TAMPEREEN TEKNILLINEN YLIOPISTO

ARTO SEPPÄ

HISTORIATIEDON TALLENNUS JA TOISTO TILANNEKUVAJÄRJESTELMÄSSÄ

Diplomityö

Tarkastajat: Prof. Tarja Systä (TTY),
DI Markku Tienhaara (iDS)
Tarkastajat ja aihe hyväksytty
tieto- ja sähkötekniikan tiedekuntaneuvoston
kokouksessa 9.12.2009

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
ARTO SEPPÄ: Historiatiedon tallennus ja toisto tilannekuvajärjestelmässä
Diplomityö, 49 sivua, 3 liitesivua
Syyskuu 2012
Pääaine: Ohjelmistotekniikka
Tarkastajat: Prof. Tarja Systä (TTY), DI Markku Tienhaara (iDS)
Avainsanat: DDS, historiatieto, sarjallistaminen, geneerisyys

Ihmisen pyrkiessä ymmärtämään ympäristöään on usein luonnollista pohtia nykyhetken lisäksi myös sitä, miten nykyiseen tilanteeseen on päädytty. Tämä voi helpottaa kokonaiskuvan muodostamista ja sitä kautta parantaa tilannetietoisuutta. Muun muassa tämän vuoksi nykyiset ilmapuolustuksen käytössä olevat johtamisjärjestelmät tukevat myös menneiden tilanteiden läpikäyntiä.

Tämä työ on tehty Insta DefSec Oy:lle osana hajautetun järjestelmän tiedonhallinnan tutkimusta. Tätä tutkimusta tehdään osana kokonaisuutta, jossa kehitetään uuden sukupolven johtamisjärjestelmiä. Työssä toteutettiin historiapalvelu, joka mahdollistaa tilannekuvatiedon tallentamisen historiatiedoksi sekä tämän historiatiedon tarkastelun myöhemmin. Historiatieto tallennetaan tiedostoihin levylle sarjallistamalla se XML-muotoon. Lisäksi toteutettiin apukirjasto järjestelmän viestintäarkkitehtuurina toimivan DDS-väylän helpompaan käyttöön sekä muokattiin olemassaolevaa karttakäyttöliittymää soveltuvaksi historiatiedon toiston vastaanottoon ja ohjaukseen.

Työssä testattiin toteutetun historiapalvelun suorituskykyä ja pyrittiin tätä kautta selvittämään onko tällainen järjestelmä suorituskyvyn puolesta järkevää toteuttaa. Lisäksi etsittiin suorituskyvystä mahdollisia pullonkauloja, joita poistamalla suorituskykyä voitaisiin parantaa. Selkeimmäksi parannuskohteeksi suorituskyvyn parantamiseen havaittiin tiedon sarjallistamisen kehittäminen. Kokonaisuutena historiapalvelun suorituskyky todettiin kuitenkin riittäväksi sille tarkoitettuun käyttöön.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
ARTO SEPPÄ: Saving and replaying of historical data in a common operational picture system
Master of Science Thesis, 49 pages, 3 Appendix pages
September 2012
Major: Software Engineering
Examiners: Prof. Tarja Systä (TUT), M.Sc. Markku Tienhaara (iDS)
Keywords: DDS, historical data, serialization, generic programming

As humans strive to better understand their environment it is often natural to reflect on the events leading to the current situation in addition to examining the current environment. This may help in forming a picture of the situation and thus improve situational awareness. This is one of the reasons current air defence command and control systems support reviewing historical information.

This thesis was made for Insta DefSec Oy as a part of research on information management in a distributed system. This research is done as a part of developing a new generation of command and control systems. The focus of this thesis is to implement a history service to enable saving operational picture data and aid in reviewing the saved data later on. This historical data is saved in files on a hard disk by serializing it in XML format. In addition to the history service a software library was implemented to aid in using the DDS data bus used in the communication architecture of the system, and an existing user interface was modified to support and control the reception of historical data.

The performance of the implemented history service was also evaluated in the thesis in an effort to find out if the system is feasible performance-wise. The results were also further evaluated in an attempt to find possible bottlenecks in the performance. In this evaluation it was found out that the best way to improve the performance would be to improve the serialization of the data. As a whole the performance of the history service was found out to be acceptable for its planned use.

ALKUSANAT

Tämä diplomityö, *Historiatiedon tallennus ja toisto tilannekuvajärjestelmässä*, on tehty Insta DefSec Oy:lle toimiessani siellä ohjelmistosuunnitteluharjoittelijana. Työ tehtiin osana hajautetun järjestelmän tiedonhallinnan tutkimusta.

Työn tarkastajina ja ohjaajina toimivat professori Tarja Systä TTY:ltä sekä DI Markku Tienhaara Insta DefSec Oy:stä. Haluan kiittää heitä kärsivällisyydestä yli kaksivuotiseksi venyneen opinnäytetyöprosessin aikana. Lisäksi haluan kiittää tiimini jäseniä, joilta sain aina arvokkaita neuvoja ongelmien yllättäessä.

Suurin kiitos kuuluu kuitenkin avovaimolleni Anna-Liisalle tuesta, kärsivällisyydestä sekä motivoinnista työn aikana.

Tampereella, 14.8.2012

Arto Seppä
Matti Tapion katu 2 B 16
33720 TAMPERE

arto.seppa@iki.fi

SISÄLLYS

TERMIT JA SYMBOLIT

AOP

Aspect-oriented programming. Ohjelmointiparadigma joka pyrkii eriyttämään useisiin ohjelman osiin liittyvät toiminnot, kuten lokituksen ja autentikoinnin irralleen ohjelman varsinaisesta toimintalogiikasta.

APP-6A

NATO-standardi karttasymboliikalle. Mahdollistaa monenlaisten joukkojen esittämisen kartalla yhtenäisellä tavalla.

Bean

Spring-ohjelmistokehyksen yhteydessä tarkoittaa ajettavaa erillistä ohjelmistokomponenttia.

CORBA

Common Object Request Broker Architecture. OMG:n määrittelemä standardi joka mahdollistaa useilla eri ohjelmointikielillä kirjoitettujen ja eri järjestelmissä toimivien ohjelmistokomponenttien yhteistoiminnan.

DCPS

Data-Centric Publish-Subscribe. DDS-standardin tärkein kerros. Tarjoaa sovellukselle toiminnallisuuden tieto-olioiden arvojen julkaisuun ja tilaamiseen.

DDS

Data Distribution Service for Real-time Systems. OMG:n standardi tietokeskeiseen julkaisija-tilaaja-mallin mukaiseen tiedonvaihtoon hajautetuissa järjestelmissä.

DLRL

Data Local Reconstruction Layer. DDS-standardin valinnainen kerros. Helpottaa DDS:n integrointia sovellukseen tarjoamalla saumattomamman yhteyden DDS:n käsitteiden ja ohjelmointikielen käsitteiden välillä.

IDL

Interface description language. Tässä työssä nimenomaan OMG:n CORBA:n yhteydessä määrittelemä kuvauskieli ohjelmistokomponentin rajapinnan määrittämiseksi.

