita *Parameter*-luokan instansseja, jotka kukin kuvaavat yhtä mitattavaa parametria, esimerkiksi EKG. Kuten aiemmin kerrottiin, parametriin liittyy kolmen tyyppistä mittausdataa: numeerisia mittauksia, aaltokäyrää ja konfiguraatioarvoja. *Parameter*-luokassa on jokaiselle niistä *QuantityList*-luokka, joka sisältää vektorillisen *AbstractQuantity*-luokan toteutuksia.

*AbstractQuantity* voi esittää mitä tahansa kolmesta mainitusta mittaustyypistä, ja se pitää sisällään tiedon mittauksen arvoista, mittayksiköistä, binääriesityksestä ja niin edelleen. *AbstractQuantity* ei kuitenkaan tee kaikkea tätä itse vaan luottaa kantaluokkiinsa ominaisuuksien toteuttamisessa. Mittausten binäärimuotoon muuttamista varten luokka peritään *BinaryData*-luokasta, ja XML-kuvausten generointia varten se peritään myös *AbstractDataDescriptor*-luokasta. *AbstractQuantity* noudattaa *nollaobjekti*-suunnittelumallia, joka mahdollistaa toiminnallisuudeltaan tyhjän *NullQuantityn* lisäämisen kaikkialle missä myös varsinaista *Quantity*-luokkaa voidaan käyttää [13, Null Object Pattern].

*Quantity*-luokka toimii *AbstractQuantityn* instantioitavana toteutuksena, joka pitää sisällään kaiken perusmittauksen esittämiseen tarvittavan tiedon. Mikäli mittauksessa on vielä jotakin lisäinformaatiota, *Quantity*-luokasta voidaan periyttää erikseen erikoistapauksia tätä lisäinformaatiota varten. *Quantity*-luokka periytyy vielä erikseen *AbstractDataDescriptorista* periytyvästä *DataDescriptor*-luokasta, jolla on saatu piilotettua joitakin *AbstractDataDescriptorin* suurimpia yleistyksiä. Tässä syntyvän kaksinkertaisen moniperintätilanteen ratkaisemiseksi on käytetty *virtuaalista periytymistä*, jolla varmistetaan, että kukin kantaluokan edustaja on perityssä luokassa ainoastaan yhden kerran [14].

### 3.4.2 Binäärityypit

Protokollan toteutuksen kannalta olennaista on, että kukin mittaustulos voidaan esittää jollakin lukuisista eri binäärityypeistä. Olennaista tämä on eritoten siksi, että siirtokaistan ollessa kapea, lähetetyn datan määrää täytyy voida pienentää jollakin järkevällä tavalla. Binääridatan serialisointiin käytetään *BinaryData*-luokkaa, mutta yleisemmin datan tallentamiseen on käytössä erillinen *BinaryAny* sekä sitä käyttävä *BinaryValue*. *BinaryValue*-luokkaa käytetään datan varastoimiseen *Quantity*-luokan sisälle, kuten kuvasta 3.5 käy ilmi.



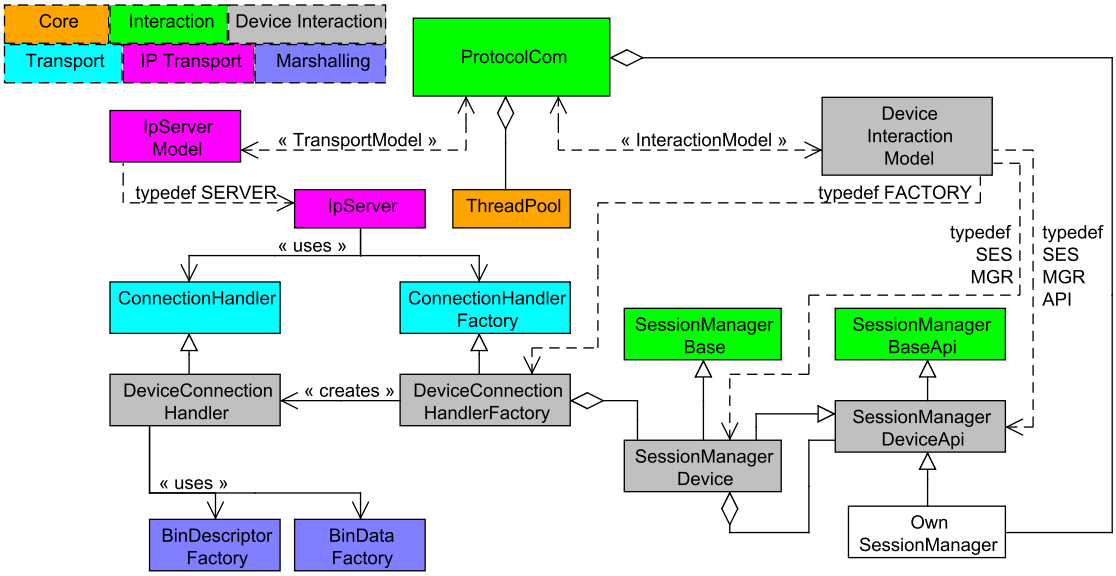

Kuva 3.5: *BinaryValue*-luokan koostuminen ja käyttö *Quantity*-luokan osana.

*BinaryAny* on idealtaan samankaltainen kuin Boost-kirjastossa löytyvä *boost::any*, mutta se sisältää lisäksi jonkin verran I/O-toiminnallisuutta. Ideana on, että luokkaan voidaan tallentaa oleellisia C++-kielen numeerisia perustietotyyppejä, joiden välillä on lisäksi pystyttävä tekemään vertailuja. Lisäksi tietoja on voitava muuttaa merkkijonoksi tai muuntaa binäärityypistä toiseen. *BinaryAny*-luokan toteutus seuraa Kevlin Henneyn *Valued Conversions*-julkaisussa annettuja periaatteita [15].

Varsinaiset binäärimuotoon muutettavat arvot on tarkoitus tallentaa *BinaryValue*- luokkaan, joka sisältää varsinaisen binääriesityksen lisäksi tiedon binääridatan tyypistä. Tyyppi talletetaan *BinaryTypeEnum*-luokalla, joka sisältää listauksen kaikista protokollan tukemista binäärityypeistä. *Quantity*-luokassa on talletettavan datan tyypistä riippuen tallessa yksi tai useampi binääriarvo. Numeroarvoisten mittausten sekä konfiguraatiodatan tapauksessa arvoja on yksi, aaltokäyrien tapauksessa useampia.

## 3.5 Interaction-kerros

*Interaction*-kerros on *Marshalling*-kerroksen jälkeen erityisesti protokollan instantiointivaiheen aikana käyttäjälle eniten näkyvä osa protokollapinossa. Kerroksen tarkoitus on yhdistää alemmilta kerroksilta luodut moduulit yhdeksi käyttökelpoiseksi kokonaisuudeksi. Oleellisena osana protokollan instantioinnissa ovat mallit, joita *Interaction*-kerroksella on käytössä *TransportModel* ja *InteractionModel*. Kerros toimii myös datan välittäjänä *Marshalling*-kerroksen dataluokista *Transport*-kerroksen *Serverin* kautta ulkomaailmaan. Yleiskuva *Interaction*-kerroksen rakenteesta ja staattisesta käyttötavasta on esitetty kuvassa 3.6.




Kuva 3.6: *Interaction*-kerroksen moduuli yhdistää *Transport*- ja *Marshalling*-kerrokset.

Alustusvaiheessa kirjaston staattisen alustuksen jälkeen luodaan *ProtocolCom*-luokan instanssi, jolle annetaan *template*-parametreina jokin *TransportModel*- sekä jokin *InteractionModel*-tyyppinen luokka. Malleja ei ole peritty mistään, mutta *ProtocolComissa* oleva käsittely odottaa tiettyjä asioita malleiksi annettavilta luokilta. Oleellisinta on, että mallit määrittävät uudestaan tietyt tyyppinimet siten, että *ProtocolCom* pystyy tunnistamaan ne. Esimerkiksi *IpServerModel* määrittää *SERVER*-nimen tyypiksi *IpServer*-luokan, jolloin *ProtocolCom* voi *SERVER*-määritelmää käyttäen luoda instanssin *IpServeristä*.

Yksi *Interaction*-kerroksen päätehtävistä on tarjota kirjaston käyttäjälle mahdollisuus protokollayhteyksien hallintaan soveltuvan luokan erikoistamiseen. Tähän kerros käyttää *InteractionModel*-tyyppistä *template*-parametria, joka kertoo tarkemmin kuinka ja minkälaisia viestejä protokollakirjasto lähettää verkkoon. *InteractionModel* määrittää *SES_MGR*- ja *SES_MGR_API*-nimien avulla protokollakerroksen ulkoiseen ja sisäiseen viestinvälitykseen käytettävien hallintaluokkien tyypit. Lisäksi on oleellista, että malli määrittää *FACTORY*-nimelle jonkin *Transport*-kerroksessa mainitun *ConnectionHandlerFactoryn* erikoistuksen, joka on tässä tapauksessa siis *DeviceConnectionHandlerFactory*. Tämän kautta protokollakirjasto ilmoittaa esimerkiksi *IpServerille*, minkälaisia erikoistoimenpiteitä sen tarvitsee suorittaa uusia yhteyksiä avattaessa.

Kaikkea ylläkuvattua tarvitaan protokollakirjaston toiminnan takaamiseksi. Kirjaston käyttäjän ei kuitenkaan yleensä tarvitse käyttää esimerkiksi uudentyppistä siirtoväylää datan välittämiseen, vaan hänelle riittävät olemassa olevat *Transport*- ja *Interaction*-mallit. Tällöin suuri osa kuvatusta toiminnallisuudesta on piilotettu *ProtocolComin template*-rajapinnan taakse. Mallien lisäksi käyttäjä antaa kirjastolleen *Core*-kerroksesta tutun *ThreadPool*-instanssin, joka sisältää halutun määrän kirjaston käyttöön annettavia säikeitä. Viimeinen, mutta samalla suuritöisin, *ProtocolComille* annettava parametri on kirjaston

käyttäjän *InteractionModelin* nimeämästä rajapintaluokasta perimä istuntojen hallitsija (*OwnSessionManager*).

Istuntojen hallintaan liittyvät *SessionManager*-hierarkian luokat ja *Transport*-kerroksen palvelinten käyttämät *ConnectionHandlerit* kuvataan tarkemmin seuraavissa alikohdissa.

### 3.5.1 SessionManagerDevice

Tähän mennessä on esitelty lähinnä yleisiä datan siirtämiseen tai esittämiseen soveltuvia luokkia. Varsinaisen protokollaistunnon hallinnan hoitaa niin sanottu *SessionManager*, joka voi toimia protokollassa joko datan lähetyspään palvelimena tai dataa vastaanottavana asiakkaana. *SessionManagerBaseApi*-luokka toimii yhteysrajapintana protokollakirjaston ja sovelluslogiikan välillä, ja se pitää sisällään kaikki funktioesittelyt, joita istunnon hallintaan tarvitaan. Asiakkaalle ja palvelimelle ei siis ole erillisiä *Client/ServerSessionManagerApi*-kantaluokkia, vaan erottelu tehdään protokollan käyttäjän määrittämässä *OwnSessionManager*-luokassa. Rajapintaluokan lisäksi *Interaction*-kerroksesta löytyy myös protokollan sisäiseen hallintaan erikoistunut *SessionManagerBase*-luokka, jossa varsinainen hallintalogiikka sijaitsee.

Valitessaan esimerkiksi *DeviceInteractionModelin* protokollakirjaston käyttäjä määrittelee, mitä yllä olevista luokista perittyjä erikoistusluokkia halutaan käyttää. *SessionManagerDeviceApi*-luokan tehtävänä on laajentaa *SessionManagerBaseApi*-rajapintaluokkaa siten, että *Device*-interaktiomallin vaatimat lisäominaisuudet voidaan välittää käyttäjälle. Varsinainen *Device*-interaktiomallin toiminnallisuus löytyy *SessionManagerDevice*-luokasta, joka on peritty myös *SessionManagerDeviceApi*-rajapintaluokasta yhtäläisen toiminnallisuuden takaamiseksi. *SessionManagerDevice* käyttää *SessionManagerDeviceApi*-osoitinta toimiessaan protokollakirjaston käyttäjän toteuttaman *OwnSessionManagerin* kanssa.

*SessionManagerDevice*-luokka käyttää siis *SessionManagerDeviceApi*-rajapintaluokkaa tiedon välittämiseen protokollakirjaston käyttäjälle. *SessionManagerDeviceApi* erikoistaa ja laajentaa *Interaction*-kerroksen perustoteutusta, mutta jättää silti osan virtuaalifunktioista määrittelemättä (*pure virtual*).

Protokollakirjaston käyttäjä periyttää *SessionManagerDeviceApi*:sta *OwnSessionManager*-luokan, joka toimii datan välittäjänä protokollakirjaston ja varsinaisen sovelluslogiikan välillä. Toimiessaan protokollan palvelimena protokollakirjasto pyytää *OwnSessionManager*-luokan kautta sovellusohjelmalta kuvauksia lähetettävistä blokeista ja hyväksyttää yhteyspyyntöjä. Toimittaessa protokollan asiakkaana *OwnSessionManager*-luokkaan periytyneiden funktioiden avulla voidaan pyytää dataa palvelimelta, minkä jälkeen protokollakirjasto antaa *OwnSessionManagerille* binääridatalla valmiiksi alustettuja blokkeja. Usein protokollakirjasto toimii laitteessa joko ainoastaan datan tuottajana tai kuluttajana. Tästä johtuen osalla käyttäjän *OwnSessionManager*-luokan funktioista ei ole tehtävää, vaan ne on määritetty ainoastaan täyttämään rajapinnan vaatimukset.

### 3.5.2 DeviceConnectionHandler

*Transport*-kerroksessa määritelty *ConnectionHandler* on varsin kevyt luokka, joka on suunniteltu *Transport*-kerrosta silmällä pitäen. Jotta tämä toiminnallisuus voi toimia valitun *InteractionModelin* kanssa, sitä on laajennettava kattamaan *SessionManagerDevicen* vaatimukset. *DeviceConnectionHandler* pitää sisällään *Device*-interaktiomalliin liittyvien viestien muodostamis- ja vastaanottologiikan.

Oikean yhteystyypin valinta tehdään käyttäen *abstrakti tehdas* -suunnittelumallia. Mallin mukaisesti *DeviceInteractionModel* määrittää, mitä konkreettista tehdasta (*DeviceConnectionHandlerFactory*) käytetään. Tämän jälkeen tehdas luo *SessionManagerDevicen* käyttöön instansseja *DeviceConnectionHandler*-luokasta. [13, Abstract Factory Pattern]

*DeviceConnectionHandler* käyttää *Marshalling*-kerroksesta löytyviä apuluokkia *SessionManagerilta* tai *Transport*-kerroksen serveriltä tulevan datan käsittelyyn. Lähetettäessä dataa ulospäin *DeviceConnectionHandler* käyttää *BinDescriptorFactory* ja *BinDataFactory*-luokkia irrottamaan blokeista halutun informaation lähetystä varten. Kun dataa vastaanotetaan verkosta, *DeviceConnectionHandler* syöttää saapuneen datan suoraan blokkeihin.

## 3.6 Rakenteellisia yksityiskohtia

Protokollakirjastossa on edellä kuvattujen asioiden lisäksi joukko rakenteellisia yksityiskohtia, joilla on suuri merkitys kirjaston käyttäjälle. Seuraavassa on listattu toteutettavan simulaattorin kannalta olennaisimmat yksityiskohdat.

### 3.6.1 Boost

Protokollakirjaston toteutuksessa on käytetty vahvasti Boost-ohjelmakirjastoa, tarkemmin sanottuna sen tiettyä osajoukkoa. Boost jakautuu useisiin alikirjastoihin, joista jokainen on suunniteltu ratkaisemaan jokin tietty osaongelma. Boost on rakennettu pitkälti toimimaan pelkkien otsikkotiedostojen (*header file*) kautta, jolloin kirjaston lisääminen projektiin on yksinkertaista. Erikseen linkittämistä ei useimpien alikirjastojen kanssa tarvita, ja tarvittaessa Boost-kirjaston automaattinen linkitysrajapinta huolehtii tästä käyttäjän puolesta. Kirjaston kehityksessä on panostettu huomattavasti geneerisyyteen, minkä vuoksi C++:n *template*-rakenne on käytössä laajasti läpi kirjaston. Boost-kirjasto on avattu myös protokollakirjaston käyttäjälle, joten seuraavassa sitä tarkastellaan myös tältä kannalta. [11]

Protokollakirjaston kehitys on kesken, mutta tällä hetkellä Boost-kirjastosta on käytössä versio 1.46.1. Versio on toimintaympäristön sanelema, sillä tässä oletetaan käytettäväksi samaa Boost-versiota kuin valmistajan tulevassa monitorissakin. Aiemmin käytössä oli Boostin versio 1.43.0, josta luopuminen mahdollisti joidenkin asioiden tekemi-

sen helpommin varsinaisessa sovelluksessa. Vaihto tapahtui kuitenkin vasta simulaattorin kehityksen ollessa käynnissä, mikä aiheutti jonkin verran lisätyötä muuttuneiden ominaisuuksien takia.

Ehkä yleisimmin käytössä oleva osa on *smart_ptr*-alikirjasto, joka pitää sisällään joukon kehittyneempiä osoittimia (*pointer*). Kehittyneistä osoittimista käytössä on *shared_ptr*, joka tallentaa osoittimen helposti kopioitavaan rakenteeseen. Rakenne pitää kirjaa samasta osoittimesta tehtyjen kopioiden lukumäärästä ja poistaa (C++:n *delete*) olion vasta, kun viimeinen osoitin on poistettu. Osoittimien kautta tehtävä toiminnallisuus helpottuu tuntuvasti *shared_ptr*-osoittimia käytettäessä, sillä suuri osa muistinhallintaongelmista katoaa tuhoamisvastuun siirtyessä osoitintoteutukselle.

