Metropolia Ammattikorkeakoulu
Tietotekniikan koulutusohjelma

Jussi Känsälä

Sovellus ilmastodatan tietokantaan tallentamista varten

Insinöörityö 10.5.2010

Ohjaaja: projektipäällikkö Asko Kippo
Ohjaava opettaja: lehtori Olli Hämäläinen

Metropolia Ammattikorkeakoulu **Insinöörityön tiivistelmä**

Tekijä Otsikko Sivumäärä Aika	Jussi Känsälä Sovellus ilmastodatan tietokantaan tallentamista varten 50 sivua 10.5.2010
Koulutusohjelma	tietotekniikka
Tutkinto	insinööri (AMK)
Ohjaaja Ohjaava opettaja	projektipäällikkö Asko Kippo lehtori Olli Hämäläinen

Tässä insinöörityössä kehitettiin huoneistoilmastodatan keräävä ja tietokantaan tallentava sovellus Metropolia Ammattikorkeakoululle Save Energy -projektin Helsinki-pilottia varten. Projektin tavoite on muuttaa kuluttajien energiankulutustapoja säästeliäämmiksi opastamalla heitä vinkeillä sekä näyttämällä heille mittaustuloksiin pohjautuvaa vertailutietoa. Työhön kuului myös itse tietokannan rakenteen suunnittelu. Kyseinen projekti on puoliksi Euroopan komission rahoittama.

Sovelluksen tuli kyetä lukemaan erilaisilta mittauslaitteilta lukemia ja tallettamaan ne SQL-relaatiotietokantaan. Sovellus ei ota kantaa mittalaitteiden tyyppiin, vaan se mikä merkitsee, on millä fyysisellä väylällä ja millä ohjelmistorajapinnalla laitteeseen pääsee käsiksi. Sovellukseen ei ollut tarvetta asettaa ehdotonta rajaa tuettaville tai ajon aikana luettaville laitteille, mutta koska tiedot luetaan säännöllisin väliajoin, on laitteita kyettävä lukemaan näiden mittaustapahtumien välissä.

Sovelluksen on tarkoitus olla jatkuvasti ajossa helposti siirrettävissä olevassa laitteessa, kuten lähiverkkoreitittimessä, johon on asennettu Linux-jakelu OpenWrt. Kun huomioon ottaa myös sen, että laite viedään toisen organisaation kiinteistöihin palomuurin taakse, ei kannattanut odottaa etähallinnan olevan mahdollista. Tämä oli luonnollisesti huomioitava sovelluksen suunnittelussa.

Sovellus suunniteltiin astah* community UML -suunnittelutyökalulla ja toteutettiin Vim-tekstieditorilla C++-ohjelmointikielellä. Ohjelmointivirheiden etsinnässä käytettiin apuna Valgrindia. Tietokantarakenne luotiin Microsoft Visio 2003 -kaavioeditorilla.

Työn tuloksena on toimiva ja tarpeeseen sopiva sovellus, jonka voisi suunnitella ja toteuttaa uusiksi projektin nykytilannetta tarkastellessa. Lähtöajatus oli, että tehdään sovellus, jolla periaatteessa pystyisi hakemaan kaikista pilotin järjestelmistä tietoa. Kuitenkin vain kaksi laitteista on sellaisia, joille on mielekästä toteuttaa tuki. Nopeampaa olisi kehittää komentosarja ajamaan sovelluksen tehtävää kutsuen pieniä laitekohtaisia sovelluksia tiedonhaussa ja lähettämään data yksiselitteisessä muodossa eteen päin.

Hakusanat	energiansäästö, save energy, ilmastodata, mittaus, mittalaite, SQL, tietokanta, C++, ohjelmointi, olio-ohjelmointi

Helsinki Metropolia University of Applied Sciences **Abstract**

Author	Jussi Känsälä
Title	Application for saving room climate data into a database
Number of Pages	50 (including appendices)
Date	10 May 2010
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Instructor	Asko Kippo, Project Manager of the Save Energy Helsinki Pilot
Supervisor	Olli Hämäläinen, Senior Lecturer

In this thesis an application for saving measured data into a database was developed for Metropolia UAS, to be used in the Helsinki pilot of the Save Energy project. The project's goal is to change consumer habits in order to save more energy by instructing them and displaying charts based on the measurement data. The database structure design was also a part of this thesis. The project is half-funded by the European Commission.

The requirements for the application were the following. It had to be able to handle multiple devices, regardless of their type and to be run at all times. A physical bus and programming interface dictated how to handle the device. An absolute limit for devices to support or how many to handle at runtime was unnecessary, but any device had to be read within a regular interval.

The UML diagrams were drawn with the astah* community UML editor and the code was written with the Vim text editor and in C++ programming language. Programming mistakes were found with the Valgrind debugging tool. The database structure was drawn with the Microsoft Visio 2003 diagram editor.

The developed application suits the need and fulfills its purpose, although it ought to be redesigned and re-implemented if the project's current status is considered. The initial idea was for it to handle any system used. However, only two devices are meaningful to support. Develop-wise it would be quicker to have small device-specific applications called and use a script to save the data into database.

Keywords	energy savings, save energy, climate data, measurement, device, SQL, database, C++, programming, object-oriented, linux

Sisällys

Tiivistelmä
Abstract
Tekniset lyhenteet

1 Johdanto 7

2 Pilotissa hyödynnettävät järjestelmät 8

3 Sovelluksen suunnittelussa huomioitavat laitteet 10

4 Uuden järjestelmän kuvaus 13

5 Tekniset ratkaisumallit 15

5.1 Selvitys ohjelmistotermeistä 15

5.2 Käyttötapaukset 17

5.3 Ohjelmakomponenttien hahmotus 19

6 OpenWrt:n asentaminen reitittimelle 31

6.1 Levykuvatiedosto 31

6.2 Asennuksen valmistelu 32

6.3 Asennuksen suoritus Microsoft Windows -työasemalla 33

6.4 Lisätietoja reitittimen etähallinnasta 33

7 Sovelluksen tuottaminen reitittimelle 35

7.1 Suunnittelu- ja toteutusohjelmistot 35

7.2 Kehitystyökalupaketin merkitys 35

7.3 Ohjelmointivirheiden korjaus 36

7.4 Lopputulos 36

8 Tietokannan rakenne 38

9 Yhteenveto 44

Lähteet 45

Liitteet

Liite 1: Säikeistetyn luokan run-metodin toteutus 47

Liite 2: Sovelluksen luokkien tärkeimpien attribuuttien ja metodien selityksiä taulukkomuodossa 48

Tekniset lyhenteet

CISC	*Complex Instruction Set Computer*; tämän arkkitehtuurin mukaisella suorittimella on pitkä konekielinen käskykanta, jonka käskyt ovat monimutkaisia. Kts. RISC.
DHCP	*Dynamic Host Configuration Protocol*; protokolla, jonka palvelinsovellus jakaa protokollan mukaisia pyyntöjä verkkoon lähettäville asiakkaille IP-osoitteen kullekin.
FIFO	*First In First Out*; ensimmäinen jonoon liittynyt on myös ensimmäinen jonosta lähtevä.
HTML	*HyperText Markup Language*; Internet-sivujen kuvauskieli. Kielestä on haarautunut useita eri variaatioita ensiversion jälkeen, mutta tässä dokumentissa lyhenne on yleiskäsite kaikille variaatioille.
IP	*Internet Protocol;* pakettikytkentäisen verkon, kuten Internetin, pohjimmainen protokolla. Vastuussa siitä, että ylempien protokollien paketit löytävät tarkotettuihin osoitteisiin.
MAC	*Media Access Control*; kutsutaan myös fyysiseksi osoitteeksi, jota voidaan käyttää laitteen tunnistamiseen. Jokaisella IP-liikenteeseen kykenevällä laitteella on yksilöllinen MAC-osoite.
MIPS	*Microprocessor without Interlocked Pipeline Stages*; eräs RISC-arkkitehtuurin mukainen suoritintyyppi.
POSIX	*Portable Operating System Interface*; käyttöjärjestelmärajapintastandardi, jonka toteuttavia funktioita ja vakioita löytyy Linuxista.
RISC	*Reduced Instruction Set Computer*; tämän arkkitehtuurin mukaisella suorittimella on lyhyt ja yksinkertainen konekielinen käskykanta.
SDK	*Software Development Kit*; ohjelmistonkehitystyökalupaketti, jonka sisältö voi vaihdella tuotteiden välillä.
SQL	*Structured Query Language*; relaatiotietokantojen yhteinen komentokieli. Kielestä löydettävissä pieniä eroja SQL-tietokantaohjelmistojen välillä.
SSH	*Secure Shell*; vahvan salauksen tiedonsiirtoprotokolla ja todennusmenetelmä.
TCP	*Transmission Control Protocol*; protokolla, jonka data kulkee IP-paketteihin sisällytettynä. Vastuussa sovellusten välisen tietoliikenteen toimivuudesta.

TFTP	*Trivial File Transfer Protocol*; alun perin Noel Chiappan suunnittelema protokolla, joka tukee vain yksittäisten tiedostojen siirtämistä järjestelmien välillä. Hakemistojen selaus ei ole mahdollista.

1 Johdanto

Tässä työssä suunniteltiin ja toteutettiin sovellus huoneilmastodatan lukemiseksi mittalaitteilta ja sen tallettamiseksi SQL-tietokantaan. Sovellus kehitettiin ajettavaksi Linux-käyttöjärjestelmäalustalla, ja se on toteutettu C++-kielellä. Työ tehtiin Metropolia Ammattikorkeakoululle (jäljempänä Metropolia).

Sovellus luotiin Euroopan Unionin puoliksi rahoittamaa Save Energy -hankkeen Helsinki-pilottia varten. Pilottiprojekti päättyy vuoden 2011 elokuussa. Hankkeessa on mukana kaikkiaan viisi eurooppalaista kaupunkia (pilotit) sekä 15 partneria eri maista, usea yritys mukaan luettuna. Hankkeen päätavoite on ”aikaansaada energiansäästöjä julkisissa rakennuksissa tuottamalla käyttäjille reaaliaikaista tietoa energiankulutuksesta mobiiliteknologian ja web-alustan kautta, motivoimalla heitä energiansäästöön pelisovelluksen sekä sosiaalisen median avulla” [1].

Pilottiin lähtivät mukaan Helsingin Ala-Malmin sekä Pihkapuiston peruskoulut, joista on määrä hakea erilaisilla mittausjärjestelmillä energian kulutukseen liittyviä lukemia. Metropolia tuottaa Helsingin kaupungille järjestelmän, joka tuottaa palvelun tarjoten eri kohderyhmille kuvaajia koulun omalla palvelimella sijaitseville HTML-sivuille [2]. Eri kohderyhmiä ovat kuluttajat eli oppilaat sekä koulujen henkilökunta ja kaupungin virkamiehet, joita kiinnostavat erityisesti energiansäästöistä tulleet rahalliset säästöt. Palvelu on toiminnassa vähintään vuoden 2016 loppuun saakka [1], ja sitä tulee ylläpitämään vähintään yksi vakinainen työntekijä. Kaupunki odottaa palvelun kuvaajien havainnollistavan energiankulutusta ymmärrettävästi ja informatiivisesti.

Mittauksia ei tehdä vain seurantaa varten, vaan jotta voitaisiin esimerkiksi kahden peräkkäisen vuoden saman kuukauden kulutustuloksia vertailemalla todeta energiankulutustasojen odotettu lasku kulutustapojen toivotun muutoksen seurauksena. Helsinki-pilotin energiansäästöajatuksiin kuuluu, että energiaa ei tule säästää siten, että ihmiset joutuisivat työskentelemään huonommissa olosuhteissa. Energiaa säästetään energiatehokkailla laitteilla sekä käyttämällä laitteita vain silloin kun tarvitaan sekä hyödyntämällä tuotettua lämpöä mahdollisimman paljon [2].

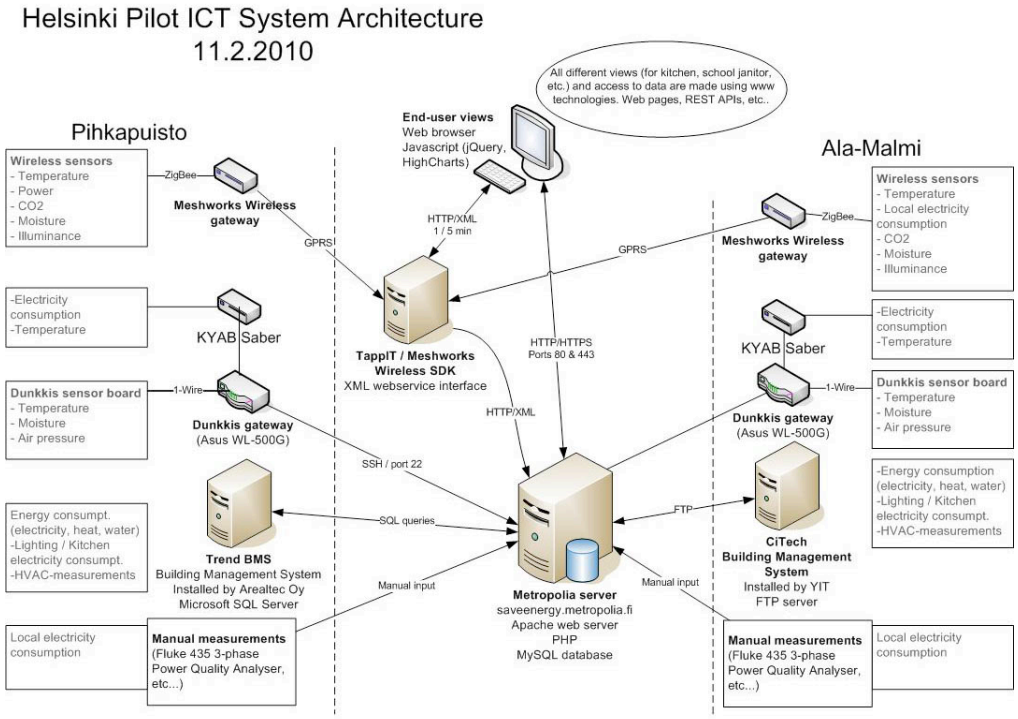
2 Pilotissa hyödynnettävät järjestelmät

Hankkeen pilottikaupungeissa päätökset mittalaitteista on delegoitu asiantuntijaorganisaatioille. Molempiin Helsinki-pilotin kouluihin on asennettu lähes vuosikymmen sitten kiinteistöautomaatiojärjestelmät (kuvassa 1 "Building Management System"), mutta ne eivät tarjoa dataa kaikista kiinnostavista kohteista tai asioista, joten lisäksi on etsitty ja ostettu muita laitteita.

Kuvasta 1 ilmenee vielä päivämäärälle 11.2.2010 päivätty pilotin järjestelmäarkkitehtuurisuunnitelma. Viimeisin kuva löytyy hankkeen kotisivuilta Helsingin alasivulta [3]. Kuvassa on sivuilla nimettyihin kouluihin kaavaillut laitteet sekä keskellä koulujen ulkopuolella sijaitsevat palvelimet, joihin tieto kerätään. Ylin kuvake on loppukäyttäjän eli kaupungin virkamiesten, koulujen henkilökunnan, oppilaiden sekä ylläpidon työasemaa kuvaava.