JOKE

Johtokeskus. Puolustusvoimien kiinteä tai liikuteltava johtokeskus, josta käsin voidaan johtaa esimerkiksi ilmapuolustusta.

Luokkapolku

Kertoo Javan virtuaalikoneelle mistä sen tulee etsiä käännettyjä luokkia ja muita tiedostoja.

OMG

Object Management Group. Konsortio, joka julkaisee standardeja liittyen muun muassa hajautettuihin järjestelmiin sekä ohjelmien ja järjestelmien mallinnukseen. Tunnetuimpia standardeja ovat UML ja CORBA.

PIM

Platform Independent Model. Alustariippumaton malli, esittää käsitteitä alustasta riippumattomalla tavalla. Käsitteet voidaan yleensä muuntaa jollakin muunnoksella alustariippuvaisen mallin käsitteiksi. PIM liittyy mallipohjaiseen ohjelmistokehitykseen (Model Driven Architecture, MDA).

PSM

Platform Spesific Model. Alustariippuvainen malli, esittää käsitteitä alustariippuvaisella tavalla. PSM saadaan tyypillisesti PIM:istä sopivalla muunnoksella. PSM liittyy mallipohjaiseen ohjelmistokehitykseen (Model Driven Architecture, MDA).

Publisher/Subscriber

Viestinvälitykseen pohjautuva viestintäarkkitehtuuri.

QoS

Quality of Service. Palvelunlaatu. Vaikuttaa tiedon käsittelyyn ja priorisointiin.

Seuranta

Ilmatilannekuvassa oleva seurattava kohde.

XML

Kuvauskieli, jolla kuvataan dokumentin sisällön lisäksi myös sen rakenne. Käytetään usein myös tietorakenteiden kuvaamiseen.

1. JOHDANTO

Puolustusvoimien Teknillisen Tutkimuslaitoksen Sotateknisen arvion ja ennusteen [1, s. 38] mukaan "Tekniikka on ja tulee kasvavassa määrin olemaan ihmisen laajennus, ihmisen apukeino laajentaa ihmisen mahdollisuuksia ja vaikutusta ympäristöönsä yleensä ja sodankäyntiin erityisesti."

Tämä toistaa tuttua ajatusta, jonka mukaan uusien teknisten järjestelmien kehittäminen tulee olemaan jatkuvasti kasvavassa roolissa kaikilla yhteiskunnan osa-alueilla. Tämä työ on tehty osana hajautetun järjestelmän tiedonhallinnan tutkimusta, joka on osa uuden sukupolven johtamisjärjestelmien kehitykseen tähtäävää kokonaisuutta. Työssä toteutettu prototyyppi tulee todennäköisesti toimimaan alustana kehitettäessä historiatiedon tallennus- ja toistopalveluita uuteen järjestelmään.

1.1 Työn tausta ja motivaatio

Nykyään ihminen käyttää päätöksenteossa apunaan monenlaisia järjestelmiä. Esimerkiksi yritysmaailma on täynnä liiketoimintatiedon hallintajärjestelmiä (Business Intelligence), joita käytetään apuna yrityksen johtoon liittyviä päätöksiä tehtäessä. Johtamisjärjestelmät ja niiden tarjoama reaaliaikainen tilannekuva toimivat samalla tavalla päätöksenteon apuna esimerkiksi ilmapuolustuksen johtamisessa. Tilannekuvassa voidaan esittää sekä omien että vihollisjoukkojen sijainti ja muuta olennaista tietoa lähes reaaliajassa.

Aina reaaliaikainen tieto ei riitä päätöksenteon tueksi. Silloin apua voidaan hakea menneiden tilanteiden analysoinnista. Joskus menneisyyttä halutaan tarkastella myös kun halutaan varmistaa esimerkiksi rajaloukkauksen tapahtuminen menneisyydessä. Tämä voidaan ratkaista lisäämällä järjestelmään mahdollisuus tallentaa tapahtumia historiatiedoksi, ja tarkastella niitä myöhemmin.

1.2 Työn tavoitteet

Tämän työn kontekstina toimii Insta DefSec Oy:ssä kehitetty reaaliaikaista tilannekuvaa esittävä ohjelmisto. Työn tarkoituksena on tutkia historiatiedon tallentamista ja toistamista hajautettua viestinvälityskerrosta (*DDS, Data Distribution System*) käyttävässä tilannekuvasovelluksessa. Tämä tehdään tuottamalla prototyyppi historiapalvelusta, joka toteuttaa nämä ominaisuudet.

Ohjelmiston käyttöönotto ei saa aiheuttaa huomattavia vaikutuksia muuhun järjestelmään. Osana työtä on historiapalvelun suorituskyvyn tarkastelu, jolla pyritään arvioimaan järjestelmän suorituskykyisyyttä todellisessa ympäristössä.

1.3 Työn rakenne

Johdannon jälkeisessä luvussa kaksi pyritään kasvattamaan lukijan ymmärtämystä työn taustalla olevasta sovellusalueesta. Luvussa kuvataan sovellusalueeseen kuuluvia asioita, aloittaen tilannekuvan tarkemmasta kuvailusta ja siirtyen hajautettujen järjestelmien kautta hajautettuihin tilannekuvajärjestelmiin. Lopuksi luodaan tarkempi katsaus historiatietoon ja sen tavanomaisimpiin käyttötarkoituksiin, erityisesti tilannekuvajärjestelmän kontekstissa.

Luvussa kolme tutustutaan tarkemmin työhön liittyviin tekniikoihin. Tavoitteena on tutustuttaa lukija työn kannalta olennaisiin tekniikoihin, jotka saattavat olla entuudestaan tuntemattomia. Luvussa tutustutaan hajautettuihin viestinvälityskerroksiin ja erityisesti DDS:ään, geneeriseen ohjelmointiin Java-ohjelmointikielellä, sarjallistamiseen ja XStream-kirjastoon, sekä joihinkin pienemmässä roolissa oleviin kirjastoihin ja ohjelmistokehyksiin.

Luvussa neljä keskitytään itse prototyyppiin. Luku aloitetaan erittelemällä tarkemmin prototyypin kehitykseen vaikuttaneet rajoitukset. Tästä jatketaan käsittelemällä yksityiskohtaisesti prototyypin arkkitehtuuria pala kerrallaan. Lopuksi käsitellään vielä tarkemmin prototyypin olennaisimpien osien toimintaa.

Luvussa viisi käsitellään prototyypin suorituskykytestausta, joka on työn toinen merkittävä osuus. Testimenettelyt ja tulokset käydään läpi, ja lopuksi saatuja tuloksia arvioidaan.

Viimeisessä luvussa tehdään yhteenveto työssä toteutetusta prototyypistä ja sille suoritettujen suorituskykytestien tuloksista. Lisäksi käydään läpi prototyypin vaikutukset ympäröivän järjestelmän toteutukseen, työn suorituksen hylätyt toteutusvaihtoehdot sekä tulevaa kehitystä varten syntyneet jatkokehitysajatukset.