*Core*-kerroksen toiminnallisuuden toteuttamiseen käytettiin Boostin *thread*- ja *asio*-alikirjastoja. Rajapinnat sokettien hallintaan sekä asynkronisten kutsujen tekemiseen datan kirjoituksessa löytyvät *asio*-alikirjastosta. Protokollakirjastossa ei käytetty suoraan *thread*-alikirjastoa, mutta se on osana protokollakirjaston mukana tulevaa Boost-osajoukkoa. Alikirjaston avulla kohdassa 3.2 esitetty *ThreadAbs*-luokka on helppo toteuttaa alustariippumattomasti.

Erityismaininnan ansaitsee *thread*-kirjastosta löytyvä *mutex*-luokka, joka tarjoaa alustariippumattoman ja atomisen tavan estää ohjelman kriittiselle alueelle pääsy enemmän kuin yhdeltä säikeeltä kerrallaan. Tämän lisäksi mukana on *interprocess*-alikirjastosta löytyvä *scoped_lock*, joka lukitsee *mutexin* rakentajassa ja avaa lukon, kun *scoped_lock*-itse tuhoutuu. Tyypillinen käyttötarkoitus on lukita jokin resurssi kiinni funktion suorituksen ajaksi, jonka jälkeen lukko puretaan riippumatta siitä, mistä funktion kohdasta *return*-lausetta kutsutaan.

Lopuksi on olennaista mainita kirjaston sisäiseen datanvälitykseen käyttämä *property_tree*-alikirjasto. Kirjastosta löytyvää *basic_ptree*-luokkaa käytetään parametrien tallentamiseen blokin sisälle. Mielivaltaisen syvässä puurakenteessa olevia avain-arvo-pareja kutsutaan *property_treeksi*, jos niillä on lisäksi tieto myös omista lapsistaan. Sinällään luokka ei määrää avainten ja arvojen tyyppiä, mutta useimmissa tapauksissa ainakin avain on *string*-muotoinen. Protokollakirjastossa käytetään *basic_ptree*-luokkaa parametrien tallentamiseen *template*-erikoistuksella, missä avaimena on *string* ja arvona *shared_ptr Parameter*-luokan instanssiin. Indeksointi on mahdollista myös *property_tree*-luokan polulla, joka muodostetaan *string*-muotoisena liittämällä yhteen ylimmän solmun sekä polulla olevien lapsien avaimet.

### 3.6.2 Enumeraatioiden kapselointi

C++-kielen oma enumeraatiojärjestelmä on melko rajoittunut, mikä osaltaan johti protokollakirjaston kehittäjien tarpeeseen tehdä perustoteutuksen ympärillä toimiva kapselointi. Oleellisin ongelma liittyi enumeraatioiden muuntamiseen *string*-merkkijonoksi ja siitä takaisin. Esimerkki kapselointiluokan otsikkotiedostosta on annettu listauksessa 3.1.

```
1  class SomeEnum
2  {
3  public:
4    static bool StaticInitialize();
5    static bool StaticTerminate();
6    enum Type
7    {
8      ValueA,   //"ValueA"
9      ValueB    //"ValueB"
10   };
11   static Type toEnum(const std::string& rSomeEnumStringP);
12   static const std::string& toString(const Type eEnumValueP);
13 private:
14   SomeEnum();
15   virtual ~SomeEnum();
16   SomeEnum( const SomeEnum& other);
17   SomeEnum& operator=(const SomeEnum& other);
18 };
```

Listaus 3.1: Enumeraatioiden kapselointi, esimerkki luokan otsikkotiedostosta.

Varsinainen kapseloitava enumeraatio tunnetaan julkisella puolella *Type*-nimellä, jossa kunkin arvon merkkijonoesitys on selkeyden vuoksi kirjattu kommenttina arvon perään. Kapselointiluokan *public*-rajapinta sisältää *Typen* lisäksi ainoastaan staattisia funktioita, ja luokan instantioiminen, kopiorakentaja- ja sijoitusoperaattorit on kielletty sijoittamalla ne *private*-puolelle. Kohdassa 3.2 esitellyn *Core*-kerroksen *Initializer*-luokan käyttöön on lisätty *StaticInitialize*()- ja *StaticTerminate*()-funktiot, jotka huolehtivat merkkijonomuutostaulukoiden alustamisesta. Funktiot *toEnum*() ja *toString*() vastaavat muutostauluja hyödyntäen enumeraatioiden muuttamisesta merkkijonomuotoon ja takaisin.

Kehitetty kapselointi on siisti ratkaisu C++:n kannalta, ja se tekee enumeraatioista huomattavasti aiempaa voimakkaamman työkalun. Kapseloinnin määrittelyssä on jääty kuitenkin osittain puolitiehen. Jostakin syystä kaikki enumeraatiot on toteutettu omassa luokassaan, eikä yhteistä kantaluokkaa ole katsottu tarpeelliseksi. Toinen kaivattu ominaisuus olisi jonkinlaisten sääntöjen sopiminen kaikkien arvojen läpikäymiseksi järjestyksessä. Tässä toimisi parhaiten iteraattoripohjainen ratkaisu, jossa jokainen luokka tarjoaisi *begin*()- ja *end*()-funktiot STL-säiliöiden tapaan. Nyt vaikuttaa joskus siltä, että hyvästä ideasta jää joitakin merkittäviä mahdollisuuksia käyttämättä.

# 4. SIMULAATTORIN SUUNNITTELUPERUSTEET

Edellä esitelty protokolla ja testausympäristö asettavat kehitettävälle simulaattorille tiettyjä vaatimuksia. Nämä on seuraavassa jaettu ympäristöön, toiminnallisiin vaatimuksiin sekä käytettyihin työkaluihin liittyviin reunaehtoihin. Lisäksi kuvataan ne helpotukset, jotka sovelluskohde tarjoaa tietyistä ohjelmistoteknisesti hankalista yksityiskohdista. Lopuksi käydään läpi vaatimusten perusteella laadittuja suunnitteluperiaatteita ja -käytäntöjä.

## 4.1 Vaatimukset

Varsinaiset vaatimukset ovat monitorin valmistajan käyttötapauksiin perustuvia vaatimuksia, jotka rajaavat toteutettavaa järjestelmää tiettyihin valintoihin. Varsinaisiin vaatimuksiin kuuluvat valmistajan toimintaympäristö, varsinaiset simulaattorin toiminnallisuuteen liittyvät vaatimukset sekä käytettyjen työkalujen ja kirjastojen yhteensopivuuteen liittyvät reunaehdot. Varsinaisiin vaatimuksiin kuuluu myös tiettyjä helpotuksia, jotka johtuvat osittain käyttötapauksista ja osittain toimintaympäristön rajoitteista.

### 4.1.1 Toimintaympäristö

Simulaattorin toimintaympäristön vaatimukset tulevat suoraan monitorin testaukseen käytetyn ympäristön vaatimuksista. Suuri osa monitorin kehitystyöstä tehdään Windows-alustalla, jolloin myös **simulaattori toimii Windows XP- sekä Windows 7- käyttöjärjestelmillä**. Windows-alusta ei rajaa pois simulaattorin hyödyntämistä varsinaisella alustalla, sillä simulaattori tulee käyttämään yleisesti tunnettuja tietoliikenneprotokollia.

Simulaattorin pääasiallinen käyttötarkoitus on toimia testiautomaation osana, joten peruskäytössä ihmiskäyttäjä ei käskytä simulaattoria suoraan. Tästä johtuen **simulaattori toimii komentoriviltä ja siinä ei ole erillistä graafista käyttöliittymää.** Kuten kohdassa 2.3 todettiin, simulaattoria tullaan käyttämään TCP-yhteyden yli. Tämä mahdollistaa jatkossa graafisen käyttöliittymän tekemisen erillään varsinaisesta simulaattorista.

Simulaattori on tarkoitettu kevyeksi työkaluksi, joten sen asentamisen on oltava vaivatonta. Tästä johtuen **simulaattori on staattisesti linkitetty eikä sillä ole merkitystä, mistä kansiosta se käynnistetään**. Kaikki simulaattoriin ulkoisesti liitettävät kirjastot on linkitettävä staattisesti, jolloin erikseen asennettavia dynaamisia kirjastoja tai muita riippuvuuksia ei synny. Poikkeuksena ovat käyttöjärjestelmän järjestelmäkirjastot, joiden voidaan olettaa löytyvän valmiina kohdealustalta.

### 4.1.2 Toiminnalliset vaatimukset

Simulaattorin toiminnalliset vaatimukset syntyvät kohdassa 2.3 kuvatun testausjärjestelyn asettamista vaatimuksista. Olennainen vaatimus on simulaattorin käyttötapa, jossa **simulaattori vastaanottaa käyttäjältä komentoja joko oletussyöttövirrasta (stdin) tai TCP-yhteyden yli**. Lisäksi myöhemmin saatetaan toteuttaa uusia käskylähteitä, mikäli simulaattorin käyttövaatimukset muuttuvat.

Automaatiotestauksessa testitapaukset ajetaan yksi toisensa jälkeen siten, että myös yksittäisen testitapauksen sisällä komennot suoritetaan yksi kerrallaan tietyssä järjestyksessä. Tästä johtuen **simulaattori toimii käsky/vastaus-periaatteella, missä edelliseen käskyyn vastataan aina ennen kuin uusi käsky otetaan vastaan**. Tämä yksinkertaistaa simulaattorin toteutusta, koska rinnakkaisesti suoritettaviin käskyihin ei tarvitse varautua. Myöhemmässä vaiheessa simulaattorin käyttövaatimukset saattavat muuttua, jolloin uusia komentoja on voitava lisätä helposti järjestelmän jatkoksi.

Koska testitapaukset tulee lopulta kirjoittamaan ihminen, simulaattorille lähetettävät komennot on voitava kirjoittaa ihmisen luettavassa muodossa. Tästä johtuen **sekä käskyt että vastaukset kirjoitetaan JSON-koodauksella.** JSON eli *Java Script Object Notation* on merkkipohjainen kevyehkö datan välitykseen soveltuva formaatti, jolle löytyy apukirjastojen avulla tuki kaikista yleisesti käytetyistä ohjelmointikielistä. JSON on suunniteltu ihmisten helposti ymmärrettäväksi, mistä on apua testitapauksia laadittaessa. [16]

Testauksen varmennettavuuden sekä simulaattorin yleisen käytön varmentamisen takia simulaattorin käytöstä on jäätävä selkeä raportti. Tämän vuoksi **simulaattori kirjaa tapahtumat aikaleimalla nimettyyn UTF8-koodattuun lokitiedostoon**. Simulaattori kirjoittaa saamansa käskyt ja protokollakirjaston kanssa käytetyt funktioparametrit lokitiedostoon. Komentoriviparametreilla voidaan määrittää tulostettavaksi myös kirjaston sisäisiä viestejä. Tarvittaessa lokitiedot voidaan myös kirjoittaa oletustulostusvirtaan (stdout).

Simulaattori rakennetaan alustavasti tukemaan rajallista joukkoa mahdollisista mittauksista, sillä monitoritoteutuksen tarvitsee tukea aluksi ainoastaan tiettyä osajoukkoa. Myöhemmässä vaiheessa simulaattoriin on syytä voida **lisätä uusia mittauksia kohtuullisen helposti ja suuria rakenteellisia muutoksia tekemättä**. Tässä rajoitutaan kuitenkin käsittelemään ainoastaan protokollakirjaston tukemia mittauksia.

### 4.1.3 Reunaehdot

Samoin kuin toimintaympäristön vaatimukset, myös työkaluvaatimukset perustuvat monitorin testaukseen ja kehittämiseen käytetyn ympäristön vaatimuksista. Simulaattori tullaan liittämään osaksi monitorin aputyökaluja, minkä vuoksi toteutusympäristöksi valitaan monitorin *WindowsDebug*-version kehityksessä käytettävä ympäristö. Tästä johtuen **simulaattori toteutetaan Microsoft Visual Studio 2010-ympäristössä**. Olennainen vaatimus on, että simulaattori käännetään VS2010-kääntäjällä.

Tulevaa käyttöä varten simulaattorin on tarkoitus testata myös toteutettavan protokollan työkaluja. Tästä syystä **simulaattorin toteutuksessa käytetään luvussa 3 kuvattua protokollakirjastoa**. Simulaattorin on tarkoitus samalla testata protokollakirjaston toimivuutta sekä tuottaa palautetta kirjaston tekijöille mahdollisista suunnittelun epäkohdista.

Simulaattorin toteutuksessa on syytä käyttää helppokäyttöisiä ja vapaasti saatavilla olevia apukirjastoja. **Simulaattori käyttää JSON-alkioiden jäsentämiseen JsonCpp-kirjastoa**, joka on kevyt ja yksinkertainen JSON-toteutus C++-kielelle. Kirjaston lisenssiehdot ovat erittäin vapaat, mikä mahdollistaa kirjaston käytön kaupallisessa tuotantoprojektissa. [17]

Lisäksi **simulaattori voi hyödyntää protokollakirjaston mukana tulevaa Boost-kirjastoa.** Simulaattorissa käytetään protokollakirjaston mukana tullutta Boost-kirjaston versiota, joka on sama kuin monitorissa oleva.

### 4.1.4 Käyttöoletukset

Käyttöympäristö sekä testausjärjestelmän vaatimukset antavat joitakin kevennyksiä toteutettavan simulaattorin toiminnallisiin vaatimuksiin. Ensiksikin **reaaliaikaominaisuuksia ei vaadita**, jolloin riittää, että simulaattori lähettää tietyssä aikaikkunassa oikean määrän paketteja. Reaaliaikaominaisuuksien toteuttaminen Windows-työpöytäkäyttöjärjestelmässä on hankalaa tai jopa mahdotonta, sillä käyttöjärjestelmästä puuttuu reaaliaikajärjestelmille ominainen ennustettavuus. Tässä toimitaan lisäksi protokollakirjaston asettamien rajoitteiden puitteissa.

Ottaen huomioon simulaattorin käyttökohteen sen ei tarvitse toimia osana suurempaa monitoriverkostoa. Tällöin riittää, että **simulaattori toteuttaa datan lähetyksen yhdelle alikohdassa 2.2.2 kuvatulle asiakasohjelmistolle kerrallaan**. Tämä yksinkertaistaa joitakin datan tallennukseen käytettäviä tietorakenteita. Simulaattori on kuitenkin syytä suunnitella siten, että useiden asiakkaiden palveleminen on myöhemmin mahdollista.

Simulaattorin ei ole tarpeen muuttaa lähetettävän mittausdatan kuvausta, kun yhteys on muodostunut. Tällöin **simulaattori ei muuta alikohdassa 2.2.3 esitettyä XML-kuvaustaan istunnon ollessa käynnissä**. Tarvittaessa yhteys voidaan katkaista simulaattoria sammuttamatta, jonka jälkeen tehdään tarvittavat muutokset ja yhteys käynnistetään uudestaan.

## 4.2 Suunnitteluperiaatteet ja käytännöt

Yksi simulaattorin oleellisemmista vaatimuksista oli laajennettavuus tulevaisuudessa. Tästä johtuen simulaattorin arkkitehtuuriin täytyy suunnitella erityisiä laajennuspisteitä [12, s.189]. Laajentamisen on tapahduttava mahdollisimman siististi, jolloin laajennuspisteiden yhteydessä voidaan sallia joitakin erikoisempia ratkaisuja, mikäli se katsotaan tarpeelliseksi laajennuskoodin selkeyttämiseksi. Esimerkki laajennuskohdasta on uusien ko-

mentojen luominen datalähtöisesti Komento-suunnittelumallia käyttäen [13, Command Pattern]. Toinen esimerkki on Strategia-suunnittelumallin mukaisten funktio-osoitti-mien käyttö parametrirakentajien valinnassa [13, Strategy Pattern].

Edellä kuvattuja tilanteita lukuun ottamatta simulaattorin toteutuksessa pyritään noudattamaan KISS-periaatetta, joka voidaan tulkita esimerkiksi sanoiksi *Keep It Short and Simple* [18]. Kyseinen periaate olisi hyvä pitää mielessä kaikessa ohjelmistotyössä, erityisesti koska järjestelmän jatkokehittäjät eivät välttämättä ole samoja henkilöitä kuin alkuperäiset toteuttajat. KISS-periaatteen noudattaminen on hyödyllistä myös sen takia, että yksinkertainen järjestelmä on myös helpompi testata ja todentaa oikein toimivaksi.

## 4.3 Ketterät menetelmät

Yleisesti simulaattorin toteutus tullaan tekemään ketteriä ohjelmistomenetelmiä hyödyntäen [19]. Käytössä oleva protokollakirjasto on vielä kehitysasteella, ja siksi perinpohjainen suunnittelu etukäteen ei ole mahdollista. Tämän lisäksi ketterät menetelmät nopeuttavat kehitystä erityisesti kevennetyn dokumentaatiomäärän vuoksi. Suunnittelun perusteena toimii yksi toteuttajalta tullut määrittelydokumentti, joka muunnettiin samalla toimimaan myös simulaattorin käyttöohjeena. Simulaattorin osat suunnitellaan ja toteutetaan iteratiivisesti noin viikon kestävissä jaksoissa.