Yksi laitteista on Nomovok Oy:n tuote, Dunkkis-gateway, jota silmällä pitäen tämän työn sovelluskin kehitettiin. Syyt miksi Dunkkis valittiin yhdeksi mahdolliseksi laitteeksi muiden joukkoon, ovat sen siirrettävyys, mahdollisuus siirtää mittausdata langattoman lähiverkon välityksellä sekä projektin luonteeseen sopivat mitattavat suureet. Dunkkis-gateway on käytännössä lähiverkkoreititin Asus WL-500gP V2. Reitittimeen liitetään Dunkkis-piirilevy, joka on varsinainen ilmastomittauslaite.

Kuvan KYAB Saber on omanlaisensa laite, jonka voi liittää ethernet-kaapelilla lähiverkkoon tai ModBus/TCP-siltauslaitteen kautta samaiseen edellä mainittuun reitittimeen [4]. Tämäkin laite on kokonsa vuoksi helposti siirrettävissä ja tarjoaa kiinnostavia suureita.



Kuva 1. Matti Peltoniemen luoma Helsinki-pilotin järjestelmäarkkitehtuurikaavio [3].

Tässä työssä tuotetun sovelluksen voi ajatella sijoittuvan Dunkkis-gatewayn eli reitittimen kohdalle molempiin kouluihin. KYAB:n Saberia kaavailtiin luettavaksi reitittimeltä käsin.

Kuvassa mainittuja kiinteistöautomaatiojärjestelmiäkin olisi ollut mahdollista lukea sovelluksella, ja Pihkapuiston järjestelmää kaavailtiinkin näin luettavaksi. Koska kyseinen järjestelmä tarjoaa datan relaatiotietokannasta, mikä vaatii SQL-lausekkeiden lähetystä tietokantapalvelimelle, on hakulausekkeiden muutoksien käyttöönotto nopeampaa, kun lukusovelluksen toteuttaa skriptikielellä. Erityisesti vain muutamaa hakulauseketta käyttävä ohjelmointikielinen sovellus on myös liian raskas ratkaisu. Skriptikielellä toteutettujen sovellusten huono puoli on, että ne vaativat skriptitulkin ylipäänsä käynnistyäkseen.

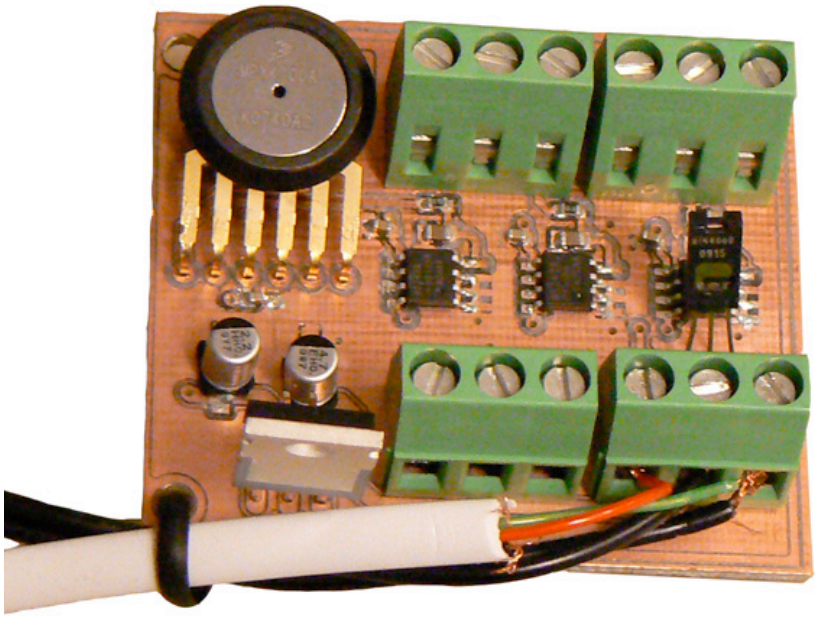
Myös kuvan Meshworks Wireless -järjestelmällä on omat, monet etunsa. Sekin jäi sovelluksen ulkopuolelle, sillä sen laitteet lähettävät datan kolmannen osapuolen palvelimelle, josta datan saa haettua jalostetussa muodossa XML-tiedostoina Internet-yhteyden välityksellä ja on siten kevyempää toteuttaa skriptatulla sovelluksella.

3 Sovelluksen suunnittelussa huomioitavat laitteet

Pilottiin osallistuviin kouluihin kaavailluista laitteista vain kahta laitetyyppiä varten suunniteltiin toteutettavan tuki. Sovelluksen alusta on Asus WL-500g Premium v2 -reititin, johon asennettiin Linux-jakelu OpenWrt. Seuraavaksi esitellään laitteet, joille tällainen sovellus olisi käypä ratkaisu niiden lukemista varten.

Dunkkis-tuotteeseen kuuluvat sovellukset sekä piirilevy, jossa on integroituna kolme anturia. Niistä saadaan ilman suhteellinen kosteus (yksikkönä %, komponenttikoodi HIH4000), paine (mBar, MPX4100A), lämpötila (°C; °F; K; °R, DS2438) sekä vapaavalintainen suure piirilevyn analogisisäänmenoon.

Virtaa saadakseen kuvan 2 piirilevy tulee kytkeä muuntimella verkkovirtaan. Tuote on suunniteltu varta vasten Asus WL-500g Premium V2 -reitittimen kanssa käytettäväksi. Se saattaa toimia muillakin reitittimillä, joihin Linux-jakelu OpenWrt on onnistuttu mukauttamaan.



Kuva 2. Dunkkis-piirilevy johtoineen.

Data haetaan piirilevyltä RJ-11 -kaapelia pitkin kuvan 3 USB-1-wire -sovittimelle, joka kytketään reitittimen USB-liitäntään. Ohjelmallisesti tulee käyttää owfs-

ohjelmakirjastoa, eli joko kirjoittaa sovellus, joka kutsuu kyseisen kirjaston sisältämiä funktioita tai ajaa valmista owread-terminaalisovellusta. RJ-11 -kaapelin johtimet kiinnitetään levylle ruuveilla, ja toinen pää liittimellä sovittimeen. Sovittimen malli on DS9490R.

Kuva 3. USB-1-Wire -sovitin DS9490R. [5]

Kuvan 4 KYAB Saberista saa mittausdatan ulos kahdella tavalla. Yrityksen suosittelema tapa on liittää laite lähiverkkoon, jossa jonkin toisen laitteen DHCP-palvelu antaa Saberille IP-osoitteen. Kun näin on tehty, laite lähettää mittaamansa datan automaattisesti KYAB:n palvelimelle. Saberin tarjoamia suureita ovat lämpötila (°C) sekä energiankulutus (kWh). Siltauslaitteen läpi tapahtuva paikallinen luku käyttää sekin itse asiassa TCP/IP-tietoliikennettä, mutta tällöin on itse kirjoitettava ModBus/TCP-protokollaohjelmakirjastoa hyödyntävä sovellus. Tämä ei ole yrityksen suosittelema tapa, mutta teknisesti mahdollinen.

Kuva 4. KYAB Saber ylhäältä kuvattuna.

Saberin voi liittää TCP/IP-verkkoon ModBus/TCP-siltauslaitteen välityksellä. Siltauslaitteen takana olevan laitteen kanssa ohjelmallinen kommunikointi toimii käyttäen laitteelle annettua IP-osoitetta ja TCP-porttia. Siltauslaite liitetään Saberissa RS485-porttiin. ModBus/TCP-laitteille on käytäntönä antaa osoite manuaalisesti eikä se saa muuttua [4].

Sekä Dunkkis-piirilevyn että Saberin voi liittää edellä mainittuun reitittimeen. Juuri tähän reitittimeen investoitiin muun muassa siksi, että Nomovok Oy on kyseisen laitteen vanhemmalla versiolla kehittänyt omalta Dunkkis-piirilevyltään datan lukevan sovelluksen, joka on toteutettu php-ohjelmointikielellä. Dunkkis-tuotteen käyttö reitittimen kanssa edellyttää Linux-jakelu OpenWrt:n asennusta reitittimeen. Dunkkis-piirilevyä voi käyttää samalla tavalla myös PC-tietokoneeseen kytkettynä, kunhan käyttöjärjestelmään on asennettu owfs-ohjelmakirjasto.

4 Uuden järjestelmän kuvaus

Järjestelmäarkkitehtuuriin kuuluu laitteiden ja palvelimen välinen *middleware* tai *mediaattori* eli tässä projektissa yksi tai useampi sovellus, joka siirtää mitatun datan laitteilta palvelimelle. Tämä mediaattorin kuvaus sopii toki vain tähän tai vastaavaan järjestelmäarkkitehtuuriin.

Toiminnalliset vaatimukset

Toiminnalliset vaatimukset määrittelevät, mitä palveluja ohjelmiston on tarjottava, miten se reagoi syötteisiin ja miten se käyttäytyy tietyissä tilanteissa [6, luku 4, kalvo 14].

Sovelluksen perustarkoitus on lukea mittalaitteelta saatava data ja tallentaa se Internetin välityksellä vähintään yhdelle tietokantapalvelimelle. Sovelluksen on pystyttävä lukemaan mittalaitteelta saatava data, joka on myös laitekohtaisessa muodossa. Voi siis olla tarve osin muokata tätä dataa ja uudelleenjärjestää eri kentät tietokantaa varten sekä luoda SQL-standardin mukainen lauseke tietojen tallentamiseksi.

Mitä kauemmin sovellusta ajetaan, sitä todennäköisemmin jossain vaiheessa tapahtuu jotakin epäideaalista, josta on hyvä raportoida. Vakavimmat tapahtumat, joista tulee raportoida, ovat toki varsinaisia virheitä ja niistä voisi samantien lähettää sähköpostia. Jos mittauksista löytyy epäkelpoja lukemia, niistäkin tulee raportoida mutta ei välttämättä lähettää sähköpostia.

Ei-toiminnalliset vaatimukset

Ei-toiminnalliset vaatimukset määrittelevät rajoitukset toiminnallisille vaatimuksille. Ne kertovat, mitä ehtoja järjestelmän on täytettävä, jotta toiminnalliset vaatimukset voidaan toteuttaa [6, luku 4, kalvo 17].

Sovellus ei saa kadottaa yhtään mittausdataa, minkä vuoksi data olisi pidettävä massamuistissa, kunnes se on varmasti tallennettu etätietokantaan. Sovellusta ei aivan

tällaiseksi suunniteltu, vaan uskallettiin luottaa järjestelmän olevan niin nopea, että massamuistiin tarvitsee tallentaa dataa vain, kun tietokantayhteys on poikki. Tietokantayhteyden ollessa poikki uutta tietokantayhteyttä on yritettävä luoda säännöllisesti. Kun tietokantayhteys saadaan luotua, aloitetaan datan purku massamuistista vanhimmasta lähtien.

Ohjelmalliset rajoitteet

Linuxin ajaminen laitteella asettaa rajoja sekä luo mahdollisuuksia sovellukselle. Sitä ei tulla ajamaan muissa käyttöjärjestelmissä, joten niitä ei tarvitse suunnitteluvaiheessa ottaa huomioon. Sovellus voi hyödyntää POSIX-yhteensopivia säikeitä (pthreads) ajaakseen useita prosesseja näennäisesti rinnakkain. Todellisuudessa käyttöjärjestelmän tehtävien ajoittamisesta vastaava prosessi päättää, mikä prosessi saa suoritusaikaa. Lisätoimintojen tuomiseksi sovellukseen voi linkittää funktiokutsuja valmiista ohjelmakirjastoista.

Laitteistorajoitteet

Reitittimessä on erityisesti lähiverkkoliikenteen prosessointiin suunniteltu suoritin, kellotaajuudeltaan 240 MHz:n Broadcom 5354, joka ei tarvitse varsinaista jäähdytysratkaisua. Sovelluksen tuottamisen kannalta on merkityksellistä, että suoritin ei ole PC-tietokoneiden tapaan osa CISC-arkkitehtuurin mukaista x86-suoritinperhettä, vaan se on 32-bittinen RISC-arkkitehtuurin mukainen MIPS, jolla on myös erilainen konekielinen käskykanta. Reitittimeen saa USB-liitäntään liitettävän syöttölaitteen kiinni, mutta sellaisen mielekäs käyttö on epävarmaa. Siitä huolimatta reititintä pääsee hallitsemaan HTML-sivuilla verkon välityksellä. Käyttöjärjestelmän sisälle pääsee verkon välityksellä Telnetillä ja SSH:lla. Keskusmuistia reitittimessä on vain 8 MB ja massamuistia flash-piirillä 32 MB. Ethernet-liitäntöjä lähiverkkoa varten on neljä sekä yksi reitittimen liittämiseen laajempaan verkkoon. Reitittimessä on USB-teknologian version 2.0 piiri sekä kaksi USB-liitäntää [7; 8].

5 Tekniset ratkaisumallit

5.1 Selvitys ohjelmistotermeistä

Ennen suunnittelusta kertovaan osuuteen perehtymistä on syytä selvittää muutaman avainsanan merkitys tässä dokumentissa, sillä niiden merkitys on riippuvainen tuotoksesta. Selityksen arvoisia ovat myös pari muuta sovelluksen rakennetta valottava asiaa.

Moduuli on yksi ohjelmistokomponentti, jonka toteutus eli lähdekoodi on erillään muista komponenteista ja jonka ohjelmakoodi käännetään konekieliseksi omana vaiheenaan. Moduuli saattaa kutsua toisessa moduulissa toteutettuja palveluita, jolloin se on riippuvainen tästä toisesta moduulista. Sovellus koostuu vähintään yhdestä moduulista. Moduulia ei tule sekoittaa myöhemmin tuleviin luokkiin tai olioihin.

Dynaamisesti ladattava moduuli tai *jaettu objekti* on samanlainen komponentti kuin moduulikin, mutta se ei ole osa sovellusta, vaan ladataan ajon aikana. Ohjelmiston tuottamisvaiheessa annetaan erityisiä parametrejä (esim `-shared -Wl,-soname,libDunkkis.so.0.1 obj/Dunkkis.o -o lib/libDunkkis.so.0.1`) linkittimelle tällaisen moduulin kohdalla. Tällaisia käytetäänkin ohjelmistojen lisäosina. Näiden sisältämän toiminnallisuuden käyttöönotto voi olla monimutkainen operaatio ohjelmoijan toteutettavaksi, ja tapoja on monia [9].

Moduulitiedoston sisältämä symbolidata ladataan keskusmuistiin valmiilla symboleidenlatausfunktiolla, jolle annetaan symbolialueen muistiosoite sekä ladattavan symbolin (funktio tai muuttuja) tunniste, kuten ”oma_funktio”. Nyt ohjelmoijan toteuttamaa C-kielistä funktiota, jonka prototyyppi on `void oma_funktio(void)`, kutsutaan ilmaisulla `(*oma_funktio)();`. [9]

Linkitin on työkalu, joka prosessoi kääntäjän ohjelmakoodista tuottamat objektitiedostot kootakseen ne yhdeksi tai useammaksi yksittäiseksi suoritettavaksi binääritiedostoksi. On täysin sovelluksen tuottajan päätettävissä, montako binääritiedostoa luodaan. Koska

sovellus voi kutsua jonkin toisen tahon kehittämän ohjelmakirjaston sisältämiä funktioita tai luokkia, linkitin selvittää myös sovelluksen koodissa tehdyt viittaukset näihin. Kaikki viittaukset on selvitettävä, jotta binääritiedosto voidaan tuottaa [10].