2. TILANNEKUVA JA HISTORIATIETO

Tämä työ toteutetaan liittyen hajautettuun tilannekuvajärjestelmään. Työn taustan ymmärtämiseksi on tärkeätä ymmärtää tilannekuvan ja hajautettujen järjestelmien käsitteiden perusteet. Tärkeätä on myös ymmärtää historiatiedon luonne ja tuntea mahdollisia käyttötarkoituksia.

2.1 Tilannekuva

Ihmiselle on luonnollista pyrkiä käsittämään ja hallitsemaan ympäristöään. Tässä prosessissa olennaista on ympäristön tutkiminen ja ymmärtäminen, ja tällä tavoin ympäristöä kuvaavan mentaalisen mallin muodostaminen. Näin ihmiselle muodostuu mieleensä käsitys ympäröivästä tilanteesta eli tilannekuva. Arkipäiväisissä tilanteissa omat aistihavainnot yleensä riittävät tilannekuvan muodostamiseen, mutta monimutkaisempien tilanteiden hallitsemiseen tarvitaan apuvälineitä. Apuvälineiksi tilannekuvan muodostamisessa voidaan ajatella vaikkapa kartalle tehtyjä merkintöjä vihollisjoukkojen oletetusta suunnasta tai suuren moottorin kierroslukumittaria.

Joskus tilanteen hahmottamiseksi tarvitaan paljon tietoa, jota havainnoija ei voi suoraan omien aistiensa kautta saada. Tällöin apuna voidaan käyttää kokonaista tilannekuvajärjestelmää. Tilannekuvajärjestelmä on järjestelmä, joka esittää useita tilanteeseen liittyviä muuttujia yhdistetysti, esimerkiksi tietokoneen ruudulla. Sen tarkoitus on esittää mahdollisimman selkeästi tilanteeseen liittyvä olennainen tieto, ja näin auttaa käyttäjäänsä saavuttamaan ymmärrys tilanteesta tämän tiedon perusteella.

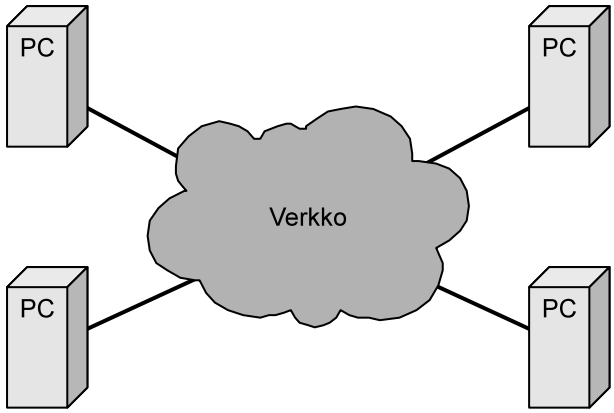
Esimerkkejä tilannekuvan käytöstä siviilitarkoituksissa voi löytää vaikkapa lentoliikenteen ohjauksesta ja suurten tuotantolaitosten valvomoista. Lähestymislennonjohdossa tilannekuvajärjestelmänä toimii tutkan ruutu, johon tutkahavaintojen perusteella muodostuvan tilannekuvan avulla lennonjohto porrastaa lentokoneet kohti turvallista laskua. Tuotantolaitosten valvomoissa tietokoneiden ruudulla näkyy jatkuvasti prosessin eri vaiheista kerättyä tilannetietoa jonka avulla valvotaan, että prosessi toimii sujuvasti sekä voidaan havaita ja ratkaista mahdolliset ongelmatilanteet ajoissa.

Tähän työhön liittyvä järjestelmä on tilannekuvajärjestelmä, joka esittää käyttäjälleen tutkien ja muiden sensorien perusteella saatua tietoa kohteista. Tässä työssä keskitytään erityisesti ilmatilassa oleviin kohteisiin. Näitä kohteita kutsutaan

seurannoiksi. Järjestelmässä voidaan esittää esimerkiksi tietoa seurantojen tyypistä, paikasta, nopeudesta ja suunnasta.

2.2 Hajautetut järjestelmät

Coulourisin, Dollimoren ja Kindbergin mukaan hajautettu järjestelmä määritellään sellaiseksi järjestelmäksi, jossa tietoverkossa olevien tietokoneiden laitteisto- ja ohjelmistokomponentit kommunikoivat ja koordinoivat toimintojaan ainoastaan välittämällä viestejä. Heidän mukaansa tästä määritelmästä aiheutuu hajautetulle järjestelmälle seuraavia tyypillisiä ominaisuuksia: rinnakkaisuus, yhteisen kellon puute ja mahdollisuus osittaisiin virhetilanteisiin joissa vain osa järjestelmästä on virhetilanteessa muiden siitä tietämättä. [2]



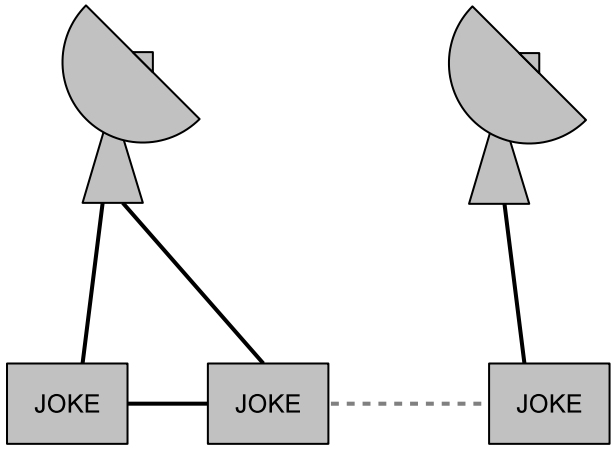
Kuva 2.1: Hajautettu järjestelmä

Hajautetussa järjestelmässä on tavallisesti kyse resurssien jakamisesta. Jaettava resurssi saattaa olla esimerkiksi tietoa tai laitteistoresurssi. Hyvä esimerkki tiedon jaosta hajautetusti on WWW (World Wide Web), jossa miljoonat tietokoneet hakevat tietoa toisilta, palvelimina toimivilta tietokoneilta. Laitteiston jakamisena voidaan ajatella vaikkapa tulostimen jakamista lähiverkon koneille tai suoritinajan jakamista jossakin massivisessa hajautetun laskennan projektissa kuten SETI@home[3] tai Folding@home[4]. Näissä projekteissa jopa sadattuhannet vapaaehtoiset antavat tietokoneidensa suoritinaikaa projektin käyttöön. Tätä valtavaa laskentatehoa käytetään erilaisiin massiivista laskentaa vaativiin kohteisiin projektista riippuen. SETI@home analysoi radioteleskoopeilta vastaanotettua dataa pyrkien löytämään niistä merkkejä maan ulkopuolisesta teknologiasta, kun taas Folding@home pyrkii simuloimalla ymmärtämään paremmin proteiinien laskostumista ja siihen liittyviä moninaisia lääketieteen ongelmia. Folding@home on laskentateholtaan yksi tehokkaimpia hajautettuja laskentajärjestelmiä yli 5,6 petaFLOPSin (floating point operations per second) suorituskyvyllään[5].