Simulaattori on kokonaisuutena niin pieni, ettei käyttöön kannattanut ottaa täysimittaista SCRUM-mallia [20]. Tässä projektissa tehtävien jaottelusta työlistoihin (*product backlog*, *sprint backlog*) ei olisi ollut hyötyä, sillä tehtävään tarvittava työmäärä olisi ylittänyt siitä saadun hyödyn. Oman projektiryhmän kanssa pidettiin päivittäinen palaveri (*daily standup*), joka vastaa pitkälti SCRUM-mallin *Daily Scrumia*. Asiakkaaseen oltiin yhteydessä noin viikon välein, minkä lisäksi oltiin vielä erikseen yhteydessä protokollakirjastoa tekevään työryhmään jokaviikkoisissa puhelinneuvotteluissa.

Yksi SCRUM-mallista poikkeava erikoispiirre on se, ettei alussa sovittu kaikille yhteistä valmiuden määritelmää (*Definition of Done, DoD*). Tämä toimii paremmin työlistoihin perustuvassa toimintamallissa, jossa yksittäinen tehtävä on pieni ja selkeästi eroteltu muusta järjestelmästä. Toteutuksen aikana simulaattorin osat valmistuivat lähes valmiiksi yhdellä kertaa, mutta erillistä suunnitelmaa tai lopetuskriteerejä ei muun muassa protokollakirjaston päivittymisen vuoksi voitu sopia kattavasti.

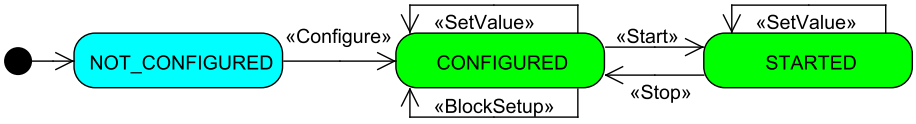
# 5. SIMULAATTORIN TOTEUTUS

Simulaattoria lähdettiin toteuttamaan luvussa 4 kuvattujen vaatimusten ja suunnitteluperusteiden pohjalta. Luvun 3 protokollakirjastosta otettiin perustoteutus, jonka lisäksi hyödynnettiin kirjaston tarjoamaa Boost-rajapintaa ja ajastimia.

Seuraavassa käydään ensin läpi simulaattorin tilamalli, jonka jälkeen perehdytään yleisarkkitehtuuriin. Tämän jälkeen käydään yksityiskohtaisesti läpi simulaattorin ytimessä toimiva käskysilmukka, siihen liittyvät parametrien käsittelijät sekä rajapinta protokollakirjastoon. Lopuksi tarkastellaan simulaattorin toteutuksessa tarvittuja aputoimintoja.

## 5.1 Tilamalli

Simulaattorille annettavia komentoja haluttiin valvoa siten, että komentojen suorittaminen väärään aikaan voidaan estää keskitetysti. Lisäksi haluttiin antaa käyttäjälle palautetta simulaattorin nykyisestä toiminnasta. Tähän tarkoitukseen simulaattorille kehitettiin tilamalli, joka on esitetty kuvassa 5.1.




Kuva 5.1: Simulaattorin sisäiset tilasiirtymät ja kussakin tilassa käytössä olevat komennot.

Tilamallissa siirrytään pääosin vasemmalta oikealle, jolloin simulaattorissa lukittujen asetusten määrä lisääntyy oikealle edetessä. Lähtötilasta siirrytään *NOT_CONFIGURED*-tilaan, joka edustaa simulaattorin puhdasta tilaa käynnistämisen jälkeen. Tilassa ei ole tarkoitus viipyä kauaa, ja ainoa sallittu komento on tilasta poistuva *Configure*. Komentoa käytetään kertomaan simulaattorille sen rooli protokollassa sekä tarkemmat tiedot yhteyden muodostamisesta protokollayhteyden toiseen päähän (IP-osoite). Kirjoitushetkellä simulaattorin vaatimuksiin on määritelty ainoastaan toiminta protokollan palvelimena (datan tuottaja), joten ainoastaan se on tuettuna komennossa.

*CONFIGURED*-tila on simulaattorin päätila datan rakenteen määrittämiseen. Tilassa voidaan antaa yksilöllisiä *BlockSetup*-komentoja, jotka luovat komennon parametreissa kuvatun blokin protokollakirjaston *Block*-luokkaa hyödyntäen. Blokin sisältämille parametreille voidaan lisäksi antaa mittausarvoja *SetValue*-komennolla. *CONFIGURED*-tilasta varsinainen simulaatio käynnistetään antamalla *Start*-komento, jolloin simulaattori siirtyy *STARTED*-tilaan.

*STARTED*-tilaan siirryttäessä luodaan protokollakirjaston rakenteet ja otetaan yhteys *Configure*-komennon määräämään kohteeseen. Kirjoitushetkellä simulaattori on määrätty toimimaan ainoastaan IP-verkossa, joten käyttöön valitaan kohdassa 3.3 esitelty *IpServerModel* ja laiteyhteyksiin sopiva *DeviceInteractionModel*. Simulaattori on toteutettu tukemaan yhteyden muodostusvaiheessa toimintaa joko TCP-palvelimena tai TCP-asiakkaana. *STARTED*-tilassa simulaattori on yhteydessä vastapuoleen ja ottaa vastaan tältä tulevia pyyntöjä datan lähettämisestä. Pyynnön saapuessa simulaattori käy läpi *CONFIGURED*-tilassa luodut blokit ja lähettää halutut mittaukset protokollayhteyden yli. Mittausten arvoja voidaan myös muuttaa *STARTED*-tilassa käyttäen *SetValue*-komentoa. Kun simulaatio halutaan keskeyttää, se voidaan tehdä *Stop*-komennolla, jonka jälkeen palataan *CONFIGURED*-tilaan, jossa blokkien muuttaminen on taas mahdollista.

Edellä kuvattujen komentojen lisäksi simulaattorilla on lukuisia muita komentoja, jotka toimivat käyttäjän tukena ja mahdollistavat joitakin protokollan ominaisuuksia tämän dokumentin piirin ulkopuolella. Tällaisia komentoja ovat *CheckState*, *DeleteBlock*, *Exit*, *PatientInfo*, *Version* ja *Wait*.

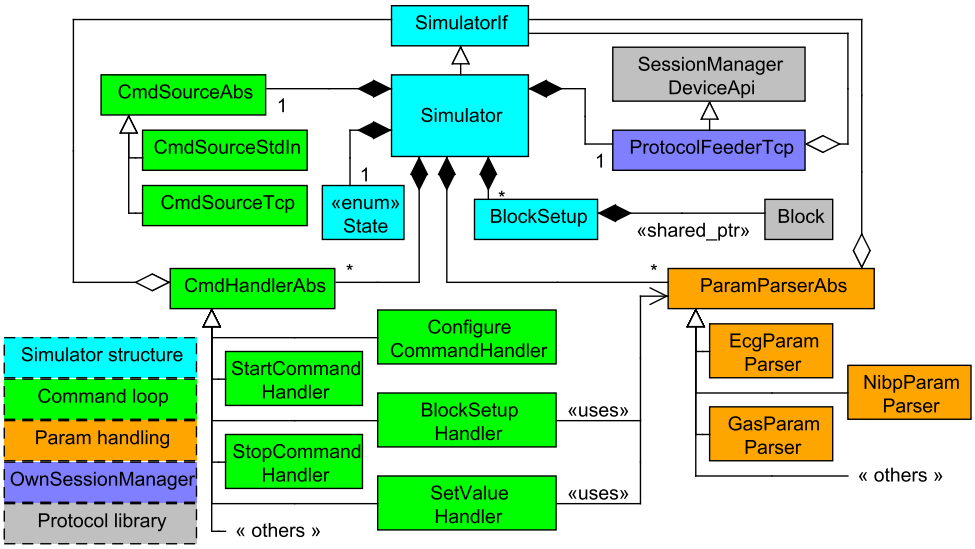
## 5.2 Yleisarkkitehtuuri

Simulaattorin yleisarkkitehtuuri jakautuu simulaattorin käyttäjän käskyjen hakemisesta ja käsittelystä huolehtivaan osaan sekä protokollakirjaston kanssa keskustelevaan osaan. Ohjelman säikeistys toimii siten, että pääsäikeessä (*main thread*) pyörii käyttäjän komentoja käsittelevä silmukka. Tämän lisäksi simulaattorin käyttöön annetaan joukko erillisiä säikeitä käyttämällä protokollakirjaston *ThreadPool* ja *ThreadAbs*-luokkia. Lisäksi protokollakirjasto voi sisäisesti käyttää vielä joitakin ylimääräisiä säikeitä palveluiden tuottamiseen. Simulaattorin perusarkkitehtuuri on esitetty kuvassa 5.2.

Simulaattorin ydin on *SimulatorIf*-rajapinnasta peritty *Simulator*-luokka, jonka tarkoitus on yhdistää kaikki simulaattorin sisäiset kokonaisuudet yhdeksi toimivaksi paketiksi. Tämän lisäksi *Simulator*-luokka pitää kirjaa simulaattorin tämänhetkisestä tilasta *State*-enumeraatiota hyväksi käyttäen. Luokka tarjoaa myös parametrinkäsittelypuolelle paikan protokollan blokkien tallentamiseen (*BlockSetup*) ennen kuin ne lähetetään eteenpäin protokollakirjaston yli.

Kuvaan on yhdeksi kokonaisuudeksi merkitty käskysilmukka (*Command loop*), joka huolehtii käskyjen käsittelystä käskylähteiden (*CmdSource*) ja käskykäsittelijöiden (*CmdHandler*) avulla. Käskylähteet tarjoavat käyttäjälle mahdollisuuden ohjeistaa simulaattorin toimintaa useita eri reittejä, esimerkiksi oletussyöttövirtaa tai TCP-yhteyttä, käyttäen. Käskykäsittelijät vastaavat kukin yhden käskyn käsittelystä, ja niille annetaan osoitin *SimulatorIf*-rajapintaan monimutkaisemman toiminnan mahdollistamiseksi.

Toisena kokonaisuutena kuvassa erottuu parametrien käsittely (*Param handling*), joka toimii *BlockSetupHandlerin* ja *SetValueHandlerin* tukena käskyjen käsittelyssä. Pääosa toiminnallisuudesta löytyy *ParamParserAbs*-luokasta, joka sisältää erikseen funktiot




Kuva 5.2: Simulaattorin yleisarkkitehtuuri jaoteltuna simulaattorin rakenneosiin, käskysilmukkaan, parametrien käsittelyyn, omaan istuntojen hallitsijaan sekä protokollakirjaston luokkiin.

parametrien lisäämiselle blokkiin sekä arvojen asettamiselle blokin sisällä. Varsinaisten käsittelyluokkien tehtäväksi jää määritellä, miten kukin eri niminen parametri tulee protokollalle alustaa. Tässä käytetään apuna *Marshalling*-kerroksen dataluokkia.

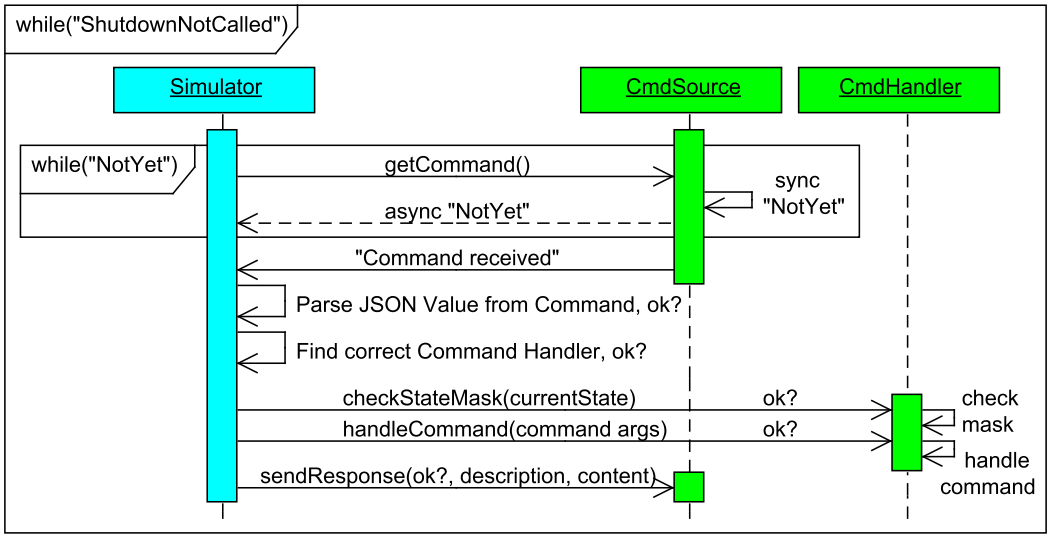
Viimeinen kokonaisuus on simulaattoria varten tehty *ProtocolFeederTcp*, joka laajentaa *SessionManagerDeviceApi*-luokkaa käyttäen *SimulatorIf* -rajapinnan kautta saatuja blokkikuvauksia. *ProtocolFeederTcp* on suunniteltu toimimaan ainoastaan TCP-verkossa tapahtuvan protokollaliikenteen datan lähettäjänä (palvelin), jolloin datan vastaanottopään funktioille on jätetty tyhjä toteutus. *ProtocolFeederTcp* huolehtii sisäisesti myös lähetettävän datan ajastuksen hallinnasta, missä se käyttää apunaan protokollakirjaston *Core*-kerroksesta löytyviä ajastinluokkia.

## 5.3 Käskysilmukka

Käskysilmukka toimii simulaattorin käyttäjän ja protokollakirjaston välissä ohjaten komentoja ja vastauksia osapuolelta toiselle. Seuraavassa käydään ensin läpi käskysilmukan yleinen toiminta, jonka jälkeen silmukan osat kuvataan yksityiskohtaisesti. Lopuksi tutkitaan, mitä käskysilmukan laajentaminen tarkoittaa uusien käskylähteiden tai simulaattorille annettavien käskyjen kannalta.

### 5.3.1 Toiminta

Käskysilmukka toimii simulaattorin pääsäikeessä, joka on varattu sen käyttöön kokonaisuudessaan. Tämä mahdollistaa silmukan toteuttamisen siten, että käyttäjä voi vapaasti valita, kysytäänkö käskyjä simulaattorin käyttäjältä synkronisesti vai asynkronisesti. Alustuksen jälkeen käynnistyvän käskysilmukan sekvenssikaavio on esitetty kuvassa 5.3.




Kuva 5.3: Sekvenssikaavio ohjelman pääsäikeessä toimivasta käskysilmukasta.

Käskysilmukan uloin osa vastaa siitä, että uusia komentoja tullaan hakemaan niin kauan kuin tiettyä simulaattorin sulkevaa käskyä ei ole annettu. Lopetuskäsky voidaan antaa *Exit*-komennolla, tai se voi tulla esimerkiksi *CmdSourceTcp*-luokalta komentoja välittävän TCP-yhteyden katketessa. Käytännössä simulaattorin sammutukseen käytetty funktio löytyy *SimulatorIf*-rajapinnasta, jolloin kuka tahansa tämän rajapinnan käyttäjä voi halutessaan sammuttaa simulaattorin hallitusti.

Yksittäisen käskyn käsittely lähtee liikkeelle sen hakemisesta käskylähteeltä, joka on simulaattorin luomisen yhteydessä valittu *CmdSourceAbs*-rajapinnan toteuttava luokka. Käskylähteeltä pyydetään komentoa *getCommand*()-funktiolla, joka palauttaa paluuarvonaan tiedon siitä, tuliko käsky kokonaisuudessaan perille. Käskylähde voi joko palata välittömästi, mikäli uutta käskyä ei ole tullut (asynkroninen toiminta), tai se voi jäädä odottamaan *getCommand*()-kutsun sisälle kokonaista käskyä (synkroninen toiminta). Kun käsky lopulta tulee, sisempi silmukka päättyy ja siirrytään käskyn jäsentämiseen ja käsittelyyn.

Käsky tulee JSON-muodossa tallennettuna merkkijonona, joten ensimmäinen vaihe on komennon jäsentäminen JsonCpp-kirjastoa käyttäen. Tämän jälkeen, jos käskyn formaatti oli kunnossa, haetaan käskyn nimen perusteella talletetuista komentokäsittelijöistä käskyä vastaava käsittelijä. Mikäli käsittelijä löytyy, siltä kysytään, voiko käskyn suorittaa simulaattorin nykyisessä tilassa kutsumalla *checkStateMask*()-funktiota. Edelleen, jos käsky on luvallista suorittaa tässä tilassa, komentosilmukassa kutsutaan varsinaista *handleCommand*()-funktiota. Tämä käynnistää komentokäsittelijän varsinaisen toiminnallisuuden, joka komennosta riippuen saattaa pyytää *SimulatorIf*-rajapinnan kautta esimerkiksi blokkilistaa tai simulaattorin sammuttamista.

Kun käsittelyfunktiosta palataan, paluuarvo kuvaa, onnistuiko käskyn käsittely vai ei. Virhetilanteessa käskykäsittelijältä kysytään tässä vaiheessa tarkemmat tiedot virheen tyypistä, minkä jälkeen vastaus lähetetään *CmdSourceAbs*-rajapinnan kautta takaisin käyttä-

jälle. Huomattavaa on, että myös jäsentämisen aikaisemmissa vaiheissa käskyn keskeyttäneistä virheistä kerrotaan käyttäjälle samaa reittiä pitkin.