Laitemoduulit säästävät keskusmuistia pitkällä tähtäimellä. Tämän sovelluksen haluttiin olevan kykenevä lukemaan periaatteessa miltä tahansa laitteelta dataa. Tällainen vaatimus edellyttää, että sovellukseen kirjoitetaan lukutuki jokaista laitetta kohden. Useiden laitteiden käyttö tulisi hankalaksi ja sovelluksen ydintä pitäisi jatkuvasti muokata, ellei toteutettaisi rajapintaa, joka yleistää jokaisen laitteen yhden olion kautta käytettäväksi. Vaikka laitetukia toteutettaisiin useita, ei niitä kaikkia ole tällä ratkaisulla pakko ladata muistiin. Siten keskusmuistiakin säästyy pitkällä tähtäimellä.

Laiterajapintaluokka *Device* osoittautuu käteväksi siten, että sen tyyppiseksi luotu osoitinmuuttuja voidaan asettaa osoittamaan dynaamisesti ladatun moduulin sisältämään laitekohtaisen luokan, kuten Dunkkiksen, olioon, tietämättä, minkä tyypin olio osoittimen päässä varsinaisesti on. Sovelluksen sisällä *Device*-rajapinnan mukaisen olion tyypin voi selvittää moduulitiedoston nimestä. Sovellus kuitenkin suunniteltiin sellaiseksi, ettei missään vaiheessa ole pakko tietää, miltä laitteelta (oliolta) mittausdata tulee. Olion tyyppiä voi käyttää vaikkapa siten hyödyksi, että asettaa olion jäsenmuuttujien arvoja tai pyytää tietoa juuri sen tyypin oliolta.

Luokan instantiointi jaetusta moduulista. Laitekohtaiset luokat toteutettiin dynaamisesti ladattavien moduulien sisään, joista ne on ladattava sekä instantioitava eli varaamalla luokalle keskusmuistista tilaa ja asettamalla jäsenmuuttujille ohjelmoijan määrittelemät alkuarvot. Laiterajapinta *Device*n lisäksi moduulinlatauksen käyttöä helpottamaan toteutettiin kiinteäksi osaksi myös dynaamisesti ladattavan moduulin latausmoduuli (luokka), jolloin riittää, että jossain sovelluksen toimesta luettavassa tekstitiedostossa listataan ladattavien moduulien nimet, eikä pääohjelmassa tarvitse erikseen mainita, mitä moduuleja ladataan.

Tässä proseduurissa käytetään symboleidenlatausfunktiota, kuten edellä on kuvailtu, poikkeuksena, että funktion tunniste on luokan instanssin palauttavan funktion tunniste.

Moduulinlatausluokassa on olion muistiosoitteen palauttava metodi, jossa tätä luontifunktiota kutsutaan. Olio on bitteinä keskusmuistissa, mutta epämääräisessä muodossa, ja siksi pitää tulkita uudelleen ennen kuin ladattua luokkaa voi käyttää. Tulkinta tehdään ilmaisulla `reinterpret_cast<Device*> (objekti)` [9].

Pääohjelma on se funktio, jota käyttöjärjestelmä kutsuu, kun sovellus käynnistetään. Siinä tehdään alustustoimenpiteet sovelluksen moduuleille, otetaan moduulit käyttöön, ja kun tehtävät on tehty tai sovellus lopetetaan, tehdään edelliset asiat päinvastaisessa järjestyksessä ja vapautetaan varatut keskusmuistialueet.

5.2 Käyttötapaukset

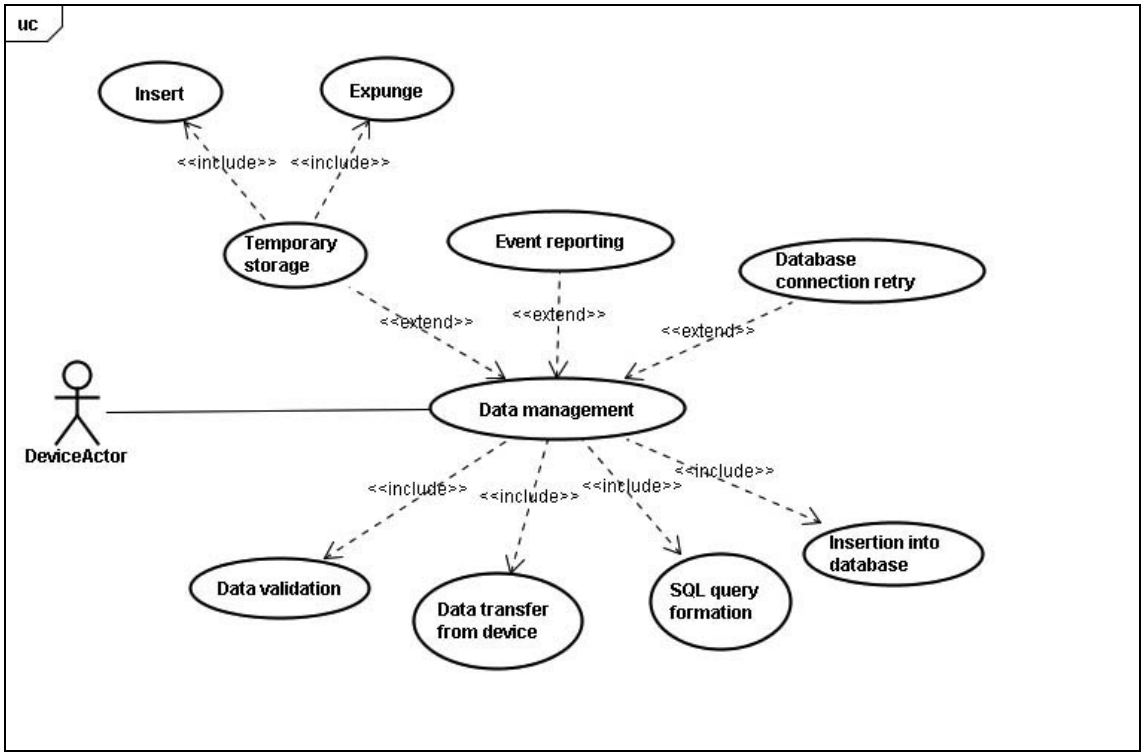
Käyttötapauksia ovat järjestelmän palveluidean ympärille suunnitellut toiminnot ja tavat käyttää järjestelmää. Käyttötapauskaaviossa esitetään järjestelmän toiminnot ulkopuolisen tarkkailijan näkökulmasta. Kaavioon on suositeltavaa kuvata pikemminkin vähän kuin paljon. Funktiot ja tapahtumaketjut toimintojen takana esitellään muissa kaavioissa [11, kpl. 4.2]. Kuvassa 5 käyttötapauksia on useampia kuin on tarpeellista, mutta vain sovelluksen toiminnan havainnollistamiseksi. Käyttötapauskaaviosta voi joissain tapauksissa hahmottaa myös sovelluksen eri moduulit ja niiden väliset suhteet. Käyttötapauskaavioita sekä muita UML-kaavioita piirretään sovelluksen rakenteen ja toiminnan havainnollistamiseksi. Kaikkia kaavioita ei suinkaan ole pakko jokaisen sovelluksen kohdalla luoda.

Käyttötapauskaaviossa kaikki avainsanalla *extend* merkityt käyttötapaukset laajentavat niiden osoittamaa käyttötapausta [11, kpl. 4.4.2]. Keskeisintä toimintoa lähinnä olevat kolme käyttötapausta katsottiin laajennuksiksi sillä perusteella, että ideaalitilassa ne eivät ole kriittisiä. Sovellus on ideaalitilassa silloin, kun tietokantayhteys toimii ja laitteista saadaan dataa.

Avainsanalla *include* merkityt toiminnot toimivat osana sitä käyttötapausta, josta niihin viitataan [11, kpl. 4.4.1]. Tällainen toiminto voi olla osa useampaakin käyttötapausta, jolloin siihen viitataan niistä kaikista. Vaikka toiminto olisikin osa vain yhtä käyttötapausta, voi sen eriyttämiseen olla muitakin syitä, kuten havainnollistaminen.

Ulkopuolisen toimijan eli aktorin ja järjestelmän vuorovaikutuksen aloittaa tässä sovelluksessa *DeviceActor*. Koska luvussa 2 mainitut laitteet eivät aiheuta laitteella keskeytystä, on aktori käytännössä osa sovellusta ja jokin olio, joka hakee säännöllisin väliajoin laitteelta uudet mittaustiedot. Lähtökohtaisesti aktori on järjestelmän ulkopuolinen käyttäjä tai toinen järjestelmä. Olio huomauttaa sovelluksen keskeisintä osaa eli kontrolleria, kun sillä on uutta dataa.

Data management -käyttötapausta laajentavat siis kolme tapausta. Väliaikaistallennuksen hoitaa *Temporary storage*, johon kuuluu datan lisäys (*Insert*) sekä haku ja poisto (*Expunge*). Se on oma olionsa, kun taas tapahtumaraportointi *Event reporting*in toiminnallisuuksia on useassa oliossa. Tietokantaoliossa taas on sellainen ominaisuus, että kun yhteyden huomataan olevan poikki, se osaa itsenäisesti ja säännöllisesti yrittää uuden yhteyden avaamista (*Database connection retry*).



Kuva 5. Sovelluksen käyttötapaukset hahmottavat myös sovelluksen moduuleja.

Datan saapuessa laitteelta (*Data transfer from device*) se tarkistetaan (*Data validation*) siltä varalta, etteivät lukemat ole mielekkäitä. Mittalukemat ovat laitekohtaisia, joten

tarkistus on paras tehdä laiteoliossa. Kun data on siirretty kontrollerin kautta tietokantaoliolle, se asetetaan muuten valmiiseen SQL-lausekkeeseen (*SQL query formation*) ja lähetetään tietoverkon läpi tietokantapalvelinohjelmalle tietokantaan tallennettavaksi (*Insertion into database*).

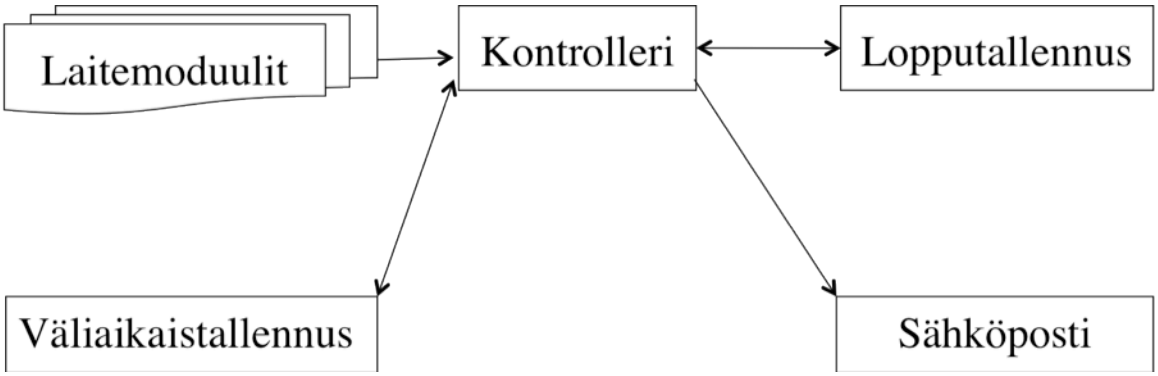
5.3 Ohjelmakomponenttien hahmotus

Vaatimusmäärittelyjen ja käyttötapauskaavion pohjalta voi muodostaa moduulikaavion (kuva 6), jossa moduulien välille on merkitty tiedonkulkusuunnat.

Sovellukseen toteutetaan yksi tai useampi laitteelta tavalla tai toisella tietoa säännöllisin väliajoin lukeva moduuli, *kontrolleri*. Kontrollerin rooli on ”vedellä muita moduuleja naruista” pääohjelman delegoimana. Kontrolleri voi olla joko pääohjelmassa tai siitä irrallaan toteutettuna. Sillä ei tarvitse olla absoluuttista valtaa sovelluksen muihin osiin, mutta sen tulee käskeä niitä.

Lopputallennusmoduuli on käytännössä tietokantapalvelimelle datan lähettävä olio. SQL-lausekkeen luonti tehdään tässä. Tietokantayhteyden ollessa poikki uutta yhteyttä yritetään luoda säännöllisesti. Kun uusi yhteys luodaan, aloitetaan datan purku massamuistista vanhimmasta rivistä lähtien.

Informatiiviset ilmoitukset, varoitukset ja virheet välitetään kaikilta moduuleilta kontrollerille, joka edelleenvälittää ne käyttöjärjestelmän tapahtumarekisteripalvelulle. Nämä moduulit eivät vielä raportoi varoituksista tai ongelmista reaaliajassa. Sitä varten sovellus kaipaisi myös sähköpostimoduulin.



Kuva 6. Sovelluksen operatiivisuuden kannalta olennaiset moduulit ja tiedonkulkusuunnat.

5.3.1 Yhtä laitetta tukeva sovellus yhtenä prosessina

Työtä aloittaessa oli lähtökohtana ajatus, että sovellukseen voi toteuttaa tuen jokaista laitetta varten, eikä ole järkevääkään toteuttaa jokaista laitetta varten omaa sovellusta, jossa on kaikki edellä mainitut ominaisuudet.

Pääohjelma toteuttaa sovelluksen pääasiallista tehtävää. Kun kaikki oliot on alustettu käyttövalmiiksi, pääohjelmassa astutaan silmukkaan, josta poistutaan vasta kun sovellus saa ulkopuolisen signaalin. Erillisestä kontrollerista olisi se hyöty, että silmukka pysyisi lyhyenä. Silmukassa on tietty odotusjakso, jonka päätyttyä laitteelta luetaan uusi data. Data välitetään tietokantaan lähetettäväksi taikka väliaikaissäilöön.

Tämä kyllä toimisi, mutta eri oliot eivät ole itsenäisiä, vaan pääohjelma päättää, milloin mikäkin olio saa suoritusaikaa. Edellä mainitut tehtävät voi pilkkoa ja delegoida olioiden itsenäisesti suoritettaviksi, jolloin sovelluksesta tulee joustavampi. ”Diktaattorisilmukka” aiheuttaa myös ylimääräisen viiveen odotusjakson lisäksi, ja se kertaantuu, jolloin datan lukufrekvenssi pitenee ja pitenee. Tämä voidaan toki jokaisella silmukan iteraatiolla korjata. Puhe on toki vain sekunnin murto-osista kuukausien tai pitempien aikajaksojen aikana. Kuitenkin tällä on vaikutusta esimerkiksi energiankulutuslukujen laskennassa.

Mikäli järjestelmään, jossa sovellusta ajetaan, kytketään toinen laite, jota halutaan lukea, tulisi toteuttaa toinen samanlainen sovellus, joka eroaisi vain laitemoduulin toteutukselta. Muistiakin kuluisi hukkaan. Toistuvaa ohjelmakoodia tulee välttää, mikä

tehdään abstrahoimalla ja pitämällä yksi osakokonaisuus yhdessä paikassa. Tällöin kehitysmuutoksetkin tarvitsee tehdä vain kertaalleen. Useamman sovelluksen tapauksessa jokainen pitäisi myös käynnistää erikseen.