2.3 Hajautettu tilannekuvajärjestelmä

Jaettu tilannetietoisuus on yhä tärkeämmäksi koettu asia. Se tarkoittaa sitä, että kaikki tilanteessa olevat tahot ovat yhtälailla tietoisia tilanteesta ja siihen vaikuttavista tekijöistä. Jaetun tilannetietoisuuden saavuttamiseksi on tärkeätä muodostaa jaettu tilannekuva. Sen muodostamiseksi kaikille on saatava siirrettyä sama tilannekuva tietoineen. Tähän liittyy kiinteästi tilannekuvajärjestelmän hajauttaminen.

Hajautetulla tilannekuvajärjestelmällä tarkoitetaan tässä työssä tilannekuvajärjestelmää, jossa sekä sensorit, että tilannekuvan tarkkailijat ovat hajautettuja sekä loogisesti, että maantieteellisesti järjestelmiin, jotka kommunikoivat toistensa kanssa verkon välityksellä. Puolustuskäytössä oleva verkko ei kuitenkaan ole täysin luotettava, sillä yhteydet saattavat vastapuolen toimista johtuen olla ajoittain poikki. Lisäksi joitakin järjestelmän osia saatetaan siirtää, ja siirron ajaksi yhteydet katkeavat. Tämä voi tarkoittaa käytännössä esimerkiksi kuvassa 2.2 esitettyä tilannetta, jossa järjestelmään kuuluu kaksi tutkaa ja kolme johtokeskusta (kuvassa JOKE) eri puolilla Suomea. Kokonaisuus on jakautunut kahteen saarekkeeseen niin, että toinen tutkista ja kaksi johtokeskusta ovat yhdistetty toisiinsa tietoverkolla, ja toinen tutkista on yhdistetty kolmanteen johtokeskukseen. Tietoa näiden saarekkeiden välillä voidaan välittää ainoastaan ajoittain kolmannen johtokeskuksen päästessä yhteyteen muiden kanssa.



Kuva 2.2: Hajautettu tilannekuvajärjestelmä

Tilannekuvajärjestelmän hajauttamisella saavutetaan monia hyötyjä. Sen lisäksi että hajauttaminen auttaa jaetun tilannetietoisuuden saavuttamisessa tekemällä tilannekuvan tarkkailijoiden lisäämisestä luonnollista, se parantaa järjestelmän taistelukestävyyttä merkittävästi. Menetetyt yhteydet voidaan korvata kierrättämällä tieto jotain toista reittiä pitkin, ja jokainen johtokeskus voi tarvittaessa toimia myös ilman yhteyttä toisiin. Tällaisen järjestelmän tekeminen kokonaan toimintakyvyttömäksi on vaikeata. Vaikka järjestelmän komponenttien välisten yhteyksien häiritse-

minen heikentääkin järjestelmän toimintaa, se säilyttää kuitenkin merkittävän osan toimintakyvystään.

Puolustuskäytössä olevan hajautetun järjestelmän luonteeseen kuuluva yhteyksien suuri epävarmuus aiheuttaa haasteita erityisesti tiedon siirrossa järjestelmän osien välillä. Tästä johtuen tilannekuvatietoa voidaan myös kerätä erikseen järjestelmän eri osissa tallenteiksi ja siirtää tallenteita järjestelmien välillä yhteyksien toimiessa, tai yhdistää tallenteita kokonaisuudeksi myöhemmin.

2.4 Historiatieto tilannekuvajärjestelmässä

Kokonaisvaltaisen tilannekuvan saavuttamiseksi ei aina riitä pelkästään nykyisen tilanteen tutkiminen, vaan täytyy ottaa huomioon myös se, miten tilanteeseen on päädytty. Menneistä tapahtumista voidaan ottaa oppia tai niihin palaamalla voidaan purkaa nopeasti edennyt tilanne askel kerrallaan ilman kiireen tuomaa painetta. Tästä syystä tilannetiedon tallentaminen historiatiedoksi ja sen tarkasteleminen myöhemmin on tilannekuvajärjestelmältä erittäin toivottu ominaisuus.

Ilmatilannekuvan historiatiedon tallentaminen on erityisen tärkeää ilmavalvonnan havaintojen myöhemmän tarkastelun kannalta. Rauhan aikana tapahtuneet mahdolliset alueloukkaukset voidaan tutkia tarkkaan tallenteiden perusteella ja usein niiden perusteella määritetään onko loukkausta tapahtunut[6]. Toinen tyypillinen ilmatilannekuvan historiatiedon käyttötarkoitus on ilmavoimien koulutuslentojen läpikäynti ja analysointi opetustarkoituksissa[7].

3. KÄYTETYT TEKNIIKAT

Sovellusalueen luoman kehyksen lisäksi sovelluksen ympäristöä määrittävät siihen liittyvät tekniset ratkaisut. Tämän prototyypin toteutuksessa merkittävää osaa näyttelevät tietoväylänä käytetty Data Distribution Service, geneerisen ohjelmoinnin menetelmät Javassa ja sarjallistamisen tekniikat.

3.1 Hajautetut välikerrokset

Hajautetuissa järjestelmissä kriittinen osa järjestelmän suunnittelua on hajautettujen komponenttien välisen viestintäarkkitehtuurin valinta ja suunnittelu. Ohjelmistokomponentteja, jotka yhdistävät hajautetun järjestelmän sovellukset toisiinsa korkeammalla tasolla kutsutaan *välikerroksiksi* (middleware). Välikerros piilottaa tietoverkon käyttöön liittyvät yksityiskohdat sovellukselta ja tarjoaa sovelluksille helpon rajapinnan kommunikointiin.

3.1.1 Proseduurikeskeiset ja viestikeskeiset välikerrokset

Välikerrokset jakautuvat pääasiassa kahteen luokkaan: *proseduurikeskeisiin* ja *viestikeskeisiin* välikerroksiin. Proseduurikeskeisissä välikerroksissa viestintä perustuu useimmiten etäproseduurikutsuihin (*Remote Procedure Call, RPC*) tai etäolioiden käyttöön etämetodien kautta. Näissä välikerroksissa pyritään useimmiten mallintamaan hajautetun järjestelmän käyttöä mahdollisimman normaalina proseduraalisen ohjelman suorittamisena, jossa toisessa järjestelmässä olevan komponentin proseduureja tai metodeita kutsutaan samoin kun paikallisessa järjestelmässä olevia. Tämä kommunikaatio on luonteeltaan useimmiten synkronista. Viestikeskeisissä välikerroksessa kommunikaatio tapahtuu vaihtamalla viestejä järjestelmien välillä. Tässä tapauksessa järjestelmien erillisyyttä ei pyritä peittämään, vaan se tuodaan esiin olennaisena osana kommunikaatiota. Kommunikoinnille tarjotaan kuitenkin helppo rajapinta, joka on luonteeltaan asynkroninen. [8]

3.1.2 Julkaisija-tilaaja-välikerros

Tässä työssä käytetty DDS (Data Distribution Service)[9] edustaa yhtä viestinvälitysarkkitehtuurin muotoa, julkaisija-tilaaja-arkkitehtuuria (publish-subscribe architechture). Tunnetussa kirjassaan suunnittelumalleista Gamma et al. määrittelevät

julkaisija-tilaaja-arkkitehtuurin tarkkailija-mallin toisena kutsumanimenä[10], mutta tässä työssä tarkennamme sen määritelmää tarkoittamaan tarkkailija-mallin yleistystä eri järjestelmiin hajautetuille osapuolille.