### 5.3.2 Käskylähteet

Käskykäsittelijöiden alustaminen tehdään *Simulator*-luokan rakentajissa siten, että alustettava käskylähde käy ilmi rakentajalle annettavista parametreista. Esimerkiksi jos rakentajalle annetaan IP-osoite ja portti, se ymmärtää alustaa käskylähteeksi *CmdSourceTcp*-luokan. Käskylähteen rajapintaluokka (*CmdSourceAbs*) on kokonaisuudessaan hyvin yksinkertainen. Luokan otsikkotiedoston oleelliset osat on esitetty listauksessa 5.1.

```
1  class CmdSourceAbs
2  {
3  public:
4    CmdSourceAbs();
5    virtual ~CmdSourceAbs();
6    bool sourceReady();
7    virtual void notifyShutdown();
8    virtual bool getCommand(std::string& rRequestP)=0;
9    virtual void sendResponse(const std::string& pResponseP)=0;
10 };
```

Listaus 5.1: *CmdSourceAbs*-luokan julkinen rajapinta.

Rajapintaluokan alustusfunktioissa ei ole mitään erikoista, mutta huomattavaa on, että kopiorakentaja ja sijoitusoperaattori on poistettu julkisesta rajapinnasta. Aiemmin käytiin läpi *getCommand*()- ja *sendResponse*()-funktioiden toiminta, joten tässä esitellään pelkästään uudet funktiot ennen itse käskylähteisiin perehtymistä. Funktiota *sourceReady*() kutsutaan ennen käskysilmukkaan siirtymistä tarkistamaan, onko käskylähteen alustaminen onnistunut. Funktiolla *notifyShutdown*() kerrotaan käskykäsittelijälle simulaattorin sammutusaikeesta, jolloin käsittelijä osaa sulkea tarvittavat rakenteet. Huomioitavaa on, että tämän on oltava erillisessä funktiossa purkajan sijaan, koska nyt voidaan esimerkiksi sulkea yhteyksiä ilman käskykäsittelijän välitöntä tuhoamista.

Kirjoitushetkellä simulaattorille on määritelty vain kaksi käskylähdettä: oletussyöttövirta ja TCP-yhteys. Näistä oletussyöttövirtaan perustuva käskylähde (*CmdSourceStdIn*) on yksinkertaisempi ja käydään sen takia läpi ensin. *CmdSourceStdIn* perustuu puhtaasti C++-perusrakenteisiin, ja se hyödyntää käskylähteiden mahdollisuutta toimia synkronisesti. Käskyt kysytään yksi kerrallaan *std::getline*()-funktiolla, jolloin *getCommand*-funktiosta palataan vasta, kun koko käsky on saatu haettua. Samoin *sendResponse*() toimii yksinkertaisesti kirjoittamalla vastaus oletustulostusvirtaan (*stdout*). Huomattavaa on, että *CmdSourceStdIn* mahdollistaa käskyjen antamisen tiedostoista unix-pohjaisista käyttöjärjestelmistä tutulla syöteoperaattorilla (*Simulator.exe < inputfile.txt*).

TCP-käskykäsittelijä (*CmdSourceTcp*) on toteutukseltaan monimutkaisempi, ja se tukeutuu Boost-kirjaston *asio*-alikirjastoon asynkronisten kirjoitus- ja lukuoperaatioiden to-

teuttamisessa. *CmdSourceTcp*-luokalle annetaan rakentajassa parametreina IP-osoite sekä portti, joita se käyttää uuden yhteyden kuuntelusoketin luomisessa. Näin simulaattori voidaan rajoittaa kuuntelemaan tulevia komentoyhteyksiä vain tietystä verkkorajapinnasta. Rakentajalle annetaan myös osoitin *SimulatorIf*-rajapintaan, jotta se osaa sulkea simulaattorin, jos TCP-yhteys jostain syystä katkeaa. Uuden käskyn lukuvaiheessa käsky haetaan *asio*-kirjaston *read_until*()-funktiolla, jolla TCP-soketista luetaan merkkejä rivin loppumerkkeihin ($\backslash r \backslash n$) asti. Vastauksen palautusvaiheessa vastausmerkkijono kirjoitetaan samaan TCP-sokettiin *asio*-kirjaston *write*()-funktiolla.

### 5.3.3 Käskykäsittelijät

Käskykäsittelijät liitetään osaksi simulaattoria Komento-suunnittelumallia mukailevalla tavalla. Käskylähteet alustetaan *Simulator*-luokan rakentajasta kutsutussa *populateHandlersAndParsers*()-funktiossa, joka huolehtii myös parametrijäsentäjien alustamisesta. Käskykäsittelijät talletetaan käskyn nimen perusteella STL:n *map*-tietorakenteeseen, ja *CmdHandlerAbs*-luokka on rakennettu muutenkin siten, että kaikki oleellinen käskyn käyttämisestä ja toiminnallisuudesta löytyy *CmdHandlerAbs*-luokasta perityn varsinaisen käsittelijäluokan sisältä. Luokan otsikkotiedoston oleelliset osat on esitetty listauksessa 5.2. [13, Command Pattern]

*CmdHandlerAbs* on abstrakti kantaluokka, josta on tarkoitus periyttää kunkin komennon käsittelevä toteutus erikseen omaan luokkaansa. Periyttämisen yhteydessä *CmdHandlerAbs*-rakentajalle kerrotaan ne tilat, joissa komento on luvallista suorittaa. Tämä koostuu kuvassa 5.2 näkyvästä *State*-enumeraation bittitason TAI-operaattorilla yhdistetyistä arvoista. Julkisesta rajapinnasta löytyvä *checkStateMask*()-funktio vertaa annettua tilalippua talletettuun arvoon ja kertoo, onko komennon suoritus luvallista. Alustamisen yhteydessä käskykäsittelijälle annetaan myös osoitin simulaattorin *setSimulator*()-funktiolla. Tämä tehdään erillisellä funktiolla rakentajan sijaan, jotta simulaattoriosoitinta ei tarvitsisi viedä kaikkien perittyjen käskykäsittelijöiden läpi erikseen.

Varsinainen käskysana ja samalla käskyn nimi palautetaan *getName*()-funktiolla, jolloin koko käskytoteutus pysyy yhden luokan sisällä. Samalla tavoin *handleCommand*()-funktio kutsuu *protected*-rajapinnan *handle*()-funktiota, kun on aika suorittaa annettu komento. Kultakin komennolta odotetaan jonkinlaista vastausta, mikäli komennon suorituksessa meni jokin pieleen. Tätä kuvataan *handleCommand*()- ja *handle*()-funktioiden *bool*-paluuarvolla. Vikatilanteen kuvaus ja varsinainen virheviesti voidaan kysyä JSON-arvona, joka muunnetaan merkkijonoksi simulaattorin puolella käskysilmukassa ennen lähettämistä takaisin käyttäjälle osana vastausviestiä.

```
1  class CmdHandlerAbs
2  {
3  public:
4    explicit CmdHandlerAbs(int simStateMaskP);
5    void setSimulator(SimulatorIf* pSimulatorP);
6    virtual ~CmdHandlerAbs();
7    virtual const std::string& getName()=0;
8    bool checkStateMask(State stateP);
9    bool handleCommand(const Json::Value& pRequestParamsP);
10   virtual Json::Value getResponseDescription();
11   virtual Json::Value getResponseContent();
12 protected:
13   virtual bool handle(const Json::Value& pRequestParamsP)=0;
14 };
```

Listaus 5.2: *CmdHandlerAbs*-luokan julkinen ja periytyvä rajapinta.

Kaikkien simulaattorin tukemien käskyjen läpikäynti on tämän dokumentin piirin ulkopuolella. Oleellista on kuitenkin tietää, millä tavoin *BlockSetupHandler* ja *SetValueHandler* kertovat simulaattorille muuttuneista mittausarvoista. Käytännössä kumpikin käsittelijä pyytää *SimulatorIf*-rajapinnan kautta simulaattorilta referenssin listaan olemassa olevista *BlockSetup*-rakenteista, jotka pitävät sisällään osoittimen varsinaiseen protokollakirjaston *Block*-luokkaan. Tämän jälkeen kumpikin käskykäsittelijä voi käskyn parametrien mukaisesti halutessaan joko lisätä listaan uusia blokkeja tai muuttaa listassa olevien blokkien arvoja. Tämän mekanismin tarkempi kuvaus jätetään kohtaan 5.4 Parametrien käsittely.

### 5.3.4 Laajentaminen

Käskysilmukan laajentaminen voi tarkoittaa sen laajentamista joko käskylähteitä tai käskykäsittelijöitä lisäämällä. Oleellista simulaattorin laajennettavuuden kannalta on, että näitä voidaan lisätä toisistaan riippumatta ja samalla mahdollisimman siistillä tavalla. Käskylähteiden laajentaminen koostuu uuden luokan periyttämisestä *CmdSourceAbs*-kantaluokasta sekä uuden rakentajan kirjoittamisesta *Simulator*-luokalle. Uutta käsittelijää toteutettaessa *getCommand*() kirjoitetaan toimimaan synkronista tai asynkronista tapaa käyttäen riippuen toteutettavan käskylähteen ominaispiirteistä. Esimerkiksi oletussyöttövirtaa käyttävä käskylähde (*CmdSourceStdIn*) toimii luonnostaan synkronisesti rivi kerrallaan, sillä virrasta ei edes pysty järkevästi lukemaan tietoa merkki kerrallaan. Toteutus *sendResponse*()-funktiolle on usein suoraviivaisempi, sillä vastausta lähetettäessä tiedetään aina, kuinka paljon dataa ollaan siirtämässä.

Uuden komennon lisääminen järjestelmään on kapseloitu lähes täysin *CmdHandlerAbs*-luokasta periytettävän luokan toteutukseen aiemmin kuvatulla tavalla. Ainoa toimenpide on käskykäsittelijän lisääminen listauksessa 5.3 kuvattua apufunktiota käyttäen.

```
1  void Simulator::addCommandHandler(CmdHandlerAbs* pHandlerP)
2  {
3    if(cmdHandlersM.find(pHandlerP->getName())!=cmdHandlersM.end())
4    {
5      printf(constants::WARNING_DUPLICATE_COMMAND_HANDLER.c_str(),
6        pHandlerP->getName().c_str());
7    }
8    cmdHandlersM[pHandlerP->getName()]=pHandlerP;
9    pHandlerP->setSimulator(this);
10 }
```

Listaus 5.3: Uuden käskykäsittelijän lisäyksen apufunktio.

Apufunktio tarkistaa, onko järjestelmään jo lisätty samanniminen komento ja lisää uuden käskyn mikäli näin ei ole. Lisäksi apufunktio antaa käskykäsittelijälle osoittimen simulaattoriin *setSimulator*()-funktiota käyttäen.

## 5.4 Parametrien käsittely

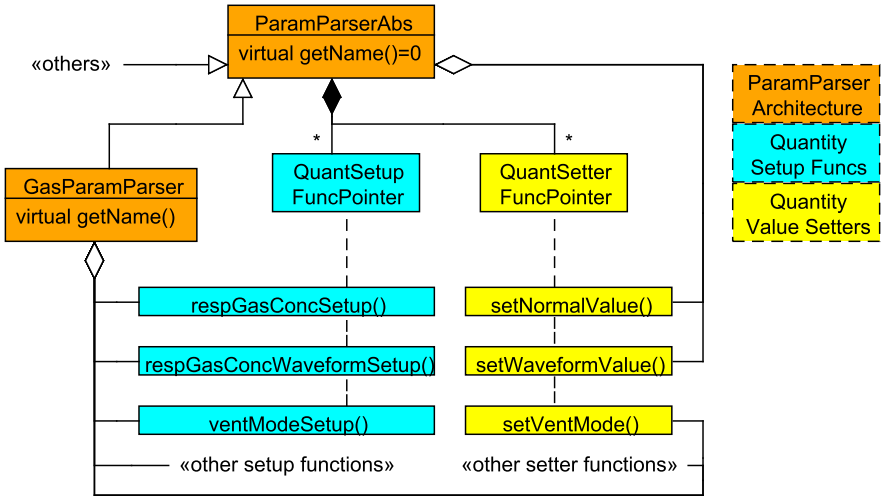
Parametrien käsittely hoidetaan simulaattorissa *ParamParserAbs*-luokan ja sen aliluokkien avulla. Käsittelyn toteutuksessa on otettu viitteitä Strategia-suunnittelumallista, jossa toteutusta voidaan vaihtaa ohjelman suorituksen aikana käsiteltävän parametrin perusteella [13, Strategy Pattern]. Järjestelmässä suunnittelumalli on toteutettu funktio-osoittimien avulla ylimääräisen työn välttämiseksi laajennusvaiheessa.

Seuraavassa käydään ensin läpi parametrikäsittelijöiden yleinen toiminta, jonka jälkeen paneudutaan tarkemmin blokkien alustuksessa ja mittausarvojen asettamisessa käytettyihin tietorakenteisiin. Lopuksi tutkitaan, miten järjestelmään voidaan lisätä uusi parametrikäsittelijä.

### 5.4.1 Toiminta

Simulaattorin laajennettavuuden takia parametrien käsittelyyn täytyi rakentaa jokin helposti uutta toiminnallisuutta mahdollistava toteutus. Uuden parametrin lisääminen koostuu parametrikäsittelijän toteuttamisesta sekä yksittäisten mittausten lisäämisestä assosiatiiviseen tauluun oikean luontifunktion löytämiseksi. Tähän käytettiin Strategia-suunnittelumallin muunnosta, jossa tietyn parametrin toiminnallisuus löytyy *std::map*-rakenteesta. Rakenteessa mittauksen nimi merkkijonona kuvautuu haluttuun toiminnallisuuteen funktio-osoittimena. Yleiskuva parametrikäsittelijöiden rakenteesta löytyy kuvassa 5.4.

Parametrikäsittelyn ydin on *ParamParserAbs*-luokka, joka sisältää pääosan käsittelyssä tarvittavasta toiminnallisuudesta. Luokka toteuttaa logiikan siinä määrin, että ainoa tehtävä varsinaisille perityille käsittelijöille (*GasParamParser*) on kertoa käsiteltävän parametrin nimi *getName*()-funktiolla ja määrittää *ParamParserAbs*-luokan tietorakenteisiin




Kuva 5.4: Luokkakaavio parametrinkäsittelijöiden käyttämistä funktio-osoittimista.

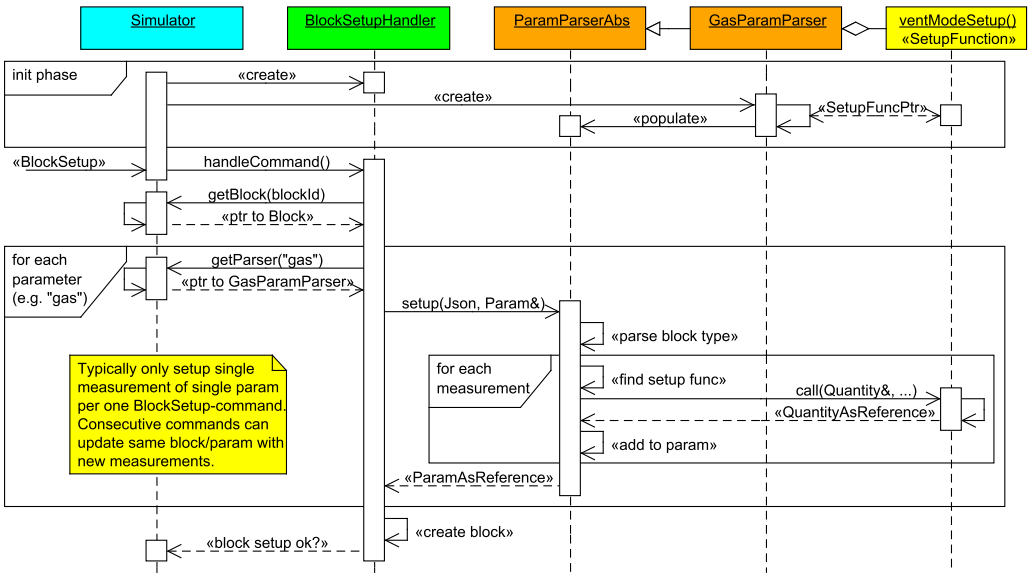
mittausten kuvautuminen. Periytetyissä luokissa asetetaan ainoastaan yksittäisiin mittauksiin (*Quantity*) liittyviä käsittelijöitä, kokonaisiin parametreihin liittyvä toiminnallisuus on kokonaisuudessaan *ParamParserAbs*-luokan sisällä.

Mittauksia käsittelevä järjestelmä koostuu kahdenlaisista funktio-osoittimista. *QuantSetupFuncPointer* huolehtii uuden mittauksen luomisesta JSON-muodossa annettuja parametreja ja mittauksen nimeä hyväksi käyttäen. Mittausten luomisessa tarvittavat toimenpiteet ovat hyvin mittauskohtaisia, joten kaikki *QuantSetupFuncPointer*-osoittimen mukaiset funktiot on kirjoitushetkellä toteutettu *ParamParserAbs*-luokasta perityissä luokissa.

Toinen osa järjestelmää ovat mittausarvojen asettamisessa käytetyt *QuantitySetterFuncPointer*-osoittimet. Protokollakirjaston *Quantity*-luokan monikäyttöisyydestä johtuen arvojen asetus on pääosalle mittauksista hyvin samantyyppinen. Tämän vuoksi *ParamParserAbs*-luokka tarjoaa toteutettavien parametrikäsittelijöiden käyttöön *setNormalValue*()- ja *setWaveformValue*()-funktiot, jotka kattavat lähes kaikki vastaan tulevat tapaukset.