5.3.2 Useamman laitteen tuki sekä usea prosessi haaroittamalla

Kun halutaan yhden sovelluksen saavan dataa usealta laitteelta, ei ole mielekästä eikä tehokasta, että kontrolleri kysyy jokaiselta laitteelta vuorollaan uudet tiedot, vaikka niitä ei välttämättä edes ole. Tehokkaampaa on luoda jokaista laitemoduulia varten yksi lapsiprosessi, jossa olevassa silmukassa hoidetaan datan luku, ilmoitus kontrollerille uuden datan saapumisesta sekä odotus ennen seuraavaa lukukertaa. Lapsiprosessi on äitiprosessista luotu kopio [12; 13], jossa voi haaroittamisen jälkeen tehdä äitiprosessista riippumattomia asioita.

Koska lapsiprosessin halutaan ilmoittavan kontrollerille uudesta datasta, lapsiprosessin on tiedettävä tai saatava jollakin tavalla selville äitiprosessista sellainen yksilöivä tieto, jota hyödyntämällä ilmoitus saadaan perille. Tämä voi olla prosessin tunnusluku tai olion muistiosoite. Olioita on mielekkäämpi käyttää säikeiden kanssa, ja tämä ratkaisu käydäänkin läpi luvussa 5.3.3.

Tunnuslukua voi käyttää hyväksi siten, että mittausdata tallennetaan tietylle alueelle keskusmuistissa, josta äitiprosessi käy lukemassa, kun on saanut lapsiprosessilta huomautussignaalin. Toinen vaatimuksiin sopiva tapa on POSIX-standardin mukainen asynkroninen tiedonsiirtopalvelu, joka vaatii samaisen prosessin tunnisteen, jotta käyttöjärjestelmä tietää, mille prosessille huomautussignaali lähetetään. Tämän palvelun käyttöönotto vaatii jokaisessa prosessissa usean parametrin esiasettamisen ja tiedonsiirron hallitsemisen sekä virheenkäsittelyn, mikä on kokonaisuutena varsin monimutkainen hyödynnettävä.

5.3.3 Useamman laitteen tuki sekä säikeet

Säikeet ovat osa pääohjelmaprosessia, mutta kuitenkin sillä tavalla erillisiä, että niillä on oma ohjelmoijan asettama funktio, jonka rungon käskyjä suorittaa. Niitä ei luoda

haaroittamalla eivätkä ne ole aluksi kopioita äitiprosessista. Ne ovat siten myös nopeampia luoda [14], mutta mikäli on tarve asettaa arvoja useille prosessille ominaisille muuttujille, täytyy niille kirjoittaa erikseen koodirivit.

Tähän sovellukseen toteutettiin *Thread*-luokka POSIX-standardin mukaisen säieohjelmakirjaston funktioiden ympärille käyttöä yksinkertaistamaan. Mikä tahansa muu luokka voi periä tämän luokan muuttujat ja metodit (luokan sisällä oleva funktio), joten perivän luokan instantioidulla oliolla ne ovat jo valmiina. Prosessin suoritus ei käynnisty vielä, mutta olion metodeja voi jo kutsua. Prosessin suoritus käynnistetään kutsumalla sen *Thread*-luokalta perimää create-metodia, jossa kutsutaan käyttöjärjestelmän funktiota pthread_create, jolla säie luodaan. Funktiolle annetaan parametreina muistiosoite, johon se tallentaa säikeen tunnisteluvun, säieattribuuttimuuttujan osoitteen, *Thread*-luokan sisäisen start-metodin osoitteen sekä perivän olion metodin osoitteen, jota halutaan ajaa muiden prosessien rinnalla [14].

Säieratkaisulla pääohjelmassa luodaan tietokantaolio, kontrolleri, laiteoliot silmukassa sekä asetetaan niiden muistiosoitteet vektorimuuttujaan. Olioiden attribuuteille asetetaan alkuarvot. Laiteoliot sekä tietokantaolio tulee esitellä kontrollerille ja kontrolleri niille, eli erityistä metodia kutsumalla antaa olion muistiosoite, jotta oliot tietävät, missä tiettyä asiaa hoitava olio muistissa sijaitsee. Tämän jälkeen oliot käynnistetään mielivaltaisessa järjestyksessä. Sovellus on valmis hoitamaan tehtäväänsä ja oliot suorittavat omia asioitansa itsenäisesti ja rinnakkain.

Mittausdatan siirto laiteoliolta kontrollerille toteutettiin seuraavasti. Laiteolio lukee säännöllisin väliajoin uusimmat mittauslukemat ja kutsuu kontrollerin metodia, jolla kerrotaan, missä muistiosoitteessa sijaitsevalla laiteoliolla on uutta dataa luettavissa. Kontrolleri asettaa tämän osoitteen sisäisen palvelujononsa loppuun. Kontrollerin silmukassa tarkistetaan joka kerta, onko palvelujonossa yhtäkään laitetta. Mikäli on, kutsuu se ensimmäisen (jonon vanhimman) laiteolion datanlukumetodia, joka antaa kontrollerille laiteolion jonon ensimmäisen tietueen muistiosoitteen. Tämä ratkaisu on myös yleisempi kuin käyttöjärjestelmäkohtaiset signaloinnit ja tiedonvälitystekniikat,

eli se toimisi missä tahansa käyttöjärjestelmässä. Tieto mahdollisesta virheestä saadaan metodin paluuarvona.

Sovelluksen alasajo tehdään siten, että kun pääohjelma saa käyttöjärjestelmältä käyttäjän lähettämänä lopetussignaalin, poistutaan pääohjelman ikisilmukasta ja astutaan toiseen silmukkaan, jossa kutsutaan jokaisen laiteolion lopetusmetodia kerrallaan. Kontrolleri- ja tietokantaolioiden lopetusmetodeja kutsutaan vielä erikseen.

5.3.4 Toimintojen suunnittelu

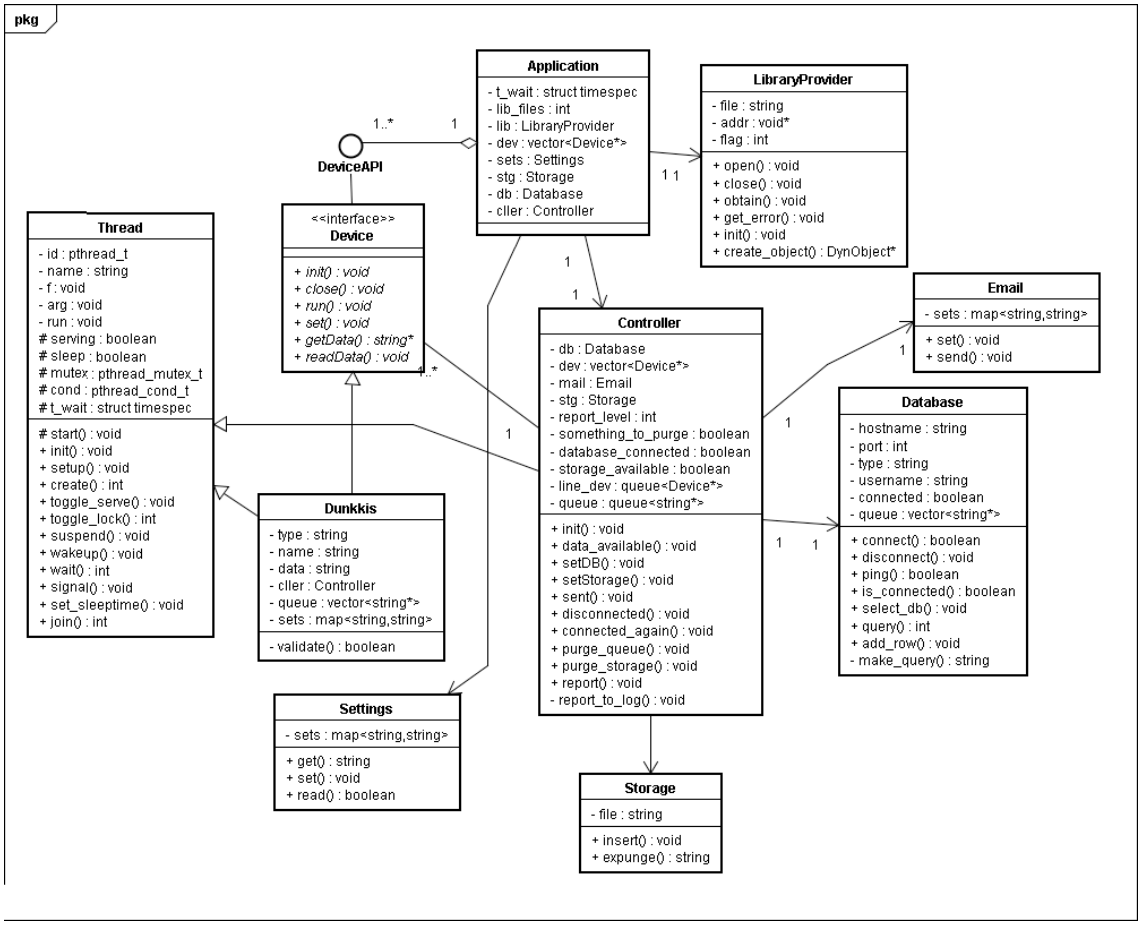
Seuraava ohjelmiston suunnittelukaavio on luokkakaavio (kuva 7), joka esittää yhden ohjelmiston kokonaisuuden luokat eli konseptit riippumatta siitä, missä tilassa sovellus on tai kuinka kauan sitä on ajettu. Ohjelmisto voi koostua useasta kokonaisuudesta eli pakkauksesta. Tämän työn sovellus on niin pieni ja yhtenäinen, että sille riittää yksi pakkaus.

Seuraavaksi selitetään tämän sovelluksen luokkakaaviota, alkaen luokasta *Application*, joka vastaa pääohjelmaa, johon siis sovelluksen käynnistyessä astutaan. Luokkakaaviossa luokan ominaisuudet eli attribuutit ovat heti luokan nimen alapuolella, ja niiden alapuolella viivalla erotettuna ovat metodit. Pääohjelmalla on siis useita muuttujia mutta ei yhtäkään metodia, koska se ei varsinaisesti ole luokka vaan yksi funktio. Muuttujat tavataan pitää mahdollisimman lyhyinä, mutta kuvaavina. Esimerkiksi merkintä `cller : Controller` merkitsee, että muuttujan cller tyyppi on luokka *Controller*. Tähän muuttujaan sijoitetaan kyseisen luokan instanssin eli olion osoite keskusmuistissa. Muuttujan ja metodin nimen vasemmalla puolella olevalla merkillä on myös merkityksensä. Viiva on merkkinä piilottamisesta luokan ulkopuolisilta luokilta, myös periviltä, plus-merkki kaikille julkisesta metodista tai periville luokille julkisesta attribuutista sekä ’#’ vain periville luokille näkyvästä attribuutista tai metodista. Metodien oikealla puolella on paluuarvon tyyppi [11, kpl. 3.1.1].

Luokkaan *Application* liitetty suunnikas merkitsee luokan *Application* koostuvan (osaksi) laiterajapinnan toteuttavista luokista. Ympyrä symboloi rajapintaa. Nuoli

kahden luokan välillä merkitsee assosiaatiota, eli pääohjelma omistaa luokan *Controller*. Valkoisella yhtenäisellä kolmiolla varustettu nuoli kuvaa yleistystä tai perintää, eli *Dunkkis* perii luokan *Thread* attribuutit ja metodit, ja toteuttaa rajapinnan *Device*, jolloin sillä on kyseisen rajapinnan metodien toteutukset. Rajapinnassa ainoastaan määritellään niiden prototyypit. Rajapintoihin ei kuulu attribuutteja. [11, kpl. 3.4.1]

Tarkkasilmäinen voi havaita, että luokassa *Thread* on attribuutti run sekä rajapinnassa *Device* metodi run(). Nämä kaksi saman nimistä symbolia eivät konfliktoi rajapinnan toteuttavassa luokassa; attribuuttiin asetetaan sen metodin muistiosoite, jossa on säikeistettävän olion tehtävää hoitava silmukka. Tämä on kätevä ratkaisu silloin kun yhdellä oliolla on kaksi hieman erilaista silmukkaa, kuten luokan *Controller* tapauksessa – attribuutin arvoksi kun voi vaihtaa toisen metodin muistiosoitteen. *Controller*in purge_queue purkaa olion sisäistä FIFO-datajonoa välittäen dataa tietokantaoliolle ja purge_storage purkaa dataa väliaikaistallennusluokka *Storage*n oliolta.



Kuva 7. Sovelluksen luokkakaavio. Rajapintaluokan **Device** *toteutusta havainnollistaa yksi luokka, tässä* **Dunkkis**.

Liitteessä 2 on luokkien tärkeimpien attribuuttien ja metodien selitykset taulukkomuodossa.

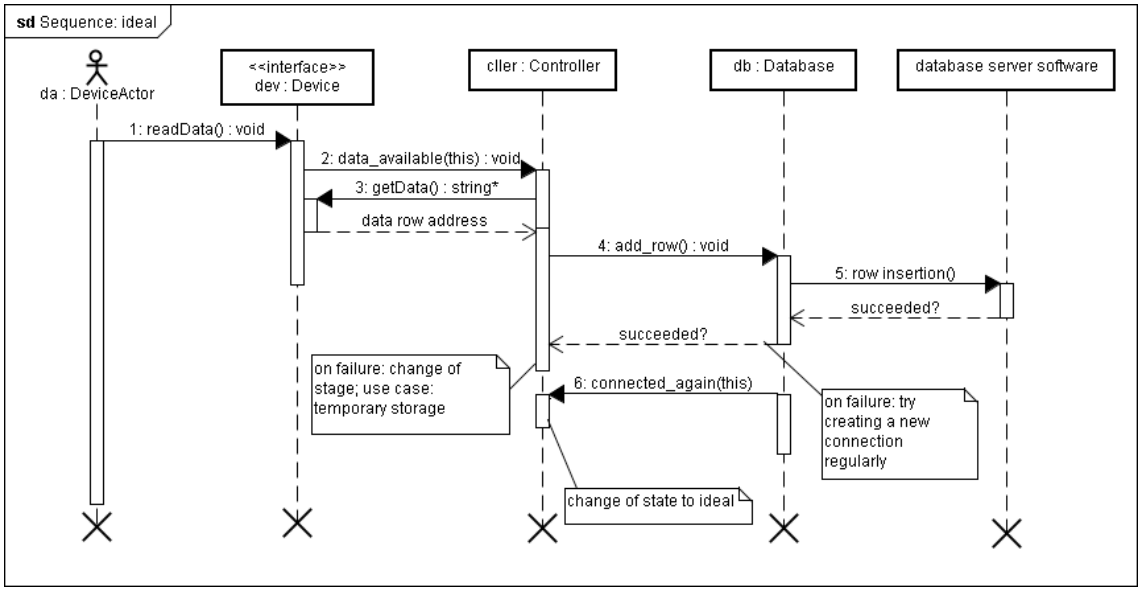
Sekvenssikaavioissa kuvataan olioiden välisiä toimintoja tapahtumaketjuina eli sitä, mitä yksi olio tekee, kun jotakin sen metodia kutsutaan joko pääohjelmasta tai toisen olion toimesta [11, kpl 6.4]. Tähänkään kaavioon ei kannata kuvata liian paljon, sillä tapahtumaketjun ymmärtäminen vaikeutuu ja kaavion vaatima alue kasvaa nopeasti.