Julkaisija-tilaaja-arkkitehtuuri koostuu kahdenlaisista osapuolista, *julkaisijoista (publisher)* ja *tilaajista (subscriber)*. Arkkitehtuurissa julkaisijat julkaisevat viestejä, joita voi vastaanottaa mielivaltainen määrä tilaajia. Viesti julkaistaan aina tiettyyn aiheeseen ja tilaajat voivat rekisteröityä kuuntelemaan tiettyyn aiheeseen julkaistavia viestejä. Aihe siis yhdistää tilaajat julkaisijoihin. Julkaisijan ei tarvitse tuntea tilaajia edeltä tai edes tietää onko niitä olemassa. Vastaavasti tilaajan ei tarvitse tuntea julkaisijoita vaan sille riittää tieto aiheesta, johon sen haluamia viestejä julkaistaan.

Hyvinä puolina tällaisessa arkkitehtuurissa on osapuolten löyhä kytkeytyneisyys toisiinsa ja selkeän rajapinnan muodostuminen niiden välille. Tämä helpottaa huomattavasti järjestelmän myöhempää laajentamista, sillä uusia komponentteja voidaan toteuttaa järjestelmään muuttamatta vanhoja.

Huonoina puolina voidaan pitää korkeampaa monimutkaisuutta kuin yksinkertaisessa toteutuksessa, ja tämän monimutkaisuuden mukanaan tuomia mahdollisia suorituskykyhaasteita. Myös matala kytkeytyneisyys voi tuoda ongelmia, jos tarjottu rajapinta ei tarjoa riittäviä ominaisuuksia toteutukseen.

3.1.3 DDS (Data Distribution Service)

DDS on OMG:n (Object Management Group) standardoima rajapintamäärittely, joka DDS-Intron[11] mukaan on "rajapintamäärittely – joka määrittelee datakeskeisen julkaisija-tilaaja-arkkitehtuurin anonyymien tiedon tuottajien ja kuluttajien yhdistämiseen." Saman lähteen mukaan tyypillistä sovellusarkkitehtuuria jossa käytetään DDS:ää voi kuvata tietoväyläarkkitehtuurina.

DDS-standardi[9] jakautuu kahteen eri tasoon, jotka ovat *DCPS (Data-Centric Publish-Subscribe)* ja *DLRL (Data Local Reconstrcuction Layer)*. DCPS kuvaa alemman tason rajapinnan, joka soveltuu hyvin datakeskeisen hajautetun järjestelmän rajapinnaksi. DLRL kuvaa DCPS:n päälle rakennetun rajapinnan, jonka tarkoitus on helpottaa sovelluksen kehittämistä tarjoamalla rajapinnat kohdekielen rakenteiden, kuten olioiden, välittämiseen sellaisenaan DCPS-rajapinnan yli. Tässä työssä käytetään vain DCPS:ää, joten DLRL:ää ei käsitellä tämän tarkemmin.

Standardi on edelleen jaettu alustariippumattomaan osaan (*Platform Independent Model, PIM*) ja alustariippuvaisiin osiin (*Platform Spesific Model, PSM*). Alustariippuvaiset osat kertovat miten alustariippumattoman osan käsitteet mallinnetaan kullekin alustalle. Alustariippuvaisia kuvauksia standardi sisältää ainoastaan yhden, OMG:n *IDL-kielelle (Interface Description Language)*. IDL on alustariippumaton rajapinnan kuvauskieli, joka on määritetty osana *CORBA*:n (*Common Ob-*

ject Request Broker Architecture) standardia[12, luku 6.1.5]. Nykytilanteessa käytännössä kaikki implementaatiot sisältävät esikääntäjän, joka tuottaa IDL-kuvauksista kohdekielen luokkia. Standardin seuraavaan versioon odotetaan alustariippuvaisia kuvauksia C++ ja Java-ohjelmointikielille.

Käsitteet

Seuraavaksi esitellään tärkeimmät DDS-ympäristössä käytetyt käsitteet. Esittelyt on kirjoitettu lähteitä[9, 13] mukaillen.

Toimialue (domain) kuvaa yhtä aluetta, johon kuuluvat sovellukset voivat kommunikoida keskenään. Jokainen julkaisija tai tilaaja voi kuulua vain yhteen toimialueeseen, eikä tätä toimialuetta voi sovelluksen ajon aikana muuttaa.

Aihe (topic) yhdistää osapuolet toisiinsa, sillä kaikki julkaistavat viestit julkaistaan aina johonkin aiheeseen. Aihe koostuu aiheen yksikäsitteisestä nimestä ja aiheeseen julkaistavan tiedon tyypistä. Kirjoittajat, lukijat, kohteet ja näytteet liittyvät aina johonkin aiheeseen.

Julkaisija (publisher) vastaa tiedon välityksestä toisille osanottajille. Julkaisija voi välittää kaiken tyyppistä tietoa, mutta tietoa voi välittää ainoastaan kirjoittajien kautta.

Tilaaja (subscriber) vastaa tiedon vastaanottamisesta ja sen saattamisesta sovelluksen saataville. Tilaaja voi vastaanottaa kaiken tyyppistä tietoa, mutta vastaanotettu tieto on käytettävissä ainoastaan lukijoiden kautta.

Kirjoittaja (writer) on olio, jonka kautta voidaan käyttää julkaisijaa tyypitetysti. Yhtä kirjoittajaa voidaan käyttää ainoastaan yhteen aiheeseen kirjoittamiseen, mutta sovelluksella voi olla useampia kirjoittajia eri aiheita ja tietotyyppejä varten.

Lukija (reader) on olio, jonka kautta voidaan käyttää tilaajaa tyypitetysti. Lukijan kautta voidaan lukea näytteitä ainoastaan yhdestä aiheesta, mutta sovelluksella voi olla useampia lukijoita eri aiheita ja tietotyyppejä varten.

Kuuntelija (listener) voidaan rekisteröidä lukijalle kuuntelemaan saapuvia näytteitä. Se ei ole toiminnan kannalta välttämätön, sillä lukijaa voi käyttää myös ilman kuuntelijaa. Kuuntelijaa käyttämällä sovellus saa heti tiedon saapuvista näytteistä tarkkailija-mallin mukaisesti. Yhdelle lukijalle voidaan haluttaessa rekisteröidä useita kuuntelijoita.

Kohde (instance) on julkaistavan tiedon yksi alkio. Kohde voi saada eri arvoja eri ajanhetkillä. Kohteella on tyypin lisäksi aina tietty aihe, johon se on julkaistu. Lisäksi sillä on yksikäsitteinen avain, joka yksilöi tietoalkion muiden samaan aiheeseen julkaistujen joukosta. Jos avainta ei määritellä, voi aiheeseen kuulua ainoastaan yksi kohde.