### 5.4.2 Blokkien alustus

Blokkien alustus on parametrien käsittelyyn liittyvistä tapauksista monimutkaisempi, ja lisäksi se tapahtuu ajallisesti ennen mittausarvojen asettamista. Tästä johtuen seuraavassa käydään läpi blokin alustus kokonaisuutena, parametrikäsittelijöiden alustuksesta alkaen *BlockSetup*-komennon suorittamiseen asti. Kuvattava tapaus koskee ensimmäisten mittausten lisäämistä uuteen blokkiin, ja kuvauksessa on tarkoituksella jätetty pois mittausten lisääminen jo olemassa olevaan blokkiin. Vanhaan blokkiin lisääminen tapahtuu kopioimalla vanhasta blokista uuteen ne mittaukset, joita uusi blokki ei itse määrittänyt. Toiminnallisuuden tarkempi läpikäynti on kuitenkin tämän dokumentin piirin ulkopuolella. Sekvenssikaavio uuden blokin alustamisesta on esitetty kuvassa 5.5.




Kuva 5.5: Kuvaus uuden blokin luontisekvenssistä funktio-osoittimia käyttäen. Olemassa olevan blokin täydentämiseen liittyvä toiminnallisuutta ei ole kuvattu.

Parametrien käsittely lähtee liikkeelle simulaattorin käynnistysvaiheessa, jolloin *Simulator*-luokka luo itselleen tarvittavat käskykäsittelijät (*BlockSetupHandler*) sekä parametrikäsittelijät (*GasParamParser*). *GasParamParser* alustaa rakentajassaan kaasuparametriin liittyvät mittausten rakentajat *QuantSetupFuncPointer*-osoitinta hyväksi käyttäen. Tämän jälkeen simulaattorin käyttäjä antaa *BlockSetup*-komennon, josta käskysilmukka jäsentää parametrit ja välittää ne edelleen *BlockSetupHandler*-luokalle. *BlockSetupHandler* pyytää parametrina tulevan blokkitunnisteen (*blockId*) perusteella *SimulatorIf*-rajapinnan kautta osoittimen *BlockSetup*-tietorakenteeseen. Huomattavaa on, että tämä blokkitunniste on eri asia kuin protokollan kuvauksessa kohdassa 2.2.3 mainittu *KuvausID*, joka kuvaa blokin sisällön protokollalle. Tämä tunniste on käyttäjän itsensä määräämä, ja sen tarkoitus on erottaa eri blokit toisistaan simulaattorin käyttäjälle.

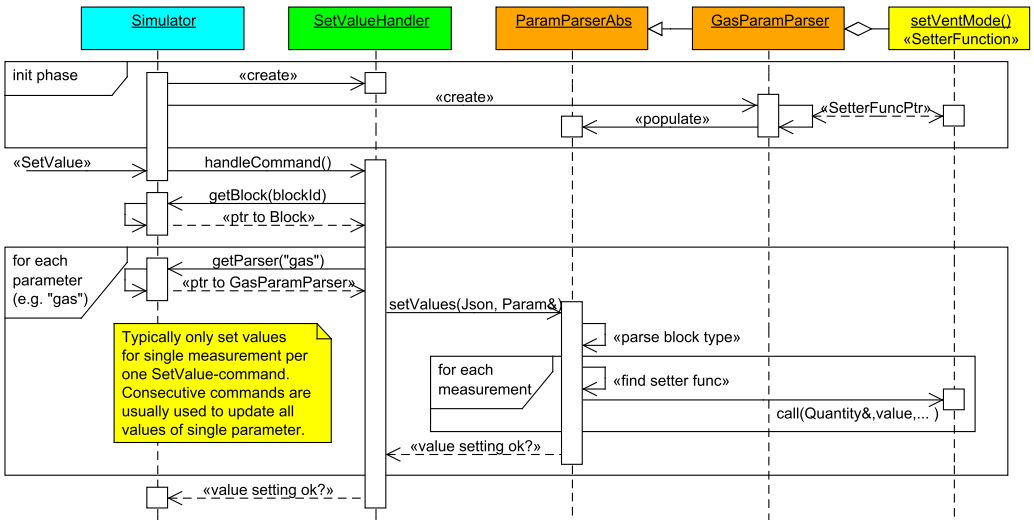
Seuraavaksi alkaa itse parametrien alustus simulaattorilta saadun *BlockSetup*-rakenteen sisältämän *shared_ptr*-osoittimen päähän. Simulaattori tukee useampien mittausten ja parametrien lisäämistä yhdellä *BlockSetup*-käskyllä, mutta yleisempi käyttötapaus on kunkin mittauksen lisääminen omalla komennollaan. Näin simulaattorille annetut komennot pysyvät siistimpänä, mutta vastapainona joudutaan toteuttamaan myös toiminnallisuus jo olemassa olevaan blokkiin lisäämiselle. Toiminnallisuudet eivät ole toisiaan poissulkevia, joten tässä käydään läpi usean parametrin ja mittauksen lisääminen samassa komennossa. Käytännössä *BlockSetupHandler*-käskykäsittelijä käy sille annettuja parametreja läpi ja alustaa kunkin löytämänsä parametrin *SimulatorIf*-rajapinnan läpi hakemansa parametrikäsittelijän avulla.

Saatuaan parametrikäsittelijän *BlockSetupHandler* kutsuu käsittelijän *setup*()-funktiota antaen sille parametriksi JSON-muodossa olevat funktioparametrinsa sekä viitteen protokollakirjaston *Parameter*-luokan olioon. Ajatuksena on, että *ParamParserAbs* luo saa-

miensa JSON-parametrien perusteella joukon mittauksia (*Quantity*) ja palauttaa näistä kootun parametrin takaisin käskykäsittelijälle viiteparametrissa. Funktio *setup*() tutkii ensin, vastaavatko saatujen mittausten blokkityypit (*num/wav/cfg*) toisiaan. Tämän jälkeen *setup*() etsii sille talletettujen *QuantSetupFuncPointer*-osoittimien joukosta tietyn mittauksen nimeä vastaavan funktion ja antaa sen alustaa uuden mittauksen protokollakirjaston *Quantity*-luokan (tai siitä perityn luokan) avulla. Mittauksen saatuaan *setup*() lisää sen parametriin ja alkaa käsittelemään seuraavaa mittausta, mikäli niitä on vielä jäljellä. Kun kaikki mittaukset on jäsennetty, suoritus palautuu *BlockSetupHandler*-luokkaan, jossa luotu parametri lisätään blokkiin. Huomattavaa on, että virhe missä tahansa edellisen ketjun kohdassa johtaa koko *BlockSetup*-komennon keskeytymiseen ja virheilmoituksen lähettämiseen simulaattorin käyttäjälle.

### 5.4.3 Mittausarvojen asettaminen

Mittaustietojen asettaminen on toimenpiteenä hyvin samankaltainen kuin blokkien alustus. Suurimpana erona jotkin vaiheista ovat yksinkertaisempia, sillä nyt kaikki käytetyt tietorakenteet ovat jo olemassa. Sekvenssikaavio mittausarvon asettamisesta on esitetty kuvassa 5.6.




Kuva 5.6: Kuvaus mittausarvojen asettamisesta funktio-osoittimia käyttäen. Tämä voi tapahtua myös simulaation ollessa käynnissä.

Mittausarvojen asettaminen lähtee liikkeelle blokkien alustuksen tapaan simulaattorin käynnistysvaiheessa, jolloin *Simulator*-luokka luo itselleen tarvittavat käskykäsittelijät (kuvassa *SetValueHandler*) ja parametrikäsittelijät (kuvassa *GasParamParser*). *GasParamParser* alustaa rakentajassaan kaasuparametriin liittyvät mittausarvojen asettajat *QuantSetterFuncPointer*-osoitinta hyväksi käyttäen. Simulaattorin käyttäjän antaessa *SetValue*-komennon käskysilmukka pyytää taas komennon mukana tulleen *blockId*-arvon perusteella oikean blokin *SimulatorIf* -rajapinnalta.

Seuraavaksi *SetValueHandler* etsii oikeaa parametria vastaavan käsittelijän (esimerkiksi "gas"), jonka jälkeen se kutsuu *setValues*()-funktiota antaen kohdeparametrin referenssinä arvojen kirjoittamista varten. Funktioparametrina annetaan myös uudet mittausarvot JsonCpp-kirjastosta tulevaan *Json::Value*-arvoon talletettuna. Myös mittausarvojen asettaminen voidaan tehdä blokkien alustuksen tapaan useampi mittausarvo kerrallaan. Tässäkin tapauksessa komennot annetaan kuitenkin yleensä yksitellen, jotta ne pysyvät yksinkertaisina.

Käytännössä *SetValueHandler*-käskykäsittelijä käy sille annettuja parametreja läpi ja tallentaa kuhunkin mittaukseen sen uuden arvon *SimulatorIf*-rajapinnan läpi haetun parametrikäsittelijän määräämien funktioiden avulla. Funktion *setValue*() toteutus tarkistaa myös, että saatujen mittausten blokkityypit (*num/wav/cfg*) vastaavat aiemmin *BlockSetup*-komen-nolla annettuja. Arvojen asettamisessa nousee esille kaksi päätapausta, jotka vastaavat *ParamParserAbs*-luokan tarjoamia *setNormalValue*()- ja *setWaveformValue*()-funktioita. Erona näissä on se, että numeeristen mittausten ja konfiguraatioiden osalta asetetaan vain yksi arvo, kun taas aaltomuotoisten käyrien tapauksessa yhdellä komennolla asetetaan koko mittausperiodin arvot kerralla. Lisäksi huomattavaa on, että tässäkin virhe missä tahansa edellisen ketjun kohdassa johtaa koko *SetValue*-komennon keskeytymiseen ja virheilmoituksen lähettämiseen simulaattorin käyttäjälle.

### 5.4.4 Laajentaminen

Protokolla tukee useita eri mittaustyyppejä, ja lisäksi kussakin mittauksessa saattaa olla joitakin mittaukselle ominaisia lisäparametreja. Esimerkiksi hengityskaasuja tutkittaessa on oleellista tietää, mitataanko hapen ($O_2$) vai hiilidioksidin ($CO_2$) konsentraatiota. Tätä tarkoitusta varten protokolla erottaa samat mittaukset eri kaasuille pistenotaatiolla ("fractionalInspiredConc.CO2"), missä mahdolliset arvot on esitetty protokollakirjaston tapaan kapseloidun enumeraation merkkijonoarvoina.

Uuden parametrin lisääminen simulaattorin hallittavaksi vaatii kokonaisuutena uuden parametrikäsittelijän periyttämistä *ParamParserAbs*-luokasta ja tämän lisäämistä simulaattorin alustuslistaan. Käytännössä uutta parametrikäsittelijää periytettäessä tarvitsee kertoa ainoastaan parametrin nimi toteuttamalla *getName*()-virtuaalifunktio ja määrittää periytettävän luokan rakentajassa, miten mittausten nimet kuvautuvat eri käsittelijäfunktioihin. Blokkien alustus ja mittausten lisääminen tehdään tässä tapauksessa hyvin samankaltaisilla metodeilla, joten tässä käydään läpi vain blokkien alustukseen tehdyn apufunktion toteutus. Koska toistuvien arvojen lisääminen on työlästä ja virhealtista, tehtävässä kannattaa käyttää *ParamParserAbs:n* tarjoamaa apufunktiota, joka on esitetty listauksessa 5.4.

```
1  //Add quantity parsers for given BlockType with all possible
2  //enum values appended to the name string
3  template<class EnumType, class EnumType2>
4  void ParamParserAbs::addQuantitySetupParser(BlockType quantTypeP,
       const std::string& rNameP, QuantSetupFuncPtr pParserFunctionP)
5  {
6    for(int i=0; i<=(int)EnumType::UNSPECIFIED;++i)
7    {
8      std::string enumName = EnumType::toString((EnumType::Type)i);
9      if(enumName.length()){enumName = "." + enumName;}
10     for(int j=0; j<=(int)EnumType2::UNSPECIFIED;++j)
11     {
12       std::string enumName2 = EnumType2::toString((EnumType2::Type)j);
13       if(enumName2.length()){enumName2= "." + enumName2;}
14       quantitySetupParsers.insert(make_pair(std::make_pair(
             quantityTypeP,name+enumName+enumName2),pParserFunctionP));
15     }
16   }
17 }
```

Listaus 5.4: *addQuantitySetupParser*()-funktion toteutus.

Apufunktion toteutus käyttää apunaan protokollakirjaston enumeraatioiden kapselointia sekä C++:n *template*-rakennetta, jolloin toistuvien, ainoastaan enumeraatioarvoiltaan muuttuvien nimikkeiden lisäys on helppoa. Mittausten alustamiseen käytetyt funktiot kuvataan mittaustyypin (*BlockType==num/wav/cfg*) ja mittauksen nimen avulla funktio-osoittimeksi, joka annetaan funktiolle muiden parametrien ohella. Tämän jälkeen nimeen lisätään *template*-parametreina annetuista enumeraatioista kaikki kombinaatiot kahdessa sisäkkäisessä silmukassa. Toteutus luottaa aiemmin kuvattuun enumeraatioiden kapselointiin eikä lisää mittauksen nimeen pistenotaation mukaista pistettä, mikäli enumeraation *toString*()-funktio palauttaa tyhjän merkkijonon.

Huomattavaa on, että toteutus hyödyntää ainakin tällä hetkellä voimassa olevaa oletusta, jonka mukaan jokaisen enumeraation viimeinen kelvollinen arvo on *UNSPECIFIED*. Tämä on vaarallinen oletus, ja siitä oltaisiin päästy kätevästi eroon mikäli enumeraatioiden kapselointi tarjoaisi aiemmin ehdotetun iteraattori-tyyppisen ratkaisun arvojen läpikäyntiin. Näin ei kuitenkaan ainakaan protokollakirjaston nykyisellä versiolla voi tehdä, joten kustakin käytetystä enumeraatiosta täytyy tarkistaa tämä oletus.

Kaikissa mittauksissa ei ole mukana äsken kuvatun kaltaista lisäparametria. Tällöin funktiolle on jotenkin voitava antaa tyhjä enumeraatio, joka ei lisää ylimääräistä tekstiä mittauksen nimeen. Protokollakirjastossa kaikki enumeraatiot ovat suunniteltu johonkin tiettyyn tarkoitukseen, joten tilanteeseen sopiva tyhjä enumeraatioluokka piti suunnitella itse. Luokka on toteutettu *ParamParserAbs*-luokan sisäluokkana (*inner class*), ja sen toteutus on kuvattu listauksessa 5.5.

```
1  //Empty enumeration type to use as default for templated typenames
2  class EmptyEnumType
3  {
4  public:
5    enum Type {UNSPECIFIED};
6    static const std::string toString(const Type){return "";}
7  };
8  typedef EmptyEnumType None;
```

Listaus 5.5: Protokollakirjaston enumeraatioiden kapselointia mukaileva tyhjä enumeraatioluokka.

Enumeraatioluokka on siis toteutettu *ParamParserAbs*-luokan esittelyn yhteydessä, ja siten se on suoraan käytettävissä periytettyjen luokkien rakentajissa. Enumeraatioluokka vastaa protokollakirjaston enumeraatioiden kapselointiperiaatteita siinä määrin kuin se on tarpeen *addQuantitySetupParser*()-funktion ehtojen täyttämiseksi. *Type*-enumeraatio pitää sisällään ainoastaan arvon *UNSPECIFIED*, joka rajaa apufunktiossa kummankin silmukan iteraatiomäärän yhteen. Tämän lisäksi enumeraatioluokan *toString*()-funktio palauttaa aina tyhjän merkkijonon, jolloin mittauksen nimeen ei lisätä ylimääräisiä pisteitä. Käytön helpottamiseksi luokka on vielä nimetty uudelleen *typedef*-määreellä nimelle *None*. Uusien mittausten käyttöesimerkki on annettu listauksessa 5.6.

```
1  addQuantitySetupParser<marshalling::RespGasEnum, None>(NUMERIC_BLOCK,
           "fractionalInspiredConc", &respiredGasConcSetup );
2  addQuantitySetupParser<marshalling::RespGasEnum, None>(NUMERIC_BLOCK,
           "endTidalConc", &respiredGasConcSetup );
```

Listaus 5.6: Esimerkki mittauksien lisäämisestä *addQuantitySetupParser*()-funktiota käyttäen. Kutsu tapahtuu *GasParamParser*-luokan rakentajassa.

Luomisen jälkeen parametrinkäsittelijä pitää vielä antaa simulaattorin käyttöön. Tämä tapahtuu hyvin samalla tavalla kuin komentokäsittelijöiden lisäys kohdassa 5.3.4. Oleellista on käsittelijän lisäyksen yksinkertaisuus, joka on saatu aikaan käyttämällä osittain samanlaista apufunktiota kuin komentokäsittelijöiden lisäyksessä. Lisäksi parametrikäsittelijöille annetaan *SimulatorIf*-osoitin erillisellä *setSimulator*()-funktiokutsulla periytysrajapinnan pitämiseksi siistinä. Apufunktion toteutus on esitetty listauksessa 5.7.

```
1  void Simulator::addParameterParser(ParamParserAbs* pParserP)
2  {
3    if(paramParsersM.find(pParserP->getName())!=paramParsersM.end())
4    {
5      printf(constants::WARNING_DUPLICATE_PARAMETER_PARSER.c_str(),
6        pParserP->getName().c_str());
7    }
8    paramParsersM[pParserP->getName()]=pParserP;
9    pParserP->setSimulator(this);
10 }
```

Listaus 5.7: Uuden parametrikäsittelijän lisäyksen apufunktio.

Funktio tarkistaa ensin, onko vastaavaa parametrijäsentäjää jo lisätty järjestelmään, ja tulostaa virheilmoituksen mikäli näin on. Tämän jälkeen parametrijäsentäjä lisätään *paramParsersM*-tietorakenteeseen, joka on *std::map*-tyyppinen kuvaus jäsennettävän parametrin nimestä itse jäsentäjään. Lopuksi jäsentäjälle asetetaan vielä osoitin *SimulatorIf*-rajapintaan, jotta jäsentäjillä on tarpeen vaatiessa suora pääsy simulaattorin yhteisiin tietorakenteisiin.