Lähtökohtaisesti sekvenssin aloittaa aktori, mutta koska UML-kaavioissa on sallittua käyttää eri kaavioihin tarkoitettuja elementtejä muissakin kaavioissa, on tämän sovelluksen sekvenssikaavioissa käytetty sekvenssin aloittajana myös oliota. Suorakulmioihin yleensä nimetään olion tunniste sekä luokan nimi, ja niistä lähtevät pystysuuntaiset katkoviivat ovat olion "elämänviiva". Ruksi merkitsee olion

lopettamista, tosin tämän sovelluksen sekvenssikaavioissa esitettyjä olioita ei tuhota kuin vasta ohjelman lopettamiskäskyn tultua. Pitkät elämänviivan suuntaiset suorakulmiot kuvastavat olion aktiivisuutta lähtien siitä, kun se kutsuu jotakin metodia tai sen metodia kutsutaan. Elämänviivasta lähtevä kokomustan pään omaava nuoli merkitsee metodikutsua, katkoviiva vastausta siihen [11, kpl 6.4].

Kun sovellus on ideaalitilassa, eli tietokantayhteyden ollessa päällä, on sekvenssi kuvan 8 mukainen. Laiteolio kutsuu omaa readData()-metodia lukeakseen uusimmat mittauslukemat, asettaa ne omaan jonoonsa ja huomauttaa *Controller*-oliota (data_available(this)) antaen oman muistiosoitteensa. *Controller* asettaa osoitteen omaan laitepalvelujonoonsa, jossa ensimmäisenä olevalta laiteoliolta käy pyytämässä sen omassa jonossa olevat lukemat yksi kerrallaan, kunnes sen jono on tyhjä.

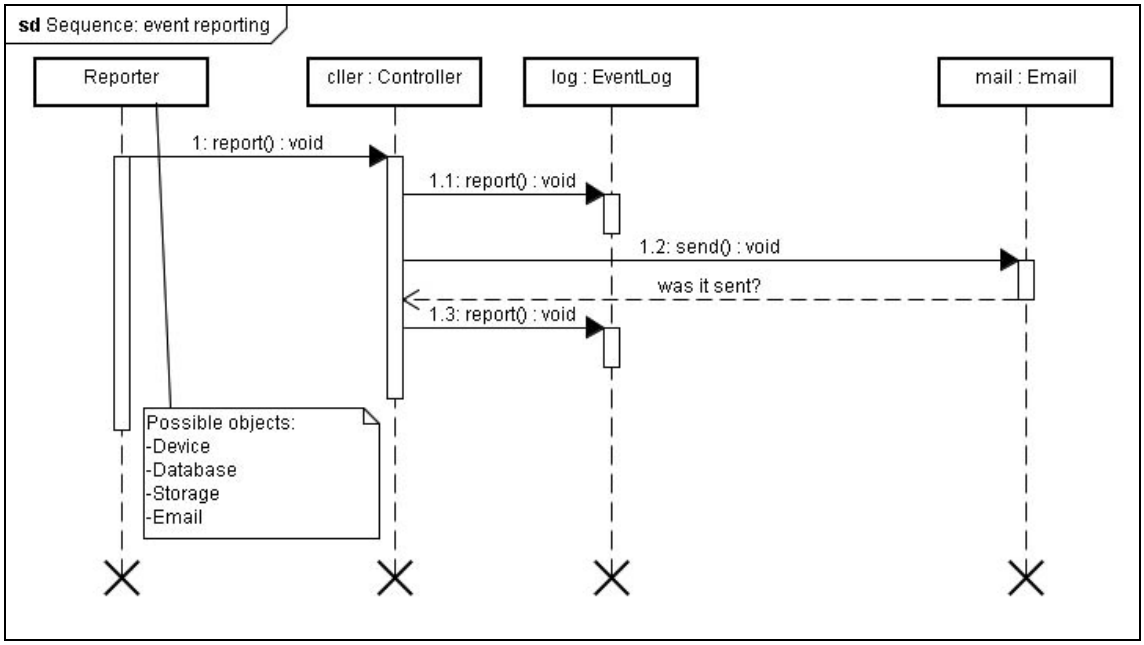
Seuraavaksi *Controller* lähtee purkamaan omaa jonoaan tietokantaolion jonoon (add_row(row)) yhden rivin muistiosoite kerrallaan. Kun tietokantaolio on tallentanut rivin onnistuneesti, se ilmoittaa siitä *Controller*ille, joka uskaltaa poistaa omasta jonostaan kyseisen muistiosoitteen. Mikäli rivin tallennus tietokantaan ei onnistu, johtuu se todennäköisesti tietoliikenneyhteyksistä ja sovelluksen tila vaihtuu.



Kuva 8. Olioiden välinen tapahtumaketju ideaalitilassa.

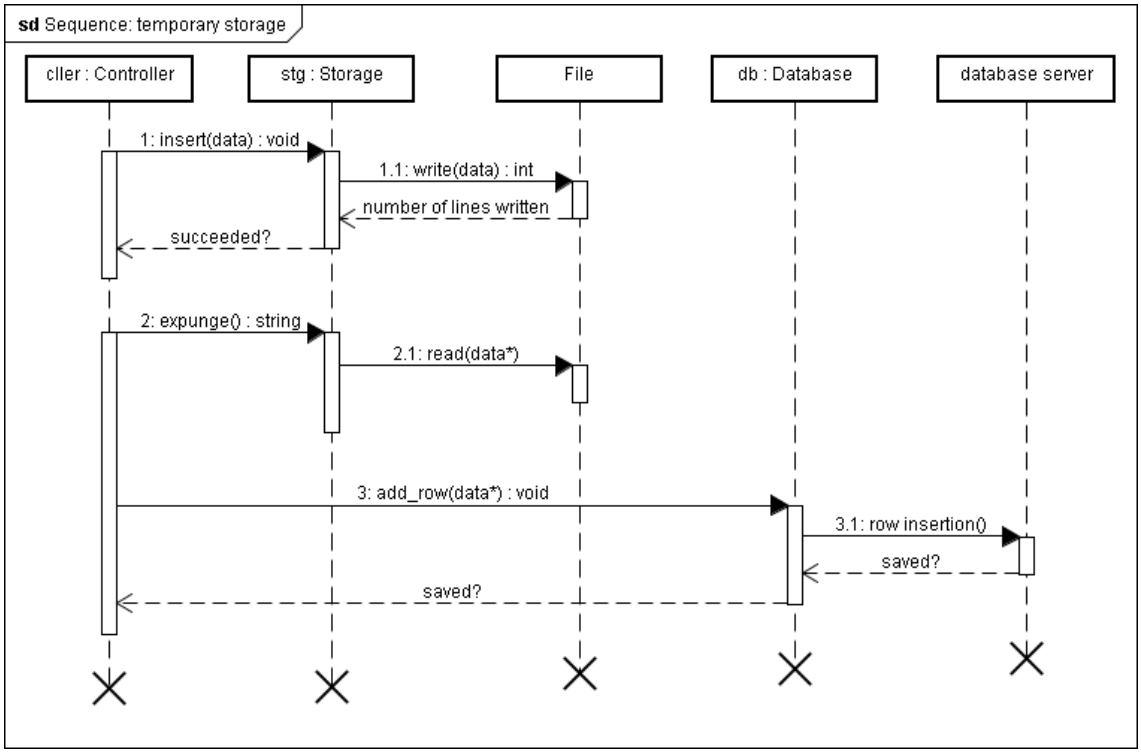
Tapahtumaraportointiketjussa raportoija voi olla pääohjelma, kontrolleri-, laite-, tietokanta-, väliaikaistallennus- tai sähköpostiolio. Nämä on kuvan 9 kaaviossa

yleistetty nimellä *Reporter*, joka kutsuu kontrollerin report()-metodia, joka lisää raportoijaolion viestiin lisätietoa ja kutsuu käyttöjärjestelmän syslog()-funktiota tallentaakseen tekstin järjestelmän tapahtumalokiin. Tämän jälkeen luokan *Email* oliota käsketään lähettämään vakavista tapahtumista sähköpostia.



Kuva 9. Olioiden välinen tapahtumaketju, kun on jotain raportoitavaa.

Väliaikaistallennusketjun tapahtumat kuvassa 10 alkavat tietokantayhteyden katketessa. *Controller* aloittaa lisäämällä *Storage*n käsittelemään väliaikaistallennustiedostoon rivin kerrallaan (insert(data)) saaden paluuarvona tiedon onnistumisesta. Poistaminen väliaikaistallennustiedostosta taas alkaa uuden tietokantayhteyden onnistuneesta luomisesta. *Controller* aloittaa taas kutsumalla *Storage*n expunge()-metodia, jolla saadaan väliaikaistallennustiedoston ensimmäinen hakematon rivi paluuarvona. *Controller* välittää rivin tietokantaoliolle, joka yrittää normaaliin tapaan tallentaa rivin tietokantaan.



Kuva 10. Olioiden välinen tapahtumaketju, kun väliaikaissäilöstä eli **Storagesta** *haetaan tai sinne lisätään dataa.*

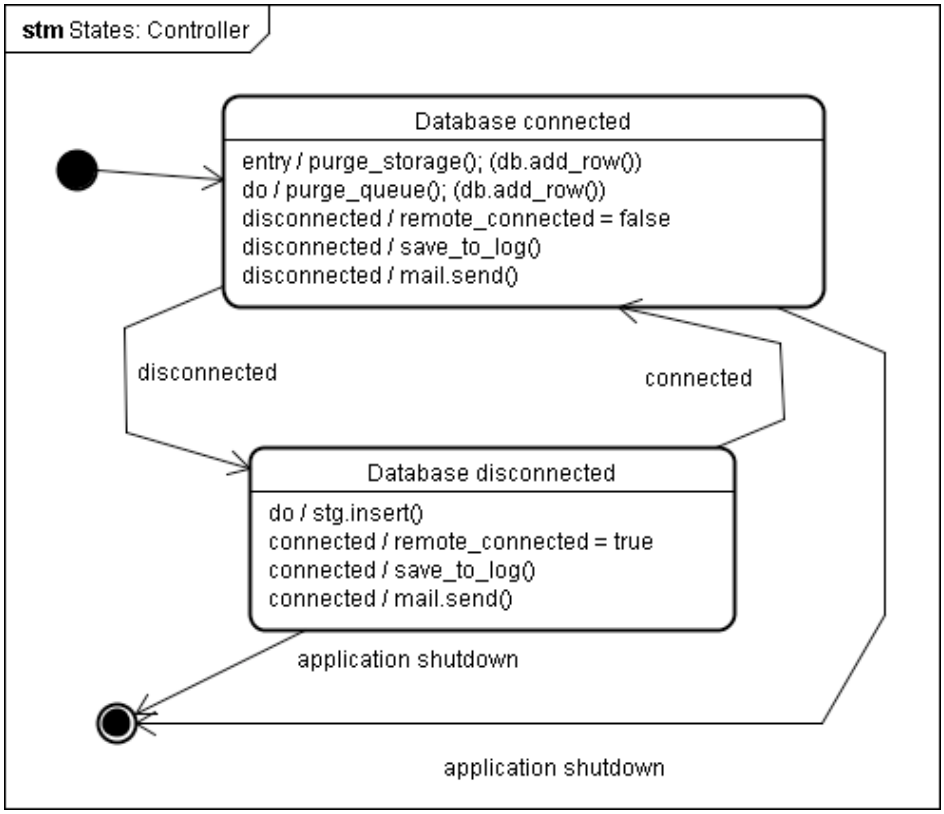
Tilakaavioissa voidaan esittää useankin olion tiloja. Tilanvaihdot voivat riippua mistä tahansa ehdosta, minkä suunnittelija on päättänyt. Tilalla on nimi, operaatioita jotka suoritetaan tilaan tultaessa, siinä oltaessa sekä siitä poistuttaessa [11, kpl 7.1]. Tilan käsite on häilyvä. Periaatteessa tila voi vaihtua tietyn taulukon pituuden funktiona.

Sovellus sekä kontrolleri ovat lähtökohtaisesti ideaalitilassa, kuvassa 11 esitettävässä 'Database connected'-tilassa. Sovelluksen käynnistyessä tarkistetaan, onko väliaikaistallennustiedostossa dataa. Mikäli dataa on, lähdetään sitä purkamaan tietokantaoliota kohden ('entry'-rivi). Tuon väliaikaistallennustiedoston tyhjennyttyä puretaan dataa jonosta taasen tietokantaoliota kohden ('do'-rivi). Tietokantayhteyden katketessa merkitään yhteys kontrollerissa katkenneeksi, tallennetaan tapahtumalokiin ilmoitus sekä lähetetään sähköpostiviesti, vaikka sekään ei välttämättä pääse perille ('disconnected'-rivit).

Tietokantayhteyden ollessa poikki vie *Controller* datajonostaan rivejä *Storage*lle väliaikaisesti tallennettavaksi. Tilasta poistuttaessa kontrolleri merkitsee tieto-

kantayhteyden toimivaksi, raportoi asiasta tapahtumalokiin sekä lähettää sähköposti-viestin.

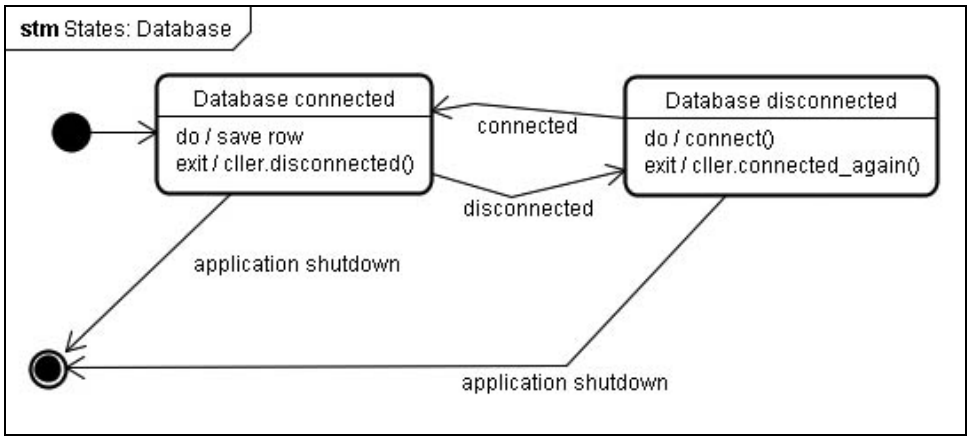
Sovelluksen saatua lopettamissignaalin suoritetaan vain tavanomaiset muistinvapauttamisoperaatiot.



Kuva 11. Kontrolleriolion tilat ja toiminnot riippuvat tietokantayhteyden olemassaolosta.

Tietokantaolion tilat jäljittelevät kontrolleriolion tiloja. Tilat on esitetty graafisessa muodossa kuvassa 12. Tietokantayhteyden ollessa päällä olio purkaa omaa jonoansa luoden SQL-lausekkeita ja lähettäen niitä tietokantapalvelinohjelmistolle. Tilasta poistuttaessa se kutsuu kontrolleriolion disconnected()-metodia ilmoittaakseen yhteyden katkeamisesta.

Yhteyden ollessa poikki yritetään uutta yhteyttä luoda säännöllisesti. Tilasta poistuttaessa tietokantaolio kutsuu kontrollerin connected_again()-metodia, joka toimii ilmoituksena kontrollerille siitä, että uusi yhteys on luotu onnistuneesti. Sovelluksen saadessa lopettamissignaalin ei mitään erityisiä toimenpiteitä tehdä.