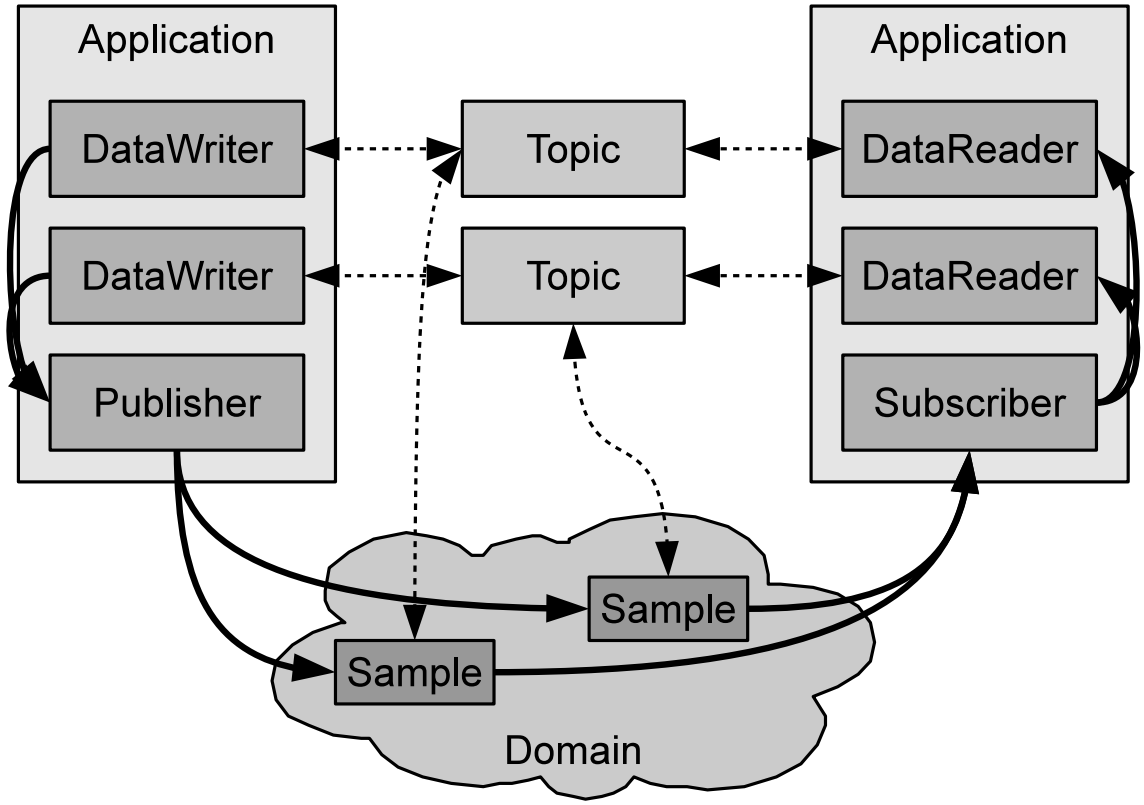
Näyte (sample) on yhteen kohteeseen kohdistuva päivitys. Päivitettävä kohde määritellään avainkentällä. Kohteen arvoksi tulkitaan aina uusin siitä saatu näy-

te. Palvelunlaatuominaisuuksista riippuen luettavissa voi olla uusimman näytteen lisäksi myös vanhempia näytteitä.

Palvelunlaatu (QoS, Quality of Service) näyttelee suurta osaa DDS:n toiminnassa. Julkaisijaan, tilaajaan, kirjoittajaan, lukijaan ja aiheeseen voidaan liittää monia palvelunlaatumääritteitä, jotka määrittävät väylän toimintaa. Niillä voidaan määritellä tiedon pysyvyyteen, välittämiseen, oikea-aikaisuuteen, resurssienkäyttöön ja konfigurointiin liittyviä asetuksia. Palvelunlaadun suunnitteleminen oikein on erittäin tärkeätä sovelluksen sujuvalle toiminnalle.

Osio (partition) on toimialuetta kevyempi tapa rajata DDS-väylän toimijoita toimimaan erillään toisistaan. Samaan osioon kuuluvat osanottajat voivat viestiä keskenään, mutta osiosta toiseen viestiminen ei onnistu. Toimijat voivat vaihtaa osiotaan sovelluksen ollessa käynnissä ja myös toimia useassa osiossa samanaikaisesti. Jokainen julkaisija, tilaaja, kirjoittaja ja lukija kuuluu aina vähintään yhteen osioon.

Tunnus (instance handle) on jokaisella DDS-komponentilla oleva järjestelmänlaajuisesti yksikäsitteinen tunnus. Tässä työssä tunnusta käytetään erottamaan useat samanaikaiset historiantoistoistunnot toisistaan.



Kuva 3.1: Asiakasohjelman kannalta olennaismmat DDS-standardin määrittelemät komponentit, mukailtu lähteestä [14, s. 15]

Kuvassa 3.1 esitellään tärkeimmät tiedonkulkuun liittyvät luokat ja niiden osuus tiedonkulussa. Vasemmassa laidassa olevassa sovelluksessa on kaksi kirjoittajaa, jotka molemmat liittyvät eri aiheisiin. Nämä lähettävät näytteen julkaisijan avustuk-

sella kaikille toimialueella oleville tilaajille, jotka ovat ilmaisseet kiinnostuksensa kyseisiin aiheisiin. Näytteet päätyvät oikeassa laidassa olevan sovelluksen tilaajakomponentille, joka edelleen välittää ne tilauksen tehneille lukijoille. Tämän jälkeen näytteet ovat sovelluksen luettavissa näitä lukijoita käyttäen.

Erikoispiirteet

Kirjoittajan ja lukijan rooleja kuvaavat DataWriter ja DataReader ovat abstrakteja tyyppejä ja niistä täytyy johtaa käytetylle tietotyypille sopiva aliluokka käyttöä varten. Käytännössä tämä hoituu automaattisesti esikääntäjällä, joka kääntää tietotyypin IDL-kuvauksesta sopivasti tyypitetyt DataWriter ja DataReader-luokat. Samalla luodaan myös monta muuta apuluokkaa, joita tarvitaan sovelluksen toteuttamisessa.

IDL-kuvaus on kielineutraali tapa määrittää rajapinta. Näin voidaan helposti toteuttaa järjestelmiä, joiden komponentit voidaan kirjoittaa eri kieliä käyttäen. Huonona puolena kielineutraaliudessa on se, että kielikohtaiset erityisominaisuudet jäävät hyödyntämättä. Tästä voi joskus aiheutua harmia, kuten aliluvussa 3.3 nähdään sarjallistamisen yhteydessä.