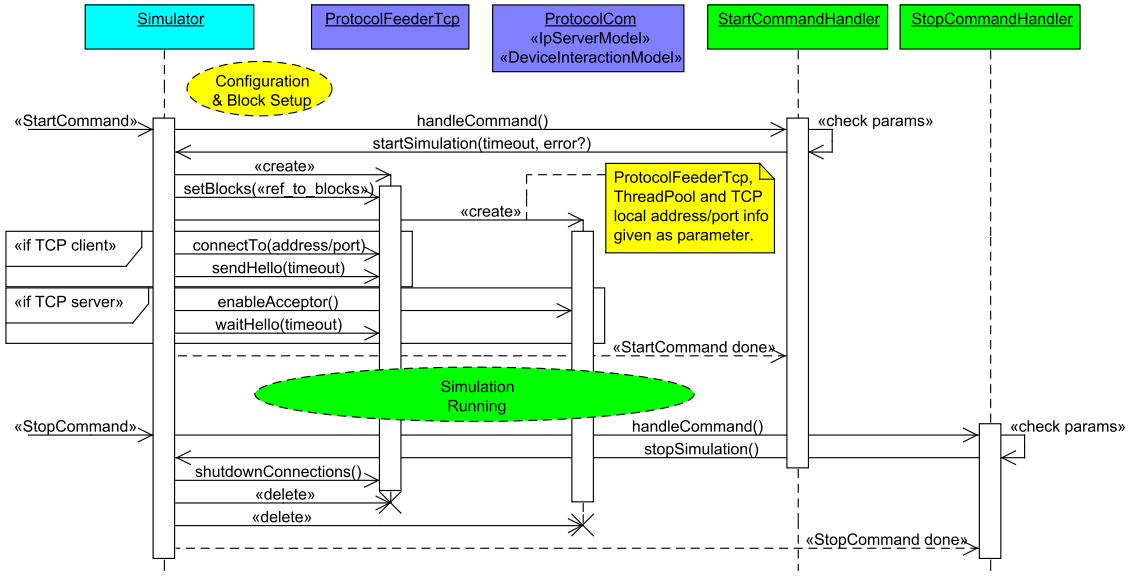
## 5.5 Kirjastorajapinta

Datan tallennukseen käytettävien dataluokkien ohella varsinainen rajapinta protokollakirjaston ja simulaattorikoodin välillä toteutetaan laajentamalla protokollakirjaston *Interaction*-kerroksen toiminnallisuutta omalla istuntojen hallitsijalla. Simulaattori toimii tässä vaiheessa ainoastaan datan tuottajana (protokollan serveri), joten tätä toiminnallisuutta tukemaan on luotu istuntojen hallitsija nimeltä *ProtocolFeederTcp*. Kuten nimikin kertoo, luokka huolehtii datan syöttämisestä protokollalle käyttäen kontrollikanavana TCP-yhteyttä. Seuraavassa käydään läpi *ProtocolFeederTcp*-luokan toiminta alustuksesta tuhoamiseen, minkä jälkeen paneudutaan vielä tarkemmin lähetettävien blokkien ajastuksen hallintaan. Lopuksi käydään läpi kirjastorajapinnan laajentamisessa vaadittavat asiat.

### 5.5.1 Toiminta

Protokollakirjaston dynaamisen toiminnan ominaisuuksista johtuen jo simulaattorin kehityksen alkuvaiheessa tehtiin periaatepäätös siitä, että protokollakirjasto instantioidaan vasta *Start*-komennon antamisen jälkeen. Luotujen blokkien hallinta helpottuu, kun simulaattorin ei tarvitse huolehtia blokkien tallentamiseen käytettävän tietorakenteen lukitsemisesta rinnakkaisuusongelmien välttämiseksi. Protokollakirjaston *Interaction*-kerroksen luokkien instantiointi sekä ohjeistus *Start* ja *Stop*-komentojen yhteydessä on esitetty kuvassa 5.7.

Kuva lähtee liikkeelle siitä oletuksesta, että simulaattorille on kerrottu *Configure*-komennolla TCP-yhteyden muodostumistapa. Kuten alikohdassa 3.3.1 kerrottiin, tämä voi olla protokollan asiakas/palvelin-jaottelusta irrallinen päätös varsinaisen TCP-yhteyden avaamisesta. Lisäksi oletuksena on, että simulaattorille on lisätty joitakin blokkeja *BlockSetup*-komennolla. Kun simulaation käynnistävä *Start*-komento tulee, simulaattori kutsuu *StartCommandHandlerin* toteutusta. Käsittelijä tarkistaa käskyn parametrit ja kutsuu edelleen *SimulatorIf*-rajapinnan kautta *startSimulation*()-funktiota, jolle annetaan parametrina suurin sallittu odotusaika, jossa yhteyden pitää olla muodostunut ja *DeviceInteractionModelin* vaatiman *Hello*-viestin vaihdettuna.




Kuva 5.7: Kuvaus protokollarajapinnan alustamisesta *Start*-komennon käsittelyn yhteydessä ja alasajosta *Stop*-komennon yhteydessä.

Varsinainen protokollakirjaston instantiointi käynnistyy *ProtocolFeederTcp*-luokan alustuksella, jonka jälkeen luokalle annetaan viitteparametrina tieto simulaattoriin lisätyistä blokeista. *ProtocolFeederTcp* periytyy protokollakirjaston *SessionManagerDeviceApi*-luokasta, jolloin se voidaan antaa *ProtocolCom* luokalle istuntojen hallitsijaksi. *ProtocolCom* on templatoitu käyttämään aiemmin kuvattuja *IpServerModel*- ja *DeviceInteractionModel*-malleja luotavien yhteyksien hallinnassa. *ProtocolCom* odottaa saavansa rakentajassaan istuntojen hallitsijan lisäksi parametrina paikallisen TCP-osoitteen ja portin, joilla kontrolloidaan mihin porttiin ja missä lähiverkossa simulaattori toimii. Luokalle annetaan lisäksi aiemmin kuvattu *ThreadPool*-luokka, joka tarjoaa joukon säikeitä protokollakirjaston sisäiseen käyttöön.

Kun protokollarajapinnan luokat on luotu, seuraava vaihe on TCP-yhteyden avaaminen. Simulaattorin rooli TCP-kontrolliyhteyden avaamisessa riippuu aiemmin annetuista *Configure*-komennon parametreista. Mikäli simulaattori on ohjeistettu toimimaan TCP-yhteydessä asiakkaana, yhteys muodostetaan kutsumalla *ProtocolCom*-luokan metodia *connectTo*(), joka muodostaa yhteyden käyttäen parametrina saatuja kohdeosoitetta ja porttia. Tämän jälkeen kutsutaan *sendHello*()-funktiota, joka aloittaa *Hello*-kättelyn ja toteaa komennon epäonnistuneeksi, mikäli kättely ei ole valmistunut parametrina annettuun aikarajaan mennessä. Mikäli taas simulaattori on ohjeistettu toimimaan TCP-palvelimena, se aloittaa sisääntulevien yhteyksien kuuntelun kutsumalla *enableAcceptor*()-funktiota. Funktio löytyy *ProtocolCom*-luokasta, ja se hyväksyy uusia yhteyksiä ainoastaan sille rakentajassa annetun IP-osoitteen ja TCP-portin kautta. Tämän jälkeen kutsutaan *waitHello*()-funktiota, joka odottaa yhteydenottoa ja kättelyn valmistumista maksimissaan annettuun takarajaan asti.

Kun *Hello*-kättely on tehty, protokollayhteys on muodostunut ja protokollan asiakas (datan vastaanottaja) voi pyytää simulaattorilta haluamansa kaltaisia blokkeja protokollan mukaisesti. Kun datan lähetys halutaan lopettaa, simulaattori voidaan sulkea *Stop*-komennolla. Alustavan parametrien tarkastuksen jälkeen *StopCommandHandler* kutsuu *SimulatorIf* -rajapinnan kautta *stopSimulation*()-funktiota. Funktio sammuttaa ensin kaikki yhteydet kutsumalla *shutdownConnections*()-funktiota, minkä jälkeen se tuhoaa järjestyksessä ensin *ProtocolFeederTcp*-instanssin ja tämän jälkeen *ProtocolCom*-instanssin.

### 5.5.2 Ajastuksen hallinta

Kun simulaatio käynnistetään blokkien alustuksen jälkeen *Start*-komennolla, simulaattori antaa *ProtocolFeederTcp*-luokalle tiedon kaikista luoduista blokeista *BlockSetup*-rakennetta apuna käyttäen. Rakenteita ei kopioida suoraan, vaan niitä välitetään simulaattorissa käyttäen Boost-kirjaston *shared_ptr*-osoitinta, jolle on annettu tässä yhteydessä nimi *BlockSetupPtr*. Osoittimia käytetään siksi, että silloin dataa ei tarvitse kopioida useaan paikkaan kun käyttäjä päivittää simulaation suorituksen aikana mittausten arvoja *SetValue*-komennolla. Käytännössä blokkilista annetaan *ProtocolFeederTcp*-luokalle *shared_ptr*-osoittimen päässä olevana *std::vector*-säiliönä, johon blokit on tallennettu *BlockSetupPtr*-osoittimia käyttäen. Säiliötyypille on annettu nimi *BlockSetupContainerPtr*, ja varsinainen *BlockSetup*-rakenteen kuvaus on esitetty listauksessa 5.8.

```
1 struct BlockSetup
2 {
3   bool enabled;//True when block is enabled and is sent if requested
4   int blockId;//Block ID, used to identify the block in Simulator
5   unsigned int formatId;//Format ID, identifies the block in Protocol
6   boost::shared_ptr<marshalling::Block> blockPtr;//Actual block
7   BlockType blockType;//Type of the block contained here (num/wac/cfg).
8   int millisecondInterval;//Block send interval in milliseconds.
9 };
```

Listaus 5.8: *BlockSetup*-rakenteen esittely.

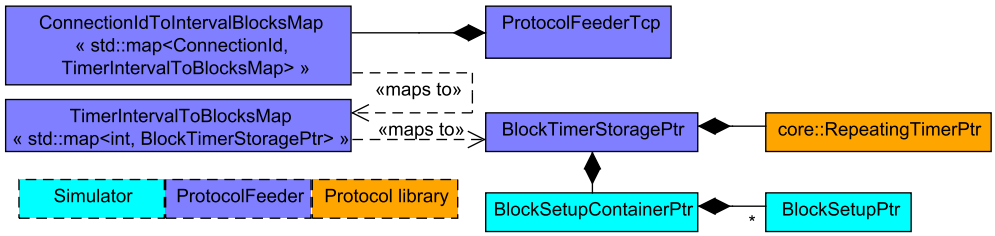
*BlockSetup* toimii pelkkänä tietovarastona ilman erillistä toiminnallisuutta, joten siitä on tehty luokan sijaan C++:n *struct*-rakenne. Ensimmäinen kenttä *enabled* on oletusarvoisesti tosi, ja kentän arvo kertoo lähetetäänkö blokin sisältämät mittaukset datan vastaanottajalle (asiakas) tämän niitä pyytäessä. Huomattavaa on, että kun asiakas pyytää simulaattorilta tiettyjä blokkeja, simulaattori lähettää blokkien kuvaukset protokollayhteyden yli riippumatta *enabled*-kentän arvosta. Tietyt mittaukset voidaan hetkellisesti keskeyttää simulaation suorituksen aikana asettamalla *enabled* epätodeksi *SetValue*-komentoa käyttäen.

Kentät *blockId* ja *formatId* ovat tiettyyn blokkiin liitettyjä tunnisteita, joilla blokki voidaan erottaa sekä simulaattorin käyttäjälle että protokollalle. Kenttä *blockId* on simulaat-

torin käyttäjän määräämä tunniste, joka säilyy muuttumattomana blokin sisällöstä riippumatta. Kenttä *formatId* on kohdassa 2.2.3 esitetty *KuvausID*, jonka arvoa kasvatetaan automaattisesti blokin sisällön muuttuessa. *KuvausID* on talletettu struktiin erillisenä sen vuoksi, että saatuaan kuvaukset simulaattorin tarjoamista blokeista asiakas pyytää yksittäisten blokkien lähetystä *KuvausID*-arvoa käyttäen.

Varsinainen blokki löytyy struktista *shared_ptr*-osoittimen päästä, jonka lisäksi struktiin talletetaan erikseen vielä blokin tyyppi (*BlockType*). Tyyppi on rakenteessa erillisenä kenttänä, koska tyypin etsiminen protokollakirjaston *Block*-luokan sisältä ei ole vaivatonta ja blokkityyppiä tarvitaan, kun asiakas pyytää simulaattorilta tiettyjen blokkien kuvauksia. Viimeisenä on kenttä *millisecondInterval*, joka vastaa blokille sen alustuksen yhteydessä määrättyä lähetystaajuutta. Jälleen kerran arvo on esitetty struktissa erillisenä kenttänä, jotta se olisi tarvittaessa helposti saatavilla.

Simulaattori tarjoaa siis *ProtocolFeederTcp*-luokalle tiedot luoduista blokeista *BlockSetup*-rakennetta käyttäen. Protokollan asiakas pyytää puolestaan ensin kaikkien simulaattorilla olevien blokkien kuvaukset XML-muodossa, minkä jälkeen se pyytää simulaattoria aloittamaan yksittäisten blokkien lähetyksen *KuvausID*-tunnisteen perusteella. Yksittäisen blokkipyynnön saapuessa *ProtocolFeederTcp* tallentaa pyynnön erilliseen tietorakenteeseen, joka on esitetty kuvassa 5.8.




Kuva 5.8: Ajastuksenhallinnan apuluokat *ProtocolFeederTcp*-luokan sisällä. Nimen sisältämä Ptr viittaa tallennustapaan *shared_ptr*-osoittimena.

Rakenne on käytännössä kaksinkertainen *std::map*, joka erottaa ensisijaisesti yhteystunnisteen perusteella, mitkä blokit on tarkoitus lähettää. Ulompi säiliö (*ConnectioIdToIntervalBlocksMap*) kuvaa yhteystunnisteen sisemmäksi säiliöksi (*TimerIntervalToBlocksMap*). Sisempi säiliö puolestaan kuvaa blokkien lähetystaajuutta vastaavan *millisecondInterval*-arvon *BlockTimerStoragePtr*-osoittimeksi. *BlockTimerStoragePtr* on *shared_ptr*-osoitin *BlockTimerStorage*-rakenteeseen, joka pitää sisällään listan tämän ajastimen hallinnoimista *BlockSetupPtr*-osoittimista sekä osoittimen protokollakirjaston tarjoamaan *RepeatingTimer*-ajastimeen. Kun asiakas pyytää alikohdassa 3.5.1 kuvatulla tavalla yksittäisen blokin lähetystä, protokollakirjasto kutsuu sille annetun istuntojen hallitsijan *notifyBlockReceiveStart*()-funktiota. *ProtocolFeederTcp*-luokan toteutus funktiolle on esitetty listauksessa 5.9.

```
1  void ProtocolFeederTcp::notifyBlockReceiveStart(
       const interaction::ConnectionId& fromId,
       const marshalling::SessionIDValue& sessionId,
       unsigned int formatId)
2  {
3    BlockSetupPtr blockSetup=getBlockSetup(formatId);
4    if(blockSetup)
5    {
6      boost::mutex::scoped_lock lock(blockTimerMutexM);
7      TimerIntervalToBlocksMap& timerMap=storedBlocksAndTimersM[fromId];
8      BlockTimerStoragePtr pStorageElement
           =timerMap[blockSetup->millisecondInterval];
9      if(!pStorageElement)
10     {//No storage element --> create it
11       pStorageElement.reset(new BlockTimerStorage());
12       timerMap[blockSetup->millisecondInterval]=pStorageElement;
13     }
14     pStorageElement->pBlockSetupContainer->push_back(blockSetup);
15     if(!pStorageElement->pRepeatingTimer)
16     {//Timer not running --> start it
17       pStorageElement->pRepeatingTimer =
             pSimulatorM->getTimerService().getRepeatingTimer(
             blockSetup->millisecondInterval);
18       pStorageElement->pRepeatingTimer->run(boost::bind(
             &ProtocolFeederTcp::sendData, this, fromId, pStorageElement));
19     }
20   }
21 }
```

Listaus 5.9: Blokkilähetyksen ajastimia hallinnoiva funktio.

Rivillä 3 *getBlockSetup*()-funktio etsii ensin simulaattorin sille asettamasta *BlockSetupContainer*-rakenteesta oikean *BlockSetup*-rakenteen *formatId*-kentän perusteella. Mikäli oikea rakenne löytyy, rivillä 6 luodaan *scoped_lock* huolehtimaan, ettei muualta kosketa *storedBlocksAndTimersM*-rakenteeseen uuden blokin lisäyksen aikana. Rivillä 7 toteutus etsii *ConnectioIdToIntervalBlocksMap*-rakenteesta yhteyttä vastaavan alikuvauksen *fromId*-yhteystunnisteen perusteella. Rivillä 8 toteutus etsii löytyneestä *TimerIntervalToBlocksMap*-kuvauksesta oikeaa millisekuntiväliä vastaavan *BlockTimerStoragePtr*-osoittimen. Mikäli jokin näistä rakenteista ei ole etsintähetkellä olemassa, *std::map*-toteutus rakentaa sille tyhjän osoittimen tulevaa käyttöä varten. Tämän jälkeen, mikäli tälle yhteydelle ei ollut vielä tämän aikavälistä *BlockTimerStorage*-rakennetta olemassa, riviltä 9 alkava *if*-lohko luo uuden rakenteen ja tallentaa sen kuvaukseen. Riville 14 tultaessa *pStorageElement* osoittaa varmasti kelvolliseen *BlockTimerStorage*-instanssiin, joten haluttu blokki voidaan lisätä sen sisältämään *BlockSetupContainer*-rakenteeseen. Lopuksi, mikäli *pStorageElement*-instanssilla ei ole vielä asetettu sopivaa ajastinta, riviltä 15 alkava *if*-

blokki pyytää protokollakirjaston *Core*-kerrokselta oikeaa millisekuntilukemaa vastaavan ajastimen. Ajastin myös käynnistetään antamalla sille *boost::bind*-rakenteella ajastimen lauetessa suoritettava funktio ja tälle oleelliset parametrit.