Kuva 12. Myös tietokantaolion tilat ja toiminnot riippuvat tietokantayhteyden olemassaolosta.

6 OpenWrt:n asentaminen reitittimelle

Koska reitittimiä ei ole suunniteltu käytettäväksi suoraan näppäimistöllä tai hiirellä, eikä niihin saa DVD- tai CD-asemaa kiinni, on uuden käyttöjärjestelmän asentaminen reitittimelle erikoisempi proseduuri kuin PC-tietokoneelle. Markkinoilla kyllä on USB-liitäntään liitettäviä näppäimistöjä, hiiriä sekä optisia asemia, mutta tällöinkin on ongelmana se, mistä käyttäjä näkisi, mitä dataa reitittimelle syöttää, kun reitittimeen ei saa näyttöä eikä sarjakaapelia kiinni. Reitittimeen tehtaalla asennetun ohjelmistonkin pitäisi moisia laitteita tukea.

6.1 Levykuvatiedosto

Levykuvatiedosto on tiedosto, johon on paketoitu erityisellä algoritmilla tiedostojärjestelmä, jota jokin ohjelmisto osaa lukea. Valmiin ohjelmiston – tässä OpenWrt – tapauksessa on ensin valmisteltu sisältö, joka edelleen on paketoitu valmiiksi osioksi. Vastaavasti osiosta, johon on asennettu esimerkiksi Microsoft Windows, voidaan ottaa levykuva ja siirtää se toiseen tietokoneeseen. Windows-käyttöjärjestelmän kohdalla on kuitenkin sellainen poikkeus, että kohdetietokoneen laitteiston tulee olla lähes, ellei täysin, sama kuin lähdetietokoneen, jotta käyttöjärjestelmä ylipäänsä toimisi.

OpenWrt:n kehittäjät ovat tuottaneet useita levykuvatiedostoja, ja tässä työssä käytetylle reitittimelle käyttöjärjestelmän uusinta versiota on kaksi – asennuspäivien aikoihin uusin oli nimeltään kamikaze, versioltaan 8.09.2. Mainittuun levykuvatiedostoon päädyttiin siksi, että siinä toimivat kaikki asiat valmiiksi ilman käyttäjän tekemää erillistä konfigurointia. Tässä tiedostossa tulee mukana versio 2.4.35.4 Linuxin ytimestä, kun vaihtoehtoisessa levykuvatiedostossa on versio 2.6 jonka yksi etu olisi huomattavasti suurempi tehokkuus.

OpenWrt:n kehitysryhmä on mitä todennäköisimmin tuottanut levykuvatiedoston sitä varten kehitetyllä automatisointipaketilla, OpenWrt-ImageBuilderilla kääntäen ensin välttämättömimmät ohjelmistot ohjelmistokehitystyöpaketin eli SDK:n puolella. Projektityöryhmämme ei ImageBuilderia ole tämänkään dokumentin kirjoituspäivään

mennessä käyttänyt yksinkertaisesti siksi, ettei ole ollut tarvetta. ImageBuilder on kätevä silloin kun halutaan tuottaa mukautettu levykuvatiedosto, jonka reitittimelle siirtämisen jälkeen vaatii mahdollisimman vähän ylimääräistä konfiguraatiota. Muita etuja ovat helppo siirrettävyys asiaan perehtyneiden tai asiasta kiinnostuneiden henkilöiden välillä sekä helppo tapa tehdä varmuuskopioita. [15; 16]

Tässä työssä kehitetty sovellus olisi voitu viedä reitittimelle levykuvatiedostoon sisällytettynä, mutta kehityksen ollessa vielä kesken riitti sovelluksen lataus SSH-protokollalla siirtämällä. Levykuvatiedoston siirtäminen ei muutenkaan ole asia, jota kannattaa tehdä usein, sillä flash-muistipiireillä on jokin kirjoituskertaraja – satoja tuhansia – ennen kuin ne menevät käyttökelvottomiksi. Myös joka kerta kun tiedoston siirtää, uskaltaa seuraavan uudelleenkäynnistyksen tehdä yhä useamman minuutin päästä, eikä tarkkaa odotusaikaa voi tietää [15; 17].

6.2 Asennuksen valmistelu

Uuden käyttöjärjestelmän asennus ainakin valitulle Asuksen reitittimelle käy muutamalla eri tavalla, mutta yksinkertaisimmaksi ja toimivaksikin osoittautui valmiin levykuvatiedoston siirtäminen TCP/IP-verkon välityksellä PC-tietokoneesta TFTP-protokollaa käyttävällä sovelluksella – Microsoft Windows 2000- sekä XP -pohjaisilta työasemilta sellainen käynnistetään komentokehoitteessa komennolla ’tftp’.
Asennus vaatii seuraavia valmisteluja. Työaseman, jolta levykuvatiedosto siirretään, sekä reitittimen on oltava samassa aliverkossa sekä yhdistetty yhdellä ethernet-kaapelilla toisiinsa. Kytkimen tai toistimen käyttö on myöskin sallittua.

Reitittimen pysyväistallennusmuistipiiriin on tallennettu useita asetuksia, joita pystyy ainakin OpenWrt:hen kirjauduttua muuttamaan. Tehtaalla reitittimen sisäisen kytkimen oletus-IP-osoitteeksi on asetettu 192.168.1.1. Kun OpenWrt on käynnistynyt onnistuneesti, on se saattanut vaihtaa osoitteen sellaiseksi, kuin se on asetustiedostossa määrätty. OpenWrt:n ensimmäisen asentamisen valmistelussa täytyy huomioida, että reitittimen IP-osoite on 192.168.1.1 ja verkkomaski 255.255.255.0, jolloin työaseman reitittimeen liitettävän verkkokortin verkkomaskiksi on laitettava sama osoite, sekä IP-

osoitteeksi vaikka 192.168.1.2 tai sellainen, jossa ensimmäiset kolme lukua ovat samoja ja viimeinen luku on välillä 2..254 [17].

6.3 Asennuksen suoritus Microsoft Windows -työasemalla

Avataan komentokehote, johon kirjoitetaan valmiiksi komento `tftp –i 192.168.1.1 put openwrt-brcm-2.4-squashfs.trx`. Levytiedosto on osoitteessa http://downloads.openwrt.org/kamikaze/8.09.2/brcm-2.4/openwrt-brcm-2.4-squashfs.trx. Avataan toinen komentokehote, johon annetaan ja käynnistetään komento `ping –t –w 100 192.168.1.1`, jotta myöhemmin nähdään oikea hetki käynnistää tftp.

Mikäli reitittimessä on virtajohto kiinni, otetaan se muutamaksi sekunniksi irti ja kytketään takaisin. Reititin käynnistyy automaattisesti uudestaan. ”Pingaus”-ikkunassa tulee vastaukseksi usea ”Hardware error”-virheilmoitus. Kun vastauksena alkaa tulla normaalin näköisiä vasteaikoja, annetaan pikaisesti tftp-komento. Levytiedoston siirrossa kestää korkeintaan muutama sekunti. Laitteessa tulee pitää virrat kytkettynä vähintään kolme minuuttia, suositellusti enemmän. Reitittimessä alkaa tiedoston kirjoitus massamuistiin [17] eli "flashays". Mitä useammin "flashays" tehdään, sitä useampi minuutti pitää odottaa ennen kuin reitittimen voi käynnistää uudelleen [17].

OpenWrt on nyt asennettu, ja Linux-järjestelmään pääsee sisälle komentamalla komentokehoitteessa `telnet 192.168.1.1`. Turvallisempi etähallintatapa on SSH, joka otetaan käyttöön Telnetin sijasta asettamalla root-tunnukselle salasana (tätä ennen tunnuksella ei ole salasanaa, eikä järjestelmässä ole muita tunnuksia). Järjestelmää pystyy nyt käyttämään ja asetuksia muuttamaan, kuten parhaakseen näkee.

6.4 Lisätietoja reitittimen etähallinnasta

Luvun alussa kerrottiin, että reitittimeen on mahdotonta liittää näyttölaite. Linux-jakeluissa on tähän yksi ratkaisu; X Window System eli Unix-pohjaisille järjestelmille kehitetty graafisten käyttöliittymien hallintaohjelmisto. Reititintä pystyy sen avulla käyttämään graafisessa ympäristössä verkon välityksellä, mutta uuden levykuvatiedos-

ton siirtämistä se ei juuri helpota. Käyttöjärjestelmän ohjelmiston pystyy kyllä päivittämään graafisessa käyttöliittymässä sekä käyttöjärjestelmän komentokehoitteessa.

7 Sovelluksen tuottaminen reitittimelle

7.1 Suunnittelu- ja toteutusohjelmistot

Sovelluksen UML-kaaviot luotiin asiaan varta vasten kehitetyllä astah* community -editorilla ja tietokantarakenne Microsoft Visio 2003 -kaavioeditorilla. Koodia kirjoitettiin aluksi NetBeans versio 6:lla, mutta siitä oli luovuttava, koska sovelluksen kirjoittaminen suoritettiin fyysisesti eri paikassa kuin missä koodin sisältämä tietokone oli. Koodin piti sijaita OpenWrt-SDK-kehitystyökalupaketin alahakemistossa, ja SDK:n Linux-järjestelmän juureen asennettuna. Sovellus kirjoitettiin lähes kokonaan terminaalissa käytettävällä tekstieditori Vim:llä.

7.2 Kehitystyökalupaketin merkitys

Jotta OpenWrt-järjestelmään saa tuotettua omia sovelluksia C/C++-kielisestä lähdekoodista, on paras vaihtoehto asentaa jollekin PC-tietokoneelle OpenWrt SDK [14; 15]. Käyttöjärjestelmänä tietokoneessa on oltava jokin Linux-jakelupaketti – tässä työssä SDK asennettiin CentOS 5.4 -järjestelmään, jota ajetaan 64-bittisen x86-suorittimen voimalla. Koska SDK:n asennus on suositeltu tehtäväksi juurihakemistoon, ei sen asennus ollut mahdollista Metropolian Linux-palvelimille joten ainut vaihtoehto oli asentaa se kotikoneelle.

OpenWrt-SDK on paketti, jonka mukana tulee automatisoidut työkalut tämän Linux-jakelun kaikkien perussovellusten lataamiseen Internetistä sekä kääntämiseen (ainakin tämän työn reitittimen tapauksessa) MIPS-arkkitehtuurille. Oman sovelluksen tuottaminen lähdekoodista suoritettavaan muotoon noudattaa samoja työvaiheita ja sääntöjä kuin muiden kehittämien sovellusten. Jakelun Internet-sivuilta löytää ohjeen, kuinka automatisointia käytetään hyväksi [15; 18].

Tässä SDK:ssa tulee mukana myös valmiiksi käännetyt ohjelmistokirjastotiedostot, joita tarvitaan sovellusta käännettäessä MIPS-arkkitehtuurin konekielelle. Tällaista koodin kääntämistä toiselle suoritinarkkitehtuurille kutsutaan ristiinkääntämiseksi. Jotta ristiinkäännöksen voisi tehdä, on kääntötyökalut oltava ensin käännetty omista

lähdekoodeistaan sellaisiksi, että ne kääntävät ohjelmakoodin halutulle suoritinarkkitehtuurille. SDK:ssa kääntötyökalut ovat jo valmiiksi tällaisia. [15; 18; 19]

7.3 Ohjelmointivirheiden korjaus

Sovelluksen toteuttamisessa haluttiin laadun kannalta olla niin tiukkoja, ettei C-kääntäjä antaisi edes varoituksia. Tähän päästiin, mutta se ei ole vielä mikään tae sovelluksen virheettömyydestä. Ensimmäisinä prototyypinkin ajokertoina ihmetystä aiheutti, mistä virheilmoitus ”Segmentation fault” saattoi tulla, vaikka koodi tarkistettiin monta kertaa. Kyseinen virheilmoitus kertoo, että sovellus yritti varata muistia kielletyltä alueelta. Vastuu on toki koodin kirjoittajalla, mutta tarkoitus ei ollut moista laittomuutta tehdä.

Alkeellisin virheenetsintätapa on käskeä sovellusta tulostamaan satunnaisissa ja tärkeissä kohdissa joidenkin käytettyjen muuttujien arvoja, jotta virheen aiheuttavat kohdat saisi haarukoitua. Tässä vaiheessa apua annettiin Linux-keskeiseltä IRC-keskustelukanavalta kehottaen käyttämään virheenhavaitsemis- ja profilointityökalu Valgrindia [20], joka havainnollisti varsin hyvin ja auttoi merkittävästi virheiden havaitsemisessa listaten funktiokutsuketjun ja sen, millä riveillä missäkin tiedostossa yritetään varata muistia kielletyltä alueelta tai sijoittaa tietoa sellaiselle.

7.4 Lopputulos

Sovelluksen virheet saatiin lopulta korjattua siihen pisteeseen asti, että dataa olisi välitetty tietokantamoduulille. Tässä vaiheessa ryhmän kesken päätettiin tietokantaolio korvata moduulilla, joka siirtäisi datan SSH-protokollalla Metropolian palvelimelle, sillä se on geneerisempi tapa eikä ota kantaa tietokannan rakenteeseen. Noin kaksi viikkoa päätöksestä sovelluksen kehityksen teki tarpeettomaksi päätös olla toteuttamatta ModBus/TCP-moduulia Saberia varten. Tässä dokumentissa kerrotaan sovelluksesta sellaisena, että siinä on tietokantamoduuli SSH-moduulin sijasta, mikä on oma ratkaisuni.

Ihmetystä kääntämisen jälkeen aiheutti myös sovelluksen binääritiedoston huomattava koko eli tiedoston, josta sovellus käynnistetään. Vertailukohteeksi voi ottaa vaikka

SDK:n kanssa samaan Linux-järjestelmään asennetun Apache-projektin http-palvelinsovellus versio 2:n, jonka vastaava binääritiedosto on kooltaan 323 kilotavua ja 68 moduulitiedostoa yhteensä 1,7 megatavua. Tässä työssä kehitetyn sovelluksen MIPS-arkkitehtuurin binääri on kooltaan 926 kt ja x86-arkkitehtuurin 938 kt. Ero ei siis selity suorittimien käskykantojen eroilla. Ero johtunee siitä, HTTP-palvelinsovellus on kirjoitettu kokonaan C-kielellä, ja tämän työn sovellus suurimmaksi osaksi C++-kielellä, jossa luokkien mukanaolo tuo paljon lisää symboleja. C++ kuitenkin helpottaa ja nopeuttaa ohjelmistokehitystä ja tuli osin siksi valituksi.

8 Tietokannan rakenne

8.1 Tietokannan tarkoitus

Metropolian järjestelmä kerää mittausdataa, ja jotta voisi tarjota palvelun perustuen mitattuihin lukemiin on data tallennettava johonkin ja se on jäsenneltävä. Tietokantoja voi olla erilaisia, mutta niiden perusominaisuus on sisältää tietoa. Metropolian järjestelmään tarvittiin SQL-relaatiotietokanta. Relaatiotietokannasta saa suurimman hyödyn ja nopeuden, kun tiedon jäsentää joukkoihin, joita voi yhdistää toisiinsa vähintään yhdellä yhteisellä tiedolla. Pitkälle viety jäsentäminen sekä yhdisteleminen ehkäisevät myös massamuistin liiallista varausta, kun toistuvaa dataa esiintyy mahdollisimman vähän.