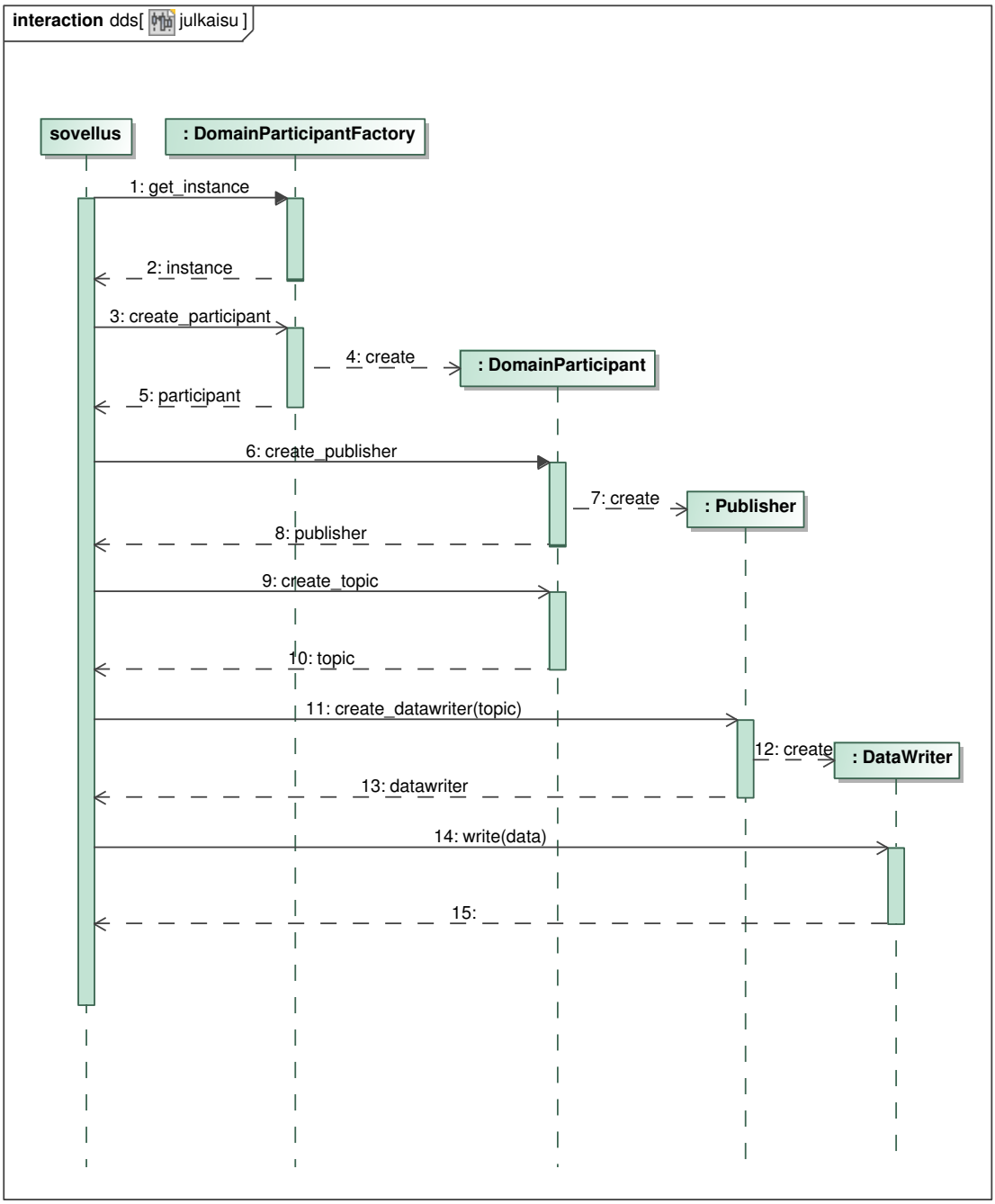
DDS-standardissa on myös joitakin ominaisuuksia jotka tukevat geneerisen ja automaattisesti toimivan ohjelmakoodin kirjoittamista. Tämän työn kannalta näistä erityisen kiinnostava on väylän sisäänrakennetut aiheet. Nämä aiheet kertovat reaaliaikaista tietoa muun muassa väylälle liittyvistä tilaajista ja julkaisijoista, sekä tämän työn kannalta tärkeimpänä kaikista väylällä olevista aiheista. Tätä ominaisuutta voitaisiin käyttää tallennuspalvelun tallennuksen automatisointiin, siten että palvelu asettuu automaattisesti kuuntelemaan kaikkia havaitsemiaan aiheita.

Käytännössä tämän ratkaisun yleiskäyttöisyyttä rajoittaa jossain määrin se, että väylällä käytetty tietotyyppi ja siihen liittyvät IDL-kuvauksesta käännetyt apuluokat täytyy olla tallennuspalvelun saatavissa, jotta tietotyypin kuvaamia tietorakenteita voidaan tallennuspalvelussa vastaanottaa.