### 5.5.3 Laajentaminen

Protokollakirjastoon liittyvä rajapinta on simulaattorin toteutuksessa ehkä kaikkein hitaimmin muuttuva osa. Uusia käskyjä jouduttiin toteutusvaiheessa lisäämään, ja myös parametrien kohdalla voidaan nähdä tarve helpolle laajentamiselle. Protokollarajapinnan osalta tilanne on kuitenkin hieman toinen, jo siitäkin syystä, että kirjoitushetkellä protokollakirjasto tukee vain IP-yhteyksiä protokollan välitysmediana. Myös simulaattorin käyttäminen datan vastaanottajana on toteutettavan testiskenaarion kannalta turhaa, vaikkakin sille voidaan myöhemmin keksiä käyttöä.

Mainituista syistä johtuen protokollakirjaston laajennettavuuteen ei panostettu kehitysaikana yhtä paljon kuin simulaattorin muihin osiin. Mikäli simulaattoria pitäisi laajentaa tukemaan esimerkiksi datan vastaanottoa, helpoin tapa olisi ottaa mallia nykyisestä *ProtocolFeederTcp*-luokasta. Datan vastaanotto eroaa toimintona niin vahvasti datan lähettämisestä, että saman toiminnallisuuden lisäämisen olemassa olevaan luokkaan ei voida katsoa olevan järkevää.

Mikäli tavoitteena on laajentaa simulaattoria käyttämään jotakin muuta välityskanavaa, olemassa olevaa *ProtocolFeederTcp*-luokkaa voidaan mahdollisesti käyttää hyväksi. Lähetyspuolella muun muassa blokkien käsittelyyn liittyvät funktiot ovat kaikille datan lähetystapauksille samat, joten ne kannattaisi luultavasti siirtää yhteiseen kantaluokkaan. Tällöin uuden luokan tehtäväksi jää lähinnä yhteyden muodostamiseen ja sulkemiseen liittyvät tehtävät.

## 5.6 Aputoiminnot

Toteutuksen kohteena olevan simulaattorin kaltainen monimutkainen kokonaisuus vaatii tuekseen joukon apufunktioita. Tämä tarve kasvaa entisestään, kun toteutuksessa käytetään kehitysasteella olevaa protokollakirjastoa, jossa tarvitun kaltaisia apufunktioita ei ole vielä toteutettu. Simulaattoriin toteutettuihin apufunktioihin saatiin jossakin määrin apua protokollakirjaston testikoodeista sekä suppeahkoista esimerkkitoteutuksista. Apufunktioiden lisäksi simulaattori käyttää protokollakirjaston tarjoamaa *Logger*-kerrosta tukemaan omien sekä protokollakirjaston lokiviestien kirjoitusta lokitiedostoon.

Ensimmäiset apufunktiot joita simulaattoriin tehtiin liittyivät simulaattorin tilan hallintaan. Näitä olivat simulaattorin *State*-enumeraation merkkijonoksi muuttava *getStateString*()-funktio sekä kaikki yhden komennon sallitut tilat merkkijonoon keräävä *listStatesInMask*(). Koska simulaattoria tehtiin iteratiivisesti osissa, huomattiin nopeasti, että

erillinen *shouldNeverHappen*()-assertio ja sitä hyödyntävä *notImplemented*()-funktio tulivat hyödyllisiksi.

Edellä mainittujen apufunktioiden lisäksi protokollakirjaston käyttäminen vaati tuekseen joukon erilaisia funktioita. Suurin joukko näistä ovat erilaisia päivämäärän ja kellonajan muuntofunktioita protokollakirjaston dataluokkien, merkkijonoesityksen ja JSON-objektien välillä. Lisäksi, koska protokollakirjaston dataluokat sisällyttävät tiedon binääridatan tyypistä enumeraatiossa, simulaattorille täytyi toteuttaa *updateBinaryValue*()-funktio, joka asettaa oikean tyyppisen arvon *marshalling::BinaryValue*-luokkaan *template*-argumenttiaan hyödyntäen. Toteutuksessa oli helpointa käyttää *template*-rakennetta, jotta sille voitaisiin antaa määrittelemättömän tyyppinen arvo uudeksi arvoksi. Tästä seurasi kuitenkin ongelmia, mikäli annettu arvo ei täsmää oletetun binäärityypin kanssa, mitä kääntäjä ei valitettavasti osaa käännösaikana havaita. Tämänkaltaisen toiminnallisuuden olettaisi ehdottomasti kuuluvan protokollakirjaston toteutukseen jopa niin, että siitä voisi tehdä osan *BinaryValue*-luokan rajapintaa.

Simulaattorille rakennettiin erillinen *Logger*-apuluokka, jonka jäsenfunktioiden avulla voidaan helposti tulostaa mitä tahansa tulosteita aikaleimalla varustettuun tiedostoon. *Logger* periytettiin protokollakirjaston *core::LoggerAbs*-rajapinnasta, jotta se voidaan antaa protokollakirjastolle sen sisäisen toiminnan seuraamista varten. Toteutuksen alkuvaiheessa pidettiin mielessä vaihtoehto *Loggerin* toteuttamisesta singleton-suunnittelumallin ([13, Singleton Pattern]) ja staattisten jäsenfunktioiden avulla, mutta tästä luovuttiin, koska pidettiin mahdollisena, että lokitietoja halutaan ehkä tulostaa useisiin eri tiedostoihin. Käytännössä *Logger* luo rakentajassaan uuden tiedoston luontihetken aikaleimalla nimettynä, minkä jälkeen kukin tiedostoon kirjoitettava rivi aikaleimataan kirjoitushetken ajalla päivämäärä mukaan lukien.

# 6. VALMIIN SIMULAATTORIN ARVIOINTI

Simulaattorista on tällä hetkellä valmiina kehityskäytössä oleva versio, jossa on kuitenkin vielä laajentamisen varaa. Simulaattorin taustalla oleva protokollakirjasto päivittyy edelleen jatkuvasti, joten vähintään tästä syystä simulaattoria tulee vielä päivittää tulevaisuudessa. Lisäksi tälläkin hetkellä protokollakirjastossa on joitakin ominaisuuksia, joita simulaattori ei tue priorisointisyistä. Kokonaisuutena projektia voidaan kuitenkin pitää sekä opinnäytetyön että asiakkaan kannalta onnistuneena. Seuraavassa arvioidaan projektin onnistumista ensin yleisesti ja sitten kohdassa 4.1 esitettyjen vaatimusten kautta. Lopuksi pohditaan simulaattorin jatkokehitystä sen todellisen potentiaalin löytämiseksi.

## 6.1 Projektin onnistuminen

Simulaattorin tärkein ei-toiminnallinen vaatimus oli laajennettavuus, jossa onnistuttiin hyvin. Uusien käskyjen lisääminen onnistuu yksinkertaisesti lisäämällä uusi komentokäsittelijä, mikä on *if-else*-ketjun laajentamista hieman työläämpää, mutta ratkaisuna siistimpi ja pitää toteutuksen selvästi erillään muista käsittelijöistä. Käskylähteiden lisääminen on niin ikään helppoa, ja simulaattori mahdollistaa käskylähteen toimimisen joko synkronisesti tai asynkronisesti. Tämä helpottaa toteutusta huomattavasti, mikä huomattiin jo oletussyöttövirta- ja TCP-käskylähteen toteutuksien erilaisuudessa.

Uusia fysiologisia parametreja pystytään niin ikään lisäämään yksinkertaisesti periyttämällä uusi parametrikäsittelijä päätoiminnallisuuden tarjoavasta luokasta. Käytännössä laajentajan vastuulle jää kertoa parametrin nimi ja alustaa kuvaukset protokollan käyttämistä merkkijonoista mittausten alustamisesta ja arvojen asettamisesta vastaaviin funktioihin funktio-osoittimia hyväksi käyttäen. Arvojen asettamisen funktiot ovat pääosalle tapauksista samat ja valmiiksi toteutettuna, joten eniten töitä laajentajalle aiheuttaa funktioiden kirjoittaminen blokin sisälle uusien mittausten alustamiseen. Funktio-osoittimien käytöstä seuraa hyvin siisti kuvaus käsittelijäluokan rakentajassa, minkä lisäksi käsittelijä koostuu ainoastaan vapaina funktioina määritellyistä alustusfunktioista sekä mahdollisista erikoisarvojen asettajista.

Simulaattori on tällä hetkellä kehityskäytössä, kun uutta protokollaa ollaan ottamassa käyttöön valmistajan tulevassa monitoriversiossa. Tähän asti simulaattori on ollut korvaamaton apu kirjoittajalle sekä muille monitoripään projektiryhmän jäsenille korvaavien ratkaisujen puuttuessa täysin. Samaa protokollakirjastoa käytetään myös monitorin toteutuksessa, ja simulaattoriprojektin aikana on myös annettu protokollakirjaston toteutusryh-

mälle palautetta protokollakirjaston käytettävyydestä ja rajapinnoista. Jatkossa simulaattori tulee toimimaan osana kohdassa 2.3 kuvattua testausympäristöä, ja lisäksi asiakas on esittänyt halunsa käyttää simulaattoria korvaamaan nykyiset suhteellisen kevyet esimerkit protokollakirjaston käytöstä. Lisäksi asiakas on ollut asiakastyytyväisyyskyselyiden perusteella hyvin tyytyväinen työn tuloksiin, aikataulussa pysymiseen sekä iteratiivisen projektin kuluessa tapahtuneeseen kommunikaatioon.

## 6.2 Prosessin arviointi

Toteutuksen alkuvaiheessa yksikkötestien kirjoittamista harkittiin ja niiden etuja ja haittoja punnittiin suhteessa kirjoittamisen aiheuttaman työn määrään. Lopulta ajatus hylättiin sillä perusteella, että toisaalta simulaattorin toteutuksen ei odotettu olevan kovin monimutkainen, ja toisaalta kaikkien osien järkevä yksikkötestaus ei tuntunut mahdolliselta. Tämä osoittautui vääräksi, sillä joitakin simulaattorin osia olisi pystynyt testaamaan automaattisesti, mikä olisi helpottanut varsinkin regressiotestauksen toteuttamista joidenkin ominaisuuksien kannalta. Toisaalta simulaattorin rakenne mahdollisti eri osakokonaisuuksien testaamisen manuaalisesti toisistaan erillään, mikä helpotti oikean toiminnan verifioimista varsinkin protokollakirjaston muutosten jälkeen.

Eräs seikka, jota toteutuksen alkuvaiheessa ei osattu ennakoida oli se, miten paljon kehitysvaiheessa oleva protokollakirjasto voi muuttua muutaman kuukauden kestävän projektin aikana. Protokollakirjastoon tuli jatkuvasti uusia päivityksiä, jotka pakottivat osittain refaktoroimaan simulaattorin ohjelmakoodia jopa täysin uudella tavalla. Pääosin muutokset olivat kuitenkin oikeaan suuntaan, ja niiden ottaminen mukaan osaksi simulaattoria helpotti joidenkin asioiden toteuttamista muun muassa dataluokkien käsittelyn yhteydessä. Valitettavasti vain tässä vaiheessa oli jo käytetty aikaa asioiden tekemiseen vaikealla tavalla, mikä toisaalta oli myös hyvä lähtökohta palautteen antamiselle protokollakirjastoa kehittävän ryhmän suuntaan.

Asiakas oli edellä kuvatulla tavalla projektiin tyytyväinen, ja ketterien menetelmien mahdollistama joustavuus otettiin hyvin vastaan kummassakin päässä. Simulaattorin määrittely oli alun pitäen suhteellisen lyhyt dokumentti, joka keskittyi pääosin simulaattorille annettavien käskyjen kuvaamiseen. Määrittelystä löytyneet pienet epätäsmällisyydet saatiin selvitettyä aktiivisen sähköpostikeskustelun tuloksena molempia osapuolia tyydyttävällä tavalla. Projektin alettua määrittelydokumentin pääasiallisen muutosvastuu siirtyi meille, ja siitä lähdettiin muokkaamaan simulaattorin käyttöohjeeksi soveltuvaa dokumenttia. Kokonaisuutena tämä onnistui hyvin ja vaadittavan dokumentaation vähyydestä oltiin positiivisesti yllättyneitä, varsinkin kun otetaan huomioon fysiologisiin mittauksiin liittyvä toimintakenttä. Simulaattoria ei tulla käyttämään todellisissa potilastilanteissa, jolloin asiakkaan vaatimukset eivät ole yhtä tiukat kuin tuotantokäyttöön tulevilla laitteilla.

## 6.3 Vaatimusten toteutuminen

Simulaattorille annettiin kohdassa 4.1 joukko vaatimuksia, jotka toimivat suunnitteluratkaisuja ohjaavina lähtökohtina. Seuraavassa käydään määritetyt vaatimukset läpi samassa järjestyksessä kuin ne aiemmin esitettiin. Ensin tarkastellaan toimintaympäristöön liittyviä vaatimuksia, minkä jälkeen tarkastetaan toiminnallisten vaatimusten sekä toteutuksen reunaehtojen toteutuminen. Lopuksi sanotaan muutama sana alussa tehdyistä simulaattorin käyttöoletuksista ja niiden vaikutuksesta suunnitteluun.

### 6.3.1 Toimintaympäristö

Ensimmäinen toimintaympäristön vaatimus oli, että **simulaattori toimii Windows XP- sekä Windows 7- käyttöjärjestelmillä**. Simulaattorin toteutustyö tehtiin Windows 7-käyttöjärjestelmällä, ja simulaattorin toteutuksen alkuvaiheessa, kun kaikki mukaan otettavat kirjastot oli linkitetty, simulaattorin toiminta testattiin Windows XP-järjestelmässä. XP-testaus suoritettiin käyttäen Windows 7-käyttöjärjestelmän mukana tulevaa XP-moodia, joka on käytännössä Windows XP:tä ajava *VirtualPC*-virtuaalikone. Myös toteutuksen myöhemmissä vaiheissa simulaattoria testattiin satunnaisesti XP-moodissa, jotta varmistuttiin siitä, ettei mikään myöhempi muutos ole rikkonut yhteensopivuutta. Vaikka toteutusalustana toimii Windows, simulaattori suunniteltiin siten, että se olisi helposti siirrettävissä jollekin toiselle Boost-kirjastoa tukevalle alustalle. Tästä poikettiin ainoastaan yhden vaatimuksen toteutuksessa, johon paneudutaan tarkemmin seuraavassa alikohdassa.

Seuraava vaatimus oli, että **simulaattori toimii komentoriviltä ja siinä ei ole erillistä graafista käyttöliittymää**. Tämä oli helppo toteuttaa, ja se oikeastaan yksinkertaisti joitakin osia simulaattorin toiminnassa, kun graafisen käyttöliittymän vaatimuksia ei tarvinnut ottaa huomioon. Vaatimusta ei testattu erikseen.

Viimeinen toimintaympäristöön liittyvä vaatimus oli se, että **simulaattori on staattisesti linkitetty eikä sillä ole merkitystä, mistä kansiosta se käynnistetään**. Tähän vaatimukseen kiinnitettiin huomiota lähinnä toteutuksen alkuvaiheessa, kun projektitiedostoja ja mukaan linkitettäviä kirjastoja oltiin kokoamassa. Osa kirjastoista jouduttiin kääntämään uudestaan staattisen linkkauksen mahdollistamiseksi, mutta tämän jälkeen ongelmia ei ollut. Vaatimusta testattiin muutamaan kertaan simulaattorin toteutuksen aikana kopioimalla ainoastaan simulaattorin yksittäinen binääritiedosto toiselle tietokoneelle ja ajamalla simulaattoria sieltä käsin.

### 6.3.2 Toiminnalliset vaatimukset

Toiminnalliset vaatimukset liittyvät lähinnä käskyjen käsittelyyn ja ensimmäinen niistä olikin, että **simulaattori vastaanottaa käyttäjältä komentoja joko oletussyöttövirras-**

**ta (stdin) tai TCP-yhteyden yli**. Toteutusta lähdettiin tekemään oletussyöttövirran toteutusta käyttämällä, sillä se nähtiin sekä simulaattorin että toteutettavan monitorikomponentin kehityksen kannalta helpompana käyttötapana. Alusta lähtien suunnittelussa pidettiin kuitenkin mukana jako synkronisten ja asynkronisten lähteiden välillä, ja TCP-lähteen lisääminen osoittautui myöhemmin helpoksi *Boost*-kirjastoa hyväksi käyttäen. Vaatimusta testattiin implisiittisesti toteutuksen yhteydessä, joten ainoa suurempi testierä jouduttiin tekemään, kun TCP-lähde pyrittiin saamaan toimimaan oikein rivien päätösmerkkien kanssa.