Tiedot (joukot) tallennetaan tietokannassa tauluihin, joissa tiedot ovat yksittäisinä yksilöllisinä ja siten tunnistettavissa olevina riveinä eli tietueina. Tiedoille suunnitellaan annetun yhtenäisen joukon piirteitä mahdollisimman hyvin vastaava rakenne. Yleensä tauluissa on

- vähintään yksi tietueen yksilöivä tieto
- tieto joka sitoo (yhteys) tietueen tauluhierarkiassa ylempänä olevaan tai varsinaista dataa täydentävään tietueeseen
- varsinainen data, kuten lämpötilalukema sekä
- ajankohta, jolloin tieto on mitattu, tallennettu, viimeksi päivitetty tai mikä ikinä kiinnostaa.

Tämän järjestelmän relaatiotietokannan tiedot on konkreettisesti tallennettu kiintolevylle tiedostoihin, joita ei kenenkään kannata lähteä muokkaamaan, vaan niiden sisältöä käsitellään asiaan tarkoitetuilla työkaluilla.

8.2 Palvelun vaatimukset tietokannan suhteen

Palvelussa halutaan tarkastella kohdekiinteistöistä saatuja mittalukemia sekä näiden pohjalta laskettuja indikaattoreita että kuvaajia. Kouluissa on useita mittausjärjestelmiä, joiden mittaama data pitää yhdistää, sillä yhdestä järjestelmästä ei saada kaikkea

haluttua tietoa. Tarpeen on myös tietää, missä mikäkin laite ja anturi sijaitsevat. Tärkeintä tallennettavaa on järjestelmistä saatavat mittaustulokset, mutta ellei yksittäisiä lämpötilalukemia, noin esimerkiksi, yhdistetä mihinkään tiettyyn laitteeseen tietokannassa, ei voida tietää, mistä laitteesta lämpötila on mitattu. Kun anturin sijainti taas on tietokannassa sidottu tiettyyn fyysiseen tilaan, tiedetään, mistä tilasta lukemat tulevat. Edelleen kun anturi liitetään laitteeseen, laite rakennukseen ja rakennus kiinteistöön, voidaan ajan myötä esittää vaikka reaaliaikainen ja nopeaan tahtiin päivittyvä lämpötilakuvaaja tietyn kiinteistön tietystä tilasta olettaen, että mittaustulokset saapuvat kuvaajan tarjoavalle palvelimelle muutaman sekunnin aikana tuloksen ensikirjauksen ajankohdasta lähtien.

8.3 Vaatimuksiin sopiva rakenne

Koska kyseessä on useampi kuin yksi kiinteistö, joista kerätään dataa, tulee kiinteistölle tehdä oma taulunsa. Tietokantapalvelinsovellus määrätään asettamaan ilman erinäistä käskyä jokaiselle kiinteistölle oma tunnistenumero (EstateID), joka inkrementoituu yhdellä jokaista tietuetta kohden. Käyttäjän antamia tietoja ovat nimi (Name) sekä vähemmän tarpeelliset katuosoite (StreetAddress) ja kaupunki (City). Kaupunginkin mainitsemisesta on hyötyä lähinnä kehittäjille siten, että tauluun voi tallentaa muissa kaupungeissa sijaitsevia kiinteistöjä välttäen sekaannusta. Tämän järjestelmän kehitystyö ei tapahtunut Helsingin rajojen sisällä, joten taulussa on mukana myös Metropolian Myyrmäen toimipiste.

Seuraava askel on listata kiinteistöön lukeutuvat rakennukset; esimerkiksi Ala-Malmin peruskoulu käsittää usean erillisen rakennuksen samassa osoitteessa. Vielä yleisempi ratkaisu akselilla rakennus – kiinteistö – kaupunki olisi esitellä tietokannalle organisaation käsite, johon lukeutuisi yksilöivä tunniste sekä nimi. Tällöin rakennus lukeutuisi kiinteistöön, jonka tiedoissa mainittaisiin katuosoite, kaupunki ja organisaation tunniste. Rakennukselle riittäisi ilman muita vaatimuksia yksilöivä tunniste, nimi sekä mihin kiinteistöön se kuuluu. Tästä ratkaisusta vaan ei tässä pilottiprojektissa ole hyötyä eikä siten tarvetta.

Tässä ratkaisussa rakennuksillekin asetetaan oma juokseva tunnistenumero (BuildingID) sekä kiinteistön tunnistenumero (EstateID), jonka perusteella tiedetään, mihin kiinteistöön rakennus lukeutuu. Rakennuksen nimi (Name) on suositeltava antaa, jotta käyttäjä tietää, mikä rakennus on kyseessä. Rakennuksissa taas on tiloja, jotka kaipaavat yksilöivän tunnisteen (RoomID), nimen (Name) sekä rakennustunnisteen (BuildingID).

Fyysiset laitteet esitetään asemina (stations). Aseman käsite tarkoittaa laitetta, kuten reititin, johon anturit on kiinnitetty ja joka välittää eteenpäin anturien tarjoaman tiedon. Asemallekin annetaan yksilöivä tunniste (StationID), sanallinen kuvaus (Description) sekä tilan tunniste (RoomID). Koska laitteet ovat erilaisia ja yksilöllinen täydentävä tieto on hyödyksi, esitellään infotaulu (station_info), johon annetaan aseman tunniste, vapaamuotoinen sanallinen attribuutti sekä sen arvo, joka voi olla sanallinen tai numeraalinen.

Anturit ovat jo niin alhaisen tason objekteja, että on pitkälti järjestelmäkohtaista, minkälainen niiden tunniste on ja miten se saadaan. Dunkkis-tuotteessa antureiden tunnistaminen on ratkaistu integroimalla Dunkkis-piirilevyyn sarjanumeron sisältäviä komponentteja (koodiltaan DS2438), joiden "vain luku" -muistipiiriin on tehtaalla tallennettu 1-Wire-teknologian mukainen 64 bittiä pitkä rekisteröintinumero, joka komponentin teknisen dokumentin mukaan [21] takaa, että jokaisella komponentilla on yksilöllinen numero. Kutakin anturia kohden on yksi tällainen komponentti. Komponentissa on myös pysyväistallennusmuistipiiri (non-volatile), johon voi kirjoittaa kahdeksan kappaletta kahdeksan tavun pituista tietoa, joita piirin dokumentaatiossa kutsutaan sivuiksi. Seuraavassa listauksessa jokaisen sivun käyttötarkoitus.

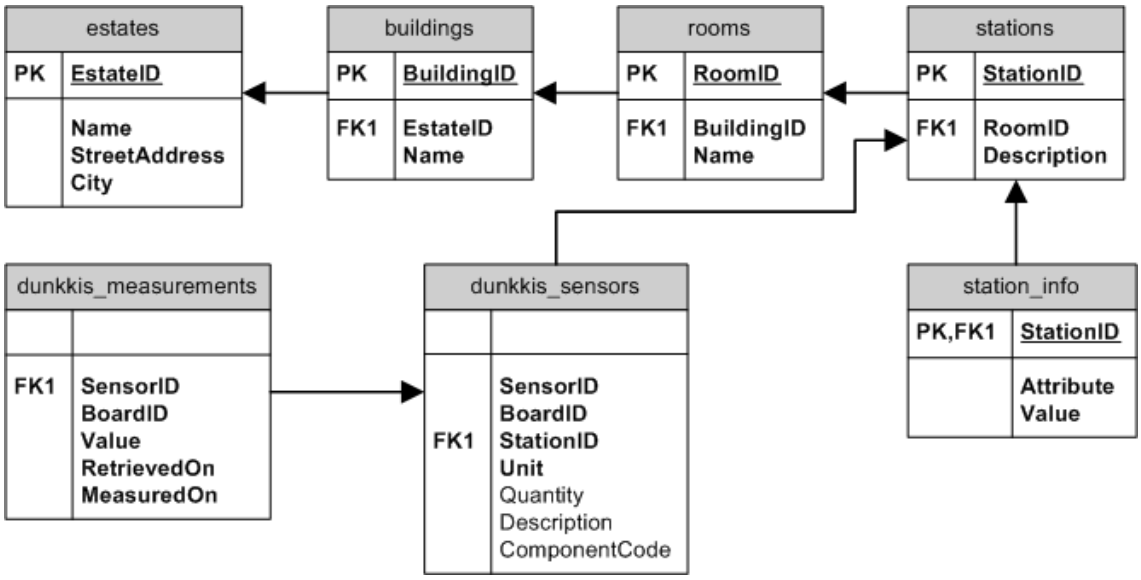
- Sivut 1 ja 2 ovat laitteen sisäisessä toiminnassa käytettäviä rekistereitä, joiden sisältöä ei kannata mennä muokkaamaan.
- Sivulle 3 uusien antureiden alustustyökalu kirjoittaa merkkijonon ”Dunkkis2” kertoakseen, että nyt on kyse version 2 Dunkkis-piirilevystä.
- Sivulle 4 työkalu kirjoittaa piirilevyn tunnisteen, joka on itseasiassa sama kuin ensimmäisen sarjanumeropiirin sisältämä tieto, jonka työkalu sattuu lukemaan.
- Sivulle 5 työkalu kirjoittaa muistipiirin läheisen anturin komponenttikoodin.

- Sivu 6 jää tyhjäksi.
- Sivulle 7 muistipiiri kirjoittaa tietyssä tapauksessa virrankulutukseensa liittyvää tietoa [21; 22].

Dunkkis-antureiden tunnistamiseen käytetään siis sekä anturin tunnistetta että piirilevyn tunnistetta. 1-Wire -ohjelmistokirjasto antaa tunnisteet eteenpäin heksadesimaalimuodossa merkkijonoina. Anturitauluun merkitään siis nämä molemmat (SensorID, BoardID), aseman tunniste (StationID), anturin mittaaman lukeman yksikkö (Unit), valinnaisesti sen suure (Quantity), kuvaus (Description) ja komponenttikoodi (ComponentCode).

Viimeisenä on varsinaisen mittausdatan sisältävä taulu, johon tallennetaan myös anturin ja piirilevyn tunnisteet, mitattu lukema (Value), tietokantaantallennusajankohta (RetrievedOn) sekä ajankohta reitittimellä, jona mittauslukema haettiin (MeasuredOn).

Kuvassa 13 näkyvät merkinnät PK, FK1 sekä lihavoinnit ovat Microsoft Visiolle tyypillisiä mutta riippumattomia tietokantapalvelinsovelluksesta. Merkinnät kertovat kyseisen kentän arvojen olevan ensisijaisia ja uniikkeja sekä toimivan indeksinä (PK, Primary Key) tai viittaavan toisen taulun vastaavaan kenttään (FK, Foreign Key). Lihavointi merkitsee, ettei kenttään hyväksytä tyhjää arvoa.



Kuva 13. Täysin kelvollinen tietokannan rakenne, tosin laitteista vain **Dunkkis** *huomioituna.*

Kuvassa ei ole Saber-mittalaitteelle tauluja, koska mukana tulleissa dokumenteissa ei mainita mitään, minkä perusteella voisi suunnitella taulurakenteen, eikä lukutukea laitetta varten ehditty edes aloittaa.

Lähtökohta mittaustietotaulurakenteen suunnittelussa on se, että on jokin mittauspisteen tunniste, itse arvo sekä tieto siitä, milloin mittaus on tehty. Mikäli mittaus tehdään muualla, kuin mihin sen tulos tallennetaan tai toisen laitteen toimesta, voi samaan tietueeseen liittää myös kentän, johon asetetaan mittauksen tallennusajankohta. Mikäli aikaero näiden kahden eri ajankohdan välillä on huomattava, kuten minuutteja, voi päätellä tietoliikenneyhteyksissä olleen jotain vikaa.

Riippuen mittauslaitteesta tai -järjestelmästä mittauspistetaulun (kuvassa "dunkkis_sensors") muoto voi vaihdella. Lähtökohta tässäkin on, että mainitaan mittauspiste sekä sen, mihin ylemmän tason objektiin se lukeutuu. Dunkkiksen kohdalla piirilevy lukeutuu asemaan, joka on reititin, johon se on kytketty. Myös yksikön ja suureen mainitseminen ovat tärkeitä mainita, ja tähän tauluun luotuina ne säästävät tilaa sen sijasta, että jompikumpi mainittaisiin mittausdatataulussa jokaisen mittauslukeman kohdalla.

Näin ollen Saberin kohdalla mittauspistetauluun voisi tallentaa IP-osoitteen ja TCP-portin, joista se vastaa pyyntöihin jonkin reitittimen (StationID) takaa. IP-osoitteelle tarvitaan uusi, asianmukainen kenttä. Mittauspistetunnisteena osoitetta voisi toki käyttää tietyin varauksin; on oltava varma, ettei kahden eri laitteen mittaukset sekoitu keskenään, jolloin ei voitaisi sanoa taulun sisältöä lukemalla, mikä mittaus tulee mistäkin.

Tähän on useita ratkaisuja. Oletetaan, että kahdella Saberilla on sama IP-osoite. Sekaantumisen välttämiseksi voidaan valita ainakin kahden vaihtoehdon välillä. Sovelluksen lähettäessä dataa on sen merkittävä jokaiseen Saberin mittalukema-tietueeseen StationID:n arvo, joka on sovelluksen asetustiedostoon annettu ja molempiin, sekä mittaustulostauluun että mittauspistetauluun, IP-osoite sekä TCP-portti

että aseman tunniste, tai mittaustulostauluun mittauspistetunniste, muotoa etumerkitön kokonaisluku sekä mittauspistetauluun IP-osoite, TCP-portti ja aseman tunniste, jolloin mittauspistetaulussa on avainkenttänä mittauspistetunniste.

Jälkimmäinen ratkaisu säästää massamuistitilaa ja on tietokantaoppien ihanteiden mukainen välttämällä toistuvan datan ilmenemistä. Koska IP-osoitetta käytetään osana tunnistusta, on oltava varma myös siitä, ettei se vaihdu hallitsemattomasti. Se pitää siis asettaa joko manuaalisesti tai määräämällä (reitittimen) DHCP-palvelun antamaan tietylle MAC-osoitteelle tietty IP-osoite joka kerta.

9 Yhteenveto

Insinöörityössä suunniteltiin ja toteutettiin toimiva sovellus, joka lukee mittalaitteen tarjoaman datan, validoi sekä lähettää sen SQL-lausekemuodossa toisessa kiinteistössä sijaitsevalle tietokantapalvelimelle. Sovellus raportoi informatiivisista tapahtumista, varoituksista sekä vakavista virheistä järjestelmälokiin sekä sähköpostitse. Tietyn tilanteen tullen sovellus alkaa tallentaa massamuistiin dataa, jota ei varmuudella ole saatu tallennettua tietokantaan. Sovelluksen tarkoitus oli olla siltana laitteiden ja tietokannan välissä siirtäen suurimmaksi osaksi laitteiden tekemiä huoneilmastomittauslukemia säännöllisin väliajoin.

Sovellus suunniteltiin astah* community UML -suunnittelutyökalulla, kirjoitettiin Linux-terminaalissa käytettävällä Vim-tekstieditorilla ja toteutettiin C/C++-ohjelmointikielellä. Ohjelmointivirheiden etsinnässä käytettiin apuna Valgrindia. Tietokantarakenne piirrettiin Microsoft Visio 2003 –kaavioeditorilla.

Loppua kohden mieleen hiipi taas epäily, että kannattaisiko sovellus sittenkin toteuttaa komentosarjana. Sovellukseen kun näytti olevan tarvetta tehdä linkitys moneen ylimääräiseen ohjelmakirjastoon venyäkseen kaikkeen haluttuun paisuten muodottomaksi. Suurin syy ohjelmointikielen käyttöön oli KYAB Saberin käyttäminen ModBus/TCP-protokollalla. Mikäli projektin aloittaisi alusta, tekisi asiat eri tavalla. Oletetaan, että pitäisi toteuttaa tuki usealle laitteelle, joille ei ole olemassa spesifistä sovellusta mutta rajapinta olemassa olevalle protokollalle. Yksinkertaisin, viisain sekä nopein ratkaisu olisi kirjoittaa pieniä laitekohtaisia C-kielisiä sovelluksia, joilla datan saa luettua keskus- tai massamuistiin ja josta komentosarjasovellus hakee, validoi ja lähettää tekstimuodossa palvelimelle SSH- tai ehkä HTTP-protokollaa hyödyntämällä, sillä molempien käyttö on yksinkertaista korkean tason komentosarja- tai ohjelmointikielillä.

Meshworks-järjestelmä korvasi KYAB Saberin pilottiprojektin edetessä. Saber-tuki oli merkittävin syy, miksi kehittää tämän työn sovellusta, joten sovellus päätettiin olla ottamatta käyttöön. Meshworks Wireless -järjestelmä tarjoaa KYAB Saberin tarjoamien suureiden lisäksi monta muuta suuretta ja on lisäksi skaalautuvampi.

Lähteet

1 Save Energy -hanke. (WWW-dokumentti.) Metropolia Ammattikorkeakoulu. <http://www.metropolia.fi/tutkimus-ja-kehitystoiminta/hankkeet/save-energy/>. Luettu 7.10.2009.

2 Kippo, Asko. Helsinki-pilotin projektipäällikkö, Metropolia Ammattikorkeakoulu, Helsinki. Keskustelut syksyllä 2009.

3 Save Energy Website, Helsinki. (WWW-dokumentti.) <http://www.ict4saveenergy.eu/helsinki>. Luettu 7.10.2009.

4 Pirinen, Jukka-Pekka. Lehtori, Metropolia Ammattikorkeakoulu, Helsinki. Keskustelu talvella 2010.

5 DS9490R. (WWW-dokumentti.) Maxim Integrated Products, Inc. <http://datasheets.maxim-ic.com/en/ds/DS9490-DS9490R.pdf>. Luettu 23.11.2009.

6 Taina, Juha. Ohjelmistotuotannon kurssimateriaali. Helsingin Yliopisto. (WWW-dokumentti.) <http://www.cs.helsinki.fi/u/taina/ohtu/k-2005/>. Luettu 12.11.2009.

7 Asus WL-500gP V2. ASUSTeK Computer Inc. (WWW-dokumentti.) <http://www.asus.com/>. Luettu 8.10.2009.

8 ASUS WL-500g Premium – OpenWrt Wiki. (WWW-dokumentti.) <http://wiki.openwrt.org/toh/asus/wl500gp#info>. Luettu 8.10.2009.

9 Ippolito, Greg. Linux Tutorial – Static, Shared Dynamic and Loadable Linux Libraries. (WWW-dokumentti.) <http://www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html>. Luettu 27.10.2009.

10 Linker (computing). (WWW-dokumentti.) Wikipedia. <http://en.wikipedia.org/wiki/Linker_%28computing%29>. Luettu 20.4.2010.

11 Learning UML. (WWW-dokumentti.) <http://etutorials.org/Programming/Learning+uml/>. Luettu 12.11.2009.

12 Ippolito, Greg. Fork, Exec and Process control. (WWW-dokumentti.) <http://www.yolinux.com/TUTORIALS/ForkExecProcesses.html>. Luettu 21.10.2009.

13 The IEEE and The Open Group. fork. (WWW-dokumentti.) <http://www.opengroup.org/onlinepubs/9699919799/functions/fork.html>. Luettu 21.10.2009.

14 Park, Alfred. Multithreaded Programming (pthreads Tutorial). (WWW-dokumentti.) <http://randu.org/tutorials/threads/>. Luettu 21.10.2009.

15 Erlin, Rami. Nomovok Oy. Keskustelu 18.11.2009.

16 About the OpenWrt Image Builder. (WWW-dokumentti.) <http://wiki.openwrt.org/oldwiki/imagebuilderhowto>. Luettu 18.11.2009.

17 Installing OpenWrt via TFTP. (WWW-dokumentti.) <http://wiki.openwrt.org/doc/howto/tftp>. Luettu 11.11.2009.

18 About the OpenWrt SDK. (WWW-dokumentti.) <http://wiki.openwrt.org/oldwiki/buildingpackageshowto>. Luettu 18.11.2009.

19 Keskustelut IRC-kanavalla #linuxfin IRCnet-verkossa marraskuussa 2009.

20 Valgrind. (WWW-dokumentti.) <http://valgrind.org/docs/manual/quick-start.html>. Luettu 28.10.2009.

21 DS2438. (WWW-dokumentti) Maxim Integrated Products, Inc. <http://datasheets.maxim-ic.com/en/ds/DS2438.pdf>. Luettu 23.11.2009.

22 Erlin, Rami. Nomovok Oy. Utilizing Dunkkis for Metropolia. (Luottamuksellinen dokumentti.) 24.11.2009. Luettu 24.11.2009.

Liite 1: Säikeistetyn luokan run-metodin toteutus

```
void Controller::run(void) {
	/* bool Thread::serving
	 * Ehto, jolla silmukka on ajossa.
	 * Arvoa on mahdollista muuttaa olion ulkopuolelta.
	 */
	serving = true;

	while (serving) {
		/* bool Thread::pause
		 * Kun metodia Thread::suspend kutsuttu,
		 * arvo vaihtuu todeksi jolloin tämä toimii
		 * ilmoituksena käskystä keskeyttää silmukan suoritus.
		 * suspend() asettaa myös muuttujan bool Thread::sleeping
		 * arvon todeksi.
		 */
		if (pause) {
			// voi huoletta nollata saman tien
			pause = false;

			/* int Thread::toggleLock()
			 * Palauttaa nollan onnistuessaan, muussa tapauksessa
			 * virhettä edustavan kokonaisluvun.
			 *
			 * Argumentilla true lukitsee Thread-luokan
			 * sisäisen muuttujan (mutex-objekti) ja falsella avaa sen.
			 */
			toggleLock(true);
			while (sleeping) {
				/* int Thread::wait()
				 * Odottaa, kunnes kontrollerin ulkopuolelta tulee käsky
				 * jatkaa silmukan suorittamista.
				 * Onnistuessaan palauttaa nollan,
				 * muussa tapauksessa virhettä edustavan kokonaisluvun.
				 */
				wait();
			}
			toggleLock(false);
		} // if (pause)

		// vector<Device*> Controller::line_dev
		// Kontrollerin jono jossa uutta dataa omaavat oliot saapumisjärjestyksessä
		// Jos jonossa on yksikin olio..
		if (line_dev.size() > 0) {
			// ..haetaan siltä data.
			retrieve(line_dev[0]);
		}

		// bool Controller::remote_connected
		// kertoo, onko tietokantayhteys päällä
		if (remote_connected) {
			// jos on, puretaan joko väliaikaissäilöä
			// tai kontrollerin omaa jonoa tietokantaoliolle.
			purge();
		}

		// standardifunktio joka toteuttaa säännöllisen odotuksen
		nanosleep(&ts, NULL);
	} // while(serving)

	serving = false;
}
```

Liite 2: Sovelluksen luokkien tärkeimpien attribuuttien ja metodien selityksiä taulukkomuodossa

Taulukko 1. Luokan **Thread** *tärkeimpiä attribuutteja ja metodeja.*

f	funktio-osoitinmuuttuja, jonka arvoksi asetetaan olion silmukan sisältävän metodin osoite
serving	olion silmukan elossa pitävä ehtomuuttuja
sleeping	arvolla tosi olion silmukassa tarkistetaan jatkuvasti nukkumislukkomuuttujan arvoa
mutex	tulee sanoista mutual exclusion; lukkomuuttuja, joka aukeaa kun ehtomuuttuja 'cond' raukeaa
cond	olion silmukan nukkumislukkoehtomuuttuja
t_wait	olion silmukan säännöllisen odotusajan (sekunteja, nanosekunteja) sisältävä muuttuja
start()	kutsuu säikeistetyn luokan run()-metodia
toggle_serve(bool t)	t:n arvolla tosi asettaa olion silmukan ehtomuuttujan arvon todeksi, arvolla epätosi asettaa sen epätodeksi, jolloin silmukasta (ja run()-metodista) poistutaan. Ehtomuuttujan arvo on oletusarvoisesti tosi.
toggle_lock(bool t)	t:n arvolla tosi lukitsee olion silmukan nukkumassa pitävän mutex-muuttujan, arvolla epätosi avaa sen
suspend()	asettaa olion silmukan nukkumaanmenokäskymuuttujan 'pause' arvon todeksi sekä nukkumissilmukkaehtomuuttujan 'sleeping' arvon todeksi
wakeup()	asettaa olion silmukan nukkumissilmukkaehtomuuttujan 'sleeping' arvon epätodeksi ja kutsuu metodia signal()
wait()	kutsuu olion silmukan nukkumislukkoehtomuuttujan arvon tarkistavaa funktiota pthread_cond_wait(). Palaa, kun ehto ei ole enää voimassa.
signal()	kutsuu olion silmukan nukkumislukkoehtomuuttujan purkavaa funktiota pthread_cond_signal()
set_sleeptime()	asettaa olion silmukalle uuden säännöllisen odotusajan
join()	kutsuu säikeen lopettavaa funktiota pthread_join()

Taulukko 2. Laiterajapinnan **Device** *sekä luokan* **Dunkkis** *tärkeimpiä attribuutteja ja metodeja.*

data	muuttuja, johon aluksi luetaan viimeisin mittausarvo Dunkkis-piirilevyltä
queue	laiteolion sisäinen FIFO-jono, jonka loppuun lisätään muuttujan 'data' sisältö riveittäin
sets	taulukko jossa säilytetään olion asetusarvoja. Joissain tapauksissa kätevämpi kuin get/set-metodi jokaista jäsenmuuttujaa kohden
init()	laitteen alustusmetodi, jolla ei Dunkkiksen kohdalla ole käyttöä
close()	laitteen lopetusmetodi, jolla ei Dunkkiksen kohdalla ole käyttöä
set()	Dunkkiksen asetuksen asetusmetodi. Ottaa vastaan asetuksen nimen sekä asetuksen arvon
getData()	antaa kutsujalle laiteolion datajonon ensimmäisen tietueen muistiosoitteen, tai mikäli jono on tyhjä, paluuarvo on NULL
readData()	laiteolion sisäiseen käyttöön tarkoitettu metodi, jolla lukee

	tekstitiedostosta viimeisimmän Dunkkis-piirilevyltä saadun mittausdatan
validate()	jokaisen mitattavan suureen lukeman oikeellisuuden hyväksyvä tai hylkäävä metodi

Taulukko 3. Luokan **Settings** *attribuutteja ja metodeja.*

sets	taulukko jossa säilytetään kaikkia sovelluksen asetustiedostosta luettuja asetuksia
get()	pyydetyn asetuksen arvon palauttava metodi
set()	asettaa uuden asetusavaimen sekä -arvon tai korvaa vanhan arvon
read()	lukee asetustiedoston

Taulukko 4. Luokan **LibraryProvider** *tärkeimpiä attribuutteja ja metodeja.*

addr	osoitinmuuttuja, johon sijiotetaan yhden dynaamisesti ladatun funktion muistiosoite
open()	lataa pyydetyn moduulitiedoston muistiin
close()	poistaa ladatut symbolit muistista
obtain()	etsii pyydetyn funktion osoitteen muistissa ja sijoittaa sen muuttujaan 'addr'
create_object()	palauttaa pyydetyn olion

Taulukko 5. Luokan **Email** *attribuutteja ja metodeja.*

sets	taulukko jossa säilytetään olion asetusarvoja.
send()	lähettää sähköpostiviestin

Taulukko 6. Luokan **Storage** *attribuutteja ja metodeja.*

file	väliaikaistallennustiedoston nimi
insert()	lisää dataa tiedoston loppuun
expunge()	sijoittaa tiedoston ensimmäisen rivin annettuun muuttujaan muistissa. Kun tiedosto on luettu loppuun, se tyhjennetään automaattisesti

Taulukko 7. Luokan **Database** *tärkeimpiä attribuutteja ja metodeja.*

hostname	tietokantapalvelimen osoite
port	TCP-portti, jossa tietokantapalvelinsovellus vastaa pyyntöihin
username	käyttäjätunnus tietokantapalveluun
connected	kertoo, onko tietokantayhteys päällä vai poikki
queue	datajono, jossa kaikki tiedot tietokantaan lähettämistä varten
ping()	tarkistaa, onko yhteys päällä lähettämällä yksinkertaisen pyynnön tietokantapalvelinsovellukselle
is_connected()	palauttaa muuttujan 'connected' arvon
select_db()	lähettää tietokantapalvelinsovellukselle pyynnön vaihtaa tietokantaa
query()	lähettää tiedontallennuslausekkeen tietokantapalvelinsovellukselle
add_row()	kontrolleriolio käyttää tätä lisätäkseen jonoon uuden rivin
make_query()	luo tiedontallennuslausekkeen yhdelle jonon alusta otetulle riville

Taulukko 8. Luokan **Controller** *tärkeimpiä attribuutteja ja metodeja.*

report_level	kynnysluku, joka tapahtuman vakavuuden aste on vähintään

	oltava jotta tapahtuma raportoitaisiin sovelluksen ulkopuolelle
something_to_purge	kertoo, onko Kontrollerin sisäisessä jonossa tai väliaikaistallennustiedostossa jotain purettavaa
storage_available	kertoo, onko väliaikaistallennustiedosto luettavissa
line_dev	dataa tarjoavien laitteiden jono
queue	FIFO-datajono, jossa data riveittäin
init()	käytetään vain sovelluksen binääritiedoston nimen antamiseen raportointia varten. Annetaan käyttöjärjestelmän tapahtumarekisteripalvelulle jotta tapahtumalokia lukiessa tietää mistä sovelluksesta viesti on tullut
data_available()	laiteolioiden käyttämä metodi, jolla Kontrollerille annetaan tieto millä laiteoliolla on uutta dataa saatavilla
disconnected()	tietokantaolio kertoo tällä Kontrollerille, että tietokantayhteys on katkennut. Metodissa vaihdetaan purkumetodiksi purge_storage().
connected_again()	tietokantaolio kertoo tällä Kontrollerille, että uusi tietokantayhteys on saatu. Metodissa raportoidaan asiasta sekä mikäli Kontrollerin jonossa on purettavaa, asetetaan purkumetodiksi purge_queue().
purge_queue()	välittää Kontrollerin datajonon ensimmäisen rivin muistiosoitteen tietokantaoliolle.
purge_storage()	välittää väliaikaistallennustiedoston ensimmäisen rivin tietokantaoliolle.
report()	raportoi Kontrollerin sekä pääohjelman tapahtumat
report_to_log()	raportoi laiteolioiden, tietokantaolion, moduulinlataajan sekä väliaikaistallennusolion ilmoittamat tapahtumat