Seuraava vaatimus edellyttää, että **simulaattori toimii käsky/vastaus-periaatteella, missä edelliseen käskyyn vastataan aina ennen kuin uusi käsky otetaan vastaan**. Tätä ei tarvinnut juurikaan erikseen testata, sillä tämä toiminnallisuus otettiin osaksi simulaattorin käskysilmukan perustoiminnallisuutta alusta alkaen. Kuten alikohdassa 4.1.2 todettiin, tämä vaatimus lähinnä helpottaa simulaattorin toteutusta, koska rinnakkaisiin käskyihin ei tarvitse varautua.

Kolmas vaatimus liittyi käskyjen koodaamiseen, josta määrättiin, että **sekä käskyt että vastaukset kirjoitetaan JSON-koodauksella**. JSON oli alkujaan tuntematon koodaus, mutta se osoittautui yksinkertaisuudessaan erittäin toimivaksi tämän kaltaiseen komentoriviltä ajettavaan sovellukseen. Vastauksena lähetettävät viestit tuottivat hieman ongelmia niissä tapauksissa, joissa vastauksen tuli sisältää sekaisin puhtaita sekä merkkijonoja että JSON-merkkijonoja. Tästä selvittiin muuttamalla JSON-merkkijonot puhtaiksi JSON-arvoiksi paluuviestinä lähtevän JSON-merkkijonon sisälle.

Seuraavassa vaatimuksessa edellytetään, että **simulaattori kirjaa tapahtumat aikaleimalla nimettyyn UTF8-koodattuun lokitiedostoon**. Vaatimus oli helppo toteuttaa erillisen lokikirjauksesta huolehtivan luokan sekä protokollakirjaston tarjoaman lokituskerroksen avulla. Varsinainen tiedostoon kirjoitus tehdään C++-tiedostovirtoja hyödyntäen, mutta mukaan jouduttiin ottamaan UTF8-merkkikonversioiden vuoksi Windowsille erityinen otsikkotiedosto. Tämä kuitenkin rajoittuu lokitusluokan sisälle ja se voidaan korvata yleisemmällä ratkaisulla, mikäli tarvetta ilmenee.

Viimeinen vaatimus liittyy simulaattorin toimintaan laajentamisen yhteydessä. Vaatimus kuuluu, että simulaattoriin täytyy voida **lisätä uusia mittauksia kohtuullisen helposti ja suuria rakenteellisia muutoksia tekemättä**. Tämän vaatimuksen täyttymistä kommentoitiin tarkemmin kohdassa 6.1, eikä siihen palata enää tässä sen tarkemmin.

### 6.3.3 Reunaehdot

Reunaehdot kuvasivat tiettyjä rajoituksia toteutusympäristölle ja käytettäville kirjastoille. Selkein ehto liittyy toteutusympäristöön ja edellytti, että **simulaattori toteutetaan Microsoft Visual Studio 2010-ympäristössä**. Toteutuksen alkuvaiheessa jouduttiin käyttämään VS2008-versiota, sillä protokollakirjaston käyttämä Boost-versio ei ollut tuettuna valmistajan 2010-toteutuksessa. Boost-kirjaston päivittyessä siirryttiin kuitenkin käyttä-

mään VS2010-versiota, jolloin osa kirjastoista jouduttiin kääntämään uudestaan. Vaihto ei kuitenkaan aiheuttanut mitään merkittävämpiä ongelmia, ja VS2010 havaittiin VS2008-versiota miellyttävämmäksi käyttää.

Protokollatoteutus saatiin valmiina, ja siihen liittyvä reunaehto edellytti, että **simulaattorin toteutuksessa käytetään luvussa 3 kuvattavaa protokollakirjastoa**. Protokollakirjaston hyvistä ja huonoista puolista on kerrottu jo aiemmin tässä luvussa, joten siihen ei paneuduta tässä tarkemmin. Kokonaisuutena protokollakirjasto tarjosi kuitenkin suhteellisen hyvän – joskin muuttuvaisen – alustan simulaattorin toteutukselle.

Seuraava reunaehto liittyi JSON-jäsennykseen ja edellytti, että **simulaattori käyttää JSON-alkioiden jäsentamiseen JsonCpp-kirjastoa**. Kirjasto on kevytrakenteinen ja lisenssiehdoiltaan hyvin joustava, minkä lisäksi siitä löytyvät kaikki ne oleelliset funktiot, joita JSON-toteutus vaatii mukaan lukien arvojen alustus sekä muutokset merkkijonojen ja kirjaston oman JSON-arvoluokan välillä. Kirjaston käyttöä voidaan suositella lähes mihin tahansa kevyttä JSON-jäsennystä tarvitsevaan toteutukseen.

Viimeinen reunaehto edellytti, että **simulaattori voi hyödyntää protokollakirjaston mukana tulevaa Boost-kirjastoa.** Tästä oli simulaattorin toteutuksessa suurta hyötyä, sillä varsinkin Boost-kirjaston tarjoamat kehittyneet osoittimet sekä TCP-yhteyksien hallintaan soveltuvat osat helpottivat toteutusta merkittävästi. Boostin käyttö ei aiheuttanut mitään suuria ongelmia, sillä kirjasto saatiin protokollakirjaston mukana valmiiksi käännettyinä. Boost-kirjasto päivittyi toteutuksen aikana, mikä ei kutenkaan vaikuttanut simulaattorin toteutukseen edellä mainittua Visual Studio-version päivitystä lukuun ottamatta.

### 6.3.4 Käyttöoletukset

Simulaattorille annettiin joitakin lieventäviä käyttöoletuksia, jotka yksinkertaistivat simulaattorin toteutusta merkittävästi. Merkittävin näistä on selkeästi se, että **reaaliaikaominaisuuksia ei vaadita**. Näiden ominaisuuksien todellinen toteuttaminen perinteisellä Windows-alustalla olisi ollut huomattavan haastavaa ellei jopa mahdotonta, minkä lisäksi reaaliaikaominaisuuksien takaaminen IP-verkossa olisi ollut hyvin vaativa tehtävä. Simulaattorilta vaadittiin ainoastaan, että se lähettää oikean määrän blokkeja esimerkiksi minuutin aikajakson aikana, mikä varmennettiin laskemalla minuutin aikana lähetettyjen pakettien lukumäärä ja vertaamalla sitä lähetystaajuuteen.

Seuraava oletus oli, että **simulaattori toteuttaa datan lähetyksen yhdelle alikohdassa 2.2.2 kuvatulle asiakasohjelmistolle kerrallaan**. Tämä ei helpottanut toteutusta kovinkaan paljon, sillä protokolla vaatii jo yhdenkin asiakkaan tapauksessa useiden yhteyksien hallinnointia. Käytännössä käyttöoletuksen poistuminen olisi pakottanut lisäämään protokollarajapintaan yhden rakennekerroksen, jossa kunkin asiakkaan yhteydet talletettaisiin omaan rakenteeseensa.

Viimeinen helpotus liittyi blokkien muuttamiseen ja siinä sanottiin, että **simulaattori ei muuta alikohdassa 2.2.3 esitettyä XML-kuvaustaan istunnon ollessa käynnissä**.

Tästä oli jonkin verran apua siksi, että simulaation käynnistyttyä voidaan luottaa blokkien pysyvän samankaltaisina. Muuttuvista blokeista olisi seurannut lisälogiikkaa ja rinnakkaisuuden kanssa mahdollisia ongelmia, joilta kuitenkin nyt vältyttiin.

## 6.4 Jatkokehitys

Vaikka simulaattori on tällä hetkellä vakaa ja tehokkaassa käytössä monitoripään kehityksessä, siitä puuttuu joitakin protokollassa olevia ominaisuuksia, ja sille on helppo keksiä uusia käyttötapoja sekä mahdollisia lisäyksiä käyttöliittymään. Uusia käskyjä ja parametreja ei tässä listata, mutta seuraavassa on käyty läpi muutamia mahdollisia kehityskohteita simulaattorin tulevaisuudessa.

Tällä hetkellä simulaattori tukee ainoastaan IP-pohjaista liikennöintiä, mutta tilanteen ei välttämättä tarvitse olla tämä. Esimerkiksi sarjaväylän yli tapahtuma kommunikointi on käytössä monissa mittalaitteissa ja sitä voitaisiin hyödyntää myös uuden protokollan kohdalla laajamittaiseen datan välitykseen. Myös uusien datalähteiden lisääminen ei varmasti ole kaukaa haettu ajatus, esimerkiksi voitaisiin luoda erillinen tiedostoformaatti, jossa käskyjen suoritusajat voitaisiin määrittää tiedostossa etukäteen aikaleimoilla. Samoin jokin keino tiettyjen käskysekvenssien toistamiseen voisi olla hyödyllinen, kuten myös mahdollisuus käskysekvenssien tallentamiseen jollakin nimellä helppoa uudelleen toistamista varten. Tämän kaltainen toiminnallisuus tekisi simulaattorista taas huomattavasti voimakkaamman työkalun, toisaalta se saattaisi venyttää joitakin simulaattorin käskysilmukalle asetettuja vaatimuksia.

Yksi selkeä käyttökohde on toiminta protokollassa datan vastaanottajana eli asiakkaana. Tämä olisi hyödyllistä niin simulaattorin omalle testaukselle kuin muidenkin protokollaa tukevien laitteiden testaukselle. Mikäli protokollaa haluttaisiin esitellä ulkopuolisille, simulaattorista voisi käynnistää instanssit sekä datan tuottajana että kuluttajana. Edelleen näitä molempia voitaisi ohjata esimerkiksi TCP-yhteyden yli yhtenäisestä graafisesta käyttöliittymästä, jossa simulaattorien takaisin lähettämät vastaukset visualisoitaisiin käyttäjälle jollakin tavoin.

Jossakin vaiheessa toteutusta oli myös esillä mahdollisuus luoda simulaattorille sisäinen *help*-järjestelmä, jonka lisääminen simulaattoriin olisi suhteellisen helppoa, koska komennot on jo valmiiksi jaoteltu omiin luokkiinsa. Näin kukin komento tietäisi, miten käyttäjää tulisi ohjeistaa, ja lisäksi komennoista voisi tulostaa lisäinformaatiota esimerkiksi sallituista simulaattorin tiloista. Järjestelmä voisi toimia simulaattorissa joko erillisenä komentona tai tulostaa ohjeet komentoriville komentoriviparametrin saatuaan.

# 7. PÄÄTELMÄT

Työssä toteutettiin uuden protokollan testausympäristön osana simulaattori fysiologisten mittausten lähettämiseen. Suunnittelussa lähdettiin liikkeelle valmistajan antamista vaatimuksista, joissa huomiota kiinnitettiin erityisesti simulaattorin laajennettavuuteen tulevaisuudessa. Simulaattorin toteutustyö tehtiin iteratiivisesti ketterien ohjelmistomenetelmien idean mukaisesti. Täysiveristä SCRUM-mallia ei kuitenkaan käytetty, sillä sen ei koettu tuovan riittävästi lisäarvoa ylimääräisen työn vastapainoksi.

Simulaattorin arkkitehtuurivalinnoissa onnistuttiin luomaan oleellisilta osiltaan helposti laajennettava kokonaisuus, sillä tulevaisuudessa simulaattori laajenee todennäköisesti lähinnä uusien fysiologisten parametrien, uusien käskylähteiden sekä mahdollisesti uusien käskyjen kautta. Kaikkien näiden lisäämiseen on olemassa selkeät laajennuspisteet simulaattorin sisällä, ja kussakin tapauksessa noudatetaan olio-ohjelmoinnin periaatteita siistin kapseloinnin saavuttamiseksi. Yksi huomionarvoinen yksityiskohta on funktio-osoittimien käyttö, joka ristiriitaisesta maineestaan huolimatta osoittautui todella hyödylliseksi rakenteeksi siististi laajennettavaa toiminnallisuutta toteutettaessa.

Simulaattorin toteutuksen pohjana käytettiin valmistajan kehitteillä olevaa protokollakirjastoa, joka tarjoaa protokollan toteutuksen C++-kielellä. Protokollakirjasto oli työn aloitusvaiheessa toimintakunnossa, mutta siihen tulleiden päivitysten ja suurten muutosten määrä yllätti simulaattorin toteutuksen aikana. Toisaalta tätä osattiin jossain määrin odottaa ja osittain muutoksia ajettiin myös simulaattorin toteutuksen yhteydessä protokollakirjastosta havaittujen puutteiden raportoinnin kautta. Kirjasto antoi jatkuvasta päivittymisestä huolimatta suhteellisen vakaan pohjan simulaattorin toteutukselle, ja refaktorointi ei aiheuttanut missään vaiheessa suuria ongelmia simulaattorin toteutuksen etenemiselle.

Protokolla ei ole luonteeltaan reaaliaikainen, mikä helpotti simulaattorin protokollasta riippumattomien osien kehittämistä huomattavasti. Niin kauan kuin reaaliaikaisuutta ei vaadita, Windows-alustalla on suhteellisen helppo tehdä tämänkaltainen simulaattori hyödyntäen olemassa olevaa protokollakirjastoa. Reaaliaikaisuusvaatimus olisi hankala toteuttaa ympäristössä, jossa reaaliaikaisuutta ei ole otettu kehitysvaiheessa huomioon.

Ketterien menetelmien käyttö osoittautui hyödylliseksi projektissa, jossa toteutettava järjestelmä ei ole alkujaan kovin tarkasti määritelty ja jonka ei tarvitse täyttää potilasvalvontamonitorien vaatimia täysiä varmistuskriteerejä. Simulaattoria kyettiin suunnittelemaan järkevästi ominaisuus kerrallaan, kun kokonaisuuksien väliset rajapinnat määriteltiin pääpiirteissään suunnittelun alkuvaiheessa. Ketterien menetelmien hyvä puoli on vä-

lidokumentoinnin vähyys ja osittain siitä johtuva suunnitelmien joustavuus, jolloin päästään kokonaisuutena lyhyempään toteutusaikatauluun sekä suurempaan näkyvyyteen projektin etenemisestä asiakkaan suuntaan. Kuitenkin on pidettävä mielessä se, että ketterät menetelmät eivät sovellu kaikkiin toteutuskohteisiin, sillä esimerkiksi varsinaisten monitorien kehitys joudutaan varmentamaan huomattavasti aputyökaluja tarkemmin.

Simulaattori on kirjoitushetkellä käytössä monitorin protokollatoteutuksen kehityksessä, jossa se on osoittautunut korvaamattomaksi apuvälineeksi uutta protokollaa käyttävien mittausmoduulien puuttuessa täysin kehityksen tässä vaiheessa. Simulaattori vastaa asiakkaan vaatimuksia ja se otetaan osaksi monitorin testausympäristöä kun protokollatoteutus monitorin päässä valmistuu. Lisäksi asiakas on esittänyt kiinnostuksensa käyttää simulaattoria protokollakirjaston käyttöesimerkkinä nykyisten suhteellisen kevyiden esimerkkien sijaan. Simulaattorin kehitystä jatketaan protokollakirjaston päivittyessä ja kun uusia vaatimuksia esimerkiksi parametreille tai kommunikaatiokanaville ilmenee.

# LÄHTEET


[1] Vamosi R. Smart Meters Interfering With Home Electronics. Internet, (Viitattu 2011). http://www.securityweek.com/smart-meters-interfering-home-electronics.

[2] Darpa. Transmission Control Protocol. Internet, (1981). http://tools.ietf.org/html/rfc793.

[3] Postel J. User Datagram Protocol. Internet, (1980). http://tools.ietf.org/html/rfc768.

[4] W3C. XML Schema Part 0: Primer. Internet, (2004). http://www.w3.org/TR/xmlschema-0.

[5] Schadow G. The Unified Code for Units of Measure. Internet, (1998). http://aurora.regenstrief.org/ ucum/ ucum.html.

[6] W3C. XML Schema Part 2: Data Types. Internet, (2004). http://www.w3.org/TR/xmlschema-2/.

[7] W3C. XML Binary Characterization. Internet, (2005). http://www.w3.org/ TR/xbc-characterization.

[8] Schwan K. Widener P., Eisenhauer G. *Open Metadata Formats: Efficient XML-Based Communication for High Performance Computing*. Georgia Institute of Technology, College of Computing, (2001). http://www.cc.gatech.edu/systems/papers/schwan/Widener01OMF.pdf.

[9] Nokia Siemens Networks. RobotFramework: A generic test automation framework. Internet, (2008). http://code.google.com/p/robotframework.

[10] Standard Template Library. Internet, (Viitattu 2011). http://en.cppreference.com/w/cpp.

[11] Boost C++ Libraries. Internet, (Viitattu 2011). http://www.boost.org/.

[12] Mikkonen T. Koskimies K. *Ohjelmistoarkkitehtuurit*. Talentum, 2005.

[13] Object Oriented Design. Internet, (Viitattu 2011). http://www.oodesign.com.

[14] Milea A. Solving the Diamond Problem with Virtual Inheritance. Internet, (Viitattu 2011). http://www.cprogramming.com/tutorial/virtual_inheritance.html/.

[15] Henney K. *Valued Conversions*. C++ Report 12(7), (Heinä/Elo 2000). http://www.two-sdg.demon.co.uk/curbralan/papers/ValuedConversions.pdf.

[16] Crockford D. The application/json Media Type for JavaScript Object Notation (JSON). Internet, (2006). http://tools.ietf.org/html/ rfc4627.

[17] Baptiste Lepilleur. JsonCpp. Internet, (2011). http://sourceforge.net/projects/jsoncpp.

[18] What does KISS stand for? Internet, (Viitattu 2011). http://people.apache.org/~fhanik/kiss.html.

[19] Understanding Agile Methodology. Internet, (Viitattu 2011). http://agilemethodology.org/.

[20] What is Scrum? a better way of working. Internet, (Viitattu 2011). http://www.scrum.org/what-is-scrum.