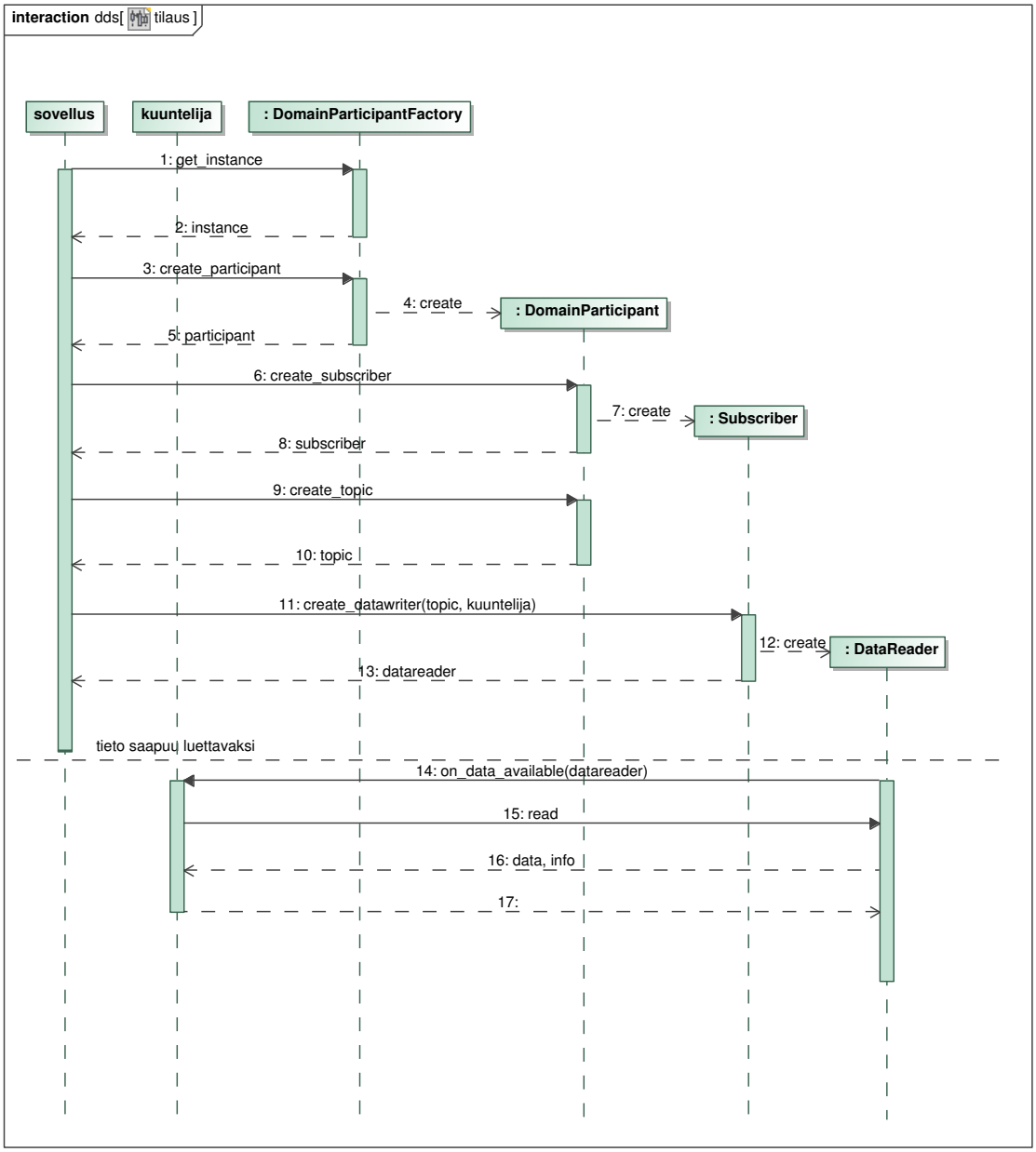
DDS:n käyttö

DDS:ää käytetään tyypillisesti julkaisemalla tietoa tyypitettyjen kirjoittajien kautta ja rekisteröimällä kuuntelijoita tyypitetyille lukijoille. Näihin tyypitettyihin olioihin käsiksi pääseminen vaatii sovellukselta muutamia toimenpiteitä, joita on yksinkertaistetusti kuvattu sekvenssikaavioina kuvissa 3.2 ja 3.3. Näistä kaavioista on jätetty selkeyden vuoksi kokonaan pois muun muassa palvelunlaadun asetusten asettaminen.

Kuvista 3.2 ja 3.3 näemme, että julkaisijan ja tilaajan alustamiseen liittyy hyvin samankaltaisia vaiheita. Ensin haetaan DomainParticipantFactoryn kautta instans-



Kuva 3.2: Yksinkertaistettu sekvenssikaavio DDS-julkaisijan käynnistymisestä ja tiedon julkaisusta, mukailtu lähteestä [15, s. 18]



Kuva 3.3: Yksinkertaistettu sekvenssikaavio DDS-tilaajan käynnistymisestä ja tiedon vastaanotosta, mukailtu lähteestä [15, s. 19]

si DomainParticipantista, joka kuvaa sovelluksen jäsenyyttä tietyssä toimialueessa. Sitä käyttämällä voidaan edelleen luoda julkaisija tai tilaaja. Tiedon vastaanottoon luodaan tyypillisesti lisäksi vielä kuuntelija. Julkaisijaa käyttäen voidaan luoda kirjoittaja tai vastaavasti tilaajaa käyttäen lukija. Vasta näitä käyttäen päästään käsiksi tiedon lähettämiseen ja vastaanottoon.

DDS:n käyttöönotto on melko suoraviivaista ja yksinkertaisen sovelluksen toteuttaminen onnistuu melko helposti esimerkkejä noudattamalla. Kompaktius tai yksinkertaisuus eivät kuitenkaan tule ensimmäisenä mieleen tarkastellessa yksinkertaisen DDS-sovelluksen lähdekoodia. Liitteessä 1 on esimerkki tällaisesta ohjelmasta. Moni yksityiskohta, joista ohjelmoijan tarvitsee vain harvoin olla kiinnostunut, näkyy suoraan osana alustuskoodia. Tämä ongelma kertautuu jos ohjelmassa käsitellään useita aiheita ja tietotyyppejä.

Nykyisellään DDS:n käyttö vaatii harmillisen paljon koodirivejä. Olennainen sisältö, kuten esimerkiksi palvelunlaatuasetusten asettaminen, katoaa helposti epäolennaisten alustustoimenpiteiden sekaan. Suuri osa alustuskoodia myös monistuu helposti sovelluksessa, joka käyttää useampia aiheita ja tietotyyppejä.

3.2 Geneerinen ohjelmointi Javalla

Java ohjelmointikielenä tarjoaa kaksi mekanismia geneerisen ohjelmakoodin tuottamiseen. Mekanismit ovat keskenään erilaisia ja niiden käyttötarkoitukset eroavat hieman toisistaan. Molemmat ovat kuitenkin olennaisessa osassa kirjoitettaessa geneerisiä komponentteja Javalla.

3.2.1 Java Generics

Yksi Javan kömpelöistä ominaisuuksista ennen version 5.0 mukanaan tuomaa Java Genericsiä oli geneerinen ohjelmointi, ja erityisesti säiliöiden (Java Collections Framework) käyttö. Säiliöiden yleiskäyttöisyyden takia niihin voidaan tallentaa minkä tyyppistä tietoa tahansa ja haluttaessa myös useamman tyyppistä tietoa samaan säiliöön. Tämä tietotyyppien sekoittaminen ei ole tyypillisesti tarpeen, mutta sen mahdollisuus on säiliön läpikäynnin kannalta ongelmallista.

Säiliöt käydään tavallisesti läpi käyttäen iteraattoreja. Koska säiliöihin voidaan tallentaa minkä tahansa tyyppistä tietoa, iteraattorin palauttama tieto voi olla ainoastaan tyyppiä java.lang.Object, joka on Javassa luokkahierarkian ylin luokka ja täten kaikkien luokkien kantaluokka. Tämä on säiliöiden yleiskäyttöisyyden kannalta hyvä ratkaisu, mutta paluuarvon käyttämiseksi sen tyyppi tarvitsee useimmiten vielä muuntaa oikeaksi. Tämä tyyppimuunnos on virhealtis ja vähentää koodin selkeyttä.

Listauksessa 3.1 on esitetty kokonaislukusäiliön luominen ja käyttö Javassa ennen versiota 5.0. Mainittu tyyppimuunnos näkyy listauksen rivillä kahdeksan. Siinä tapauksessa että aikaisemmin koodissa (rivillä kaksi) olisikin lisätty listaan jotain muuta kun Integer-tyyppiä oleva olio, tämä tyyppimuunnos epäonnistuu. Tämä on ongelma siksi, että virhe tapahtuu vasta ajonaikaisesti, eikä kääntäjä voi sitä havaita.

```
1  List myIntList = new LinkedList();
2  myIntList.add(new Integer(2));
3  myIntList.add(new Integer(3));
4  myIntList.add(new Integer(5));
5
6  Iterator myIntIt = myIntList.iterator();
7  while(myIntIt.hasNext()) {
8      Integer x = (Integer) myIntList.iterator().next();
9      System.out.println(x);
10 }
```

Listaus 3.1: Kokonaislukusäiliö ilman Genericsiä

Javan Generics-mekanismi on Javan versiossa 5.0 kieleen mukaan tullut mekanismi joka helpottaa geneeristä ohjelmointia ja tekee siitä käännösaikaisten tarkastusten myötä vähemmän virhealtista. Samankaltaisia mekanismeja on useissa muissakin kielissä, joista erityisesti voi mainita C++:n mallit (template). Useimmat Javaan tutustuvat ohjelmoijat oppivat Genericsin käytön ensimmäisenä Javan säiliöitä (Java Collections Framework) käyttäessään. Java Generics takaa geneeriselle ohjelmoinnille käännösaikaisen turvallisuuden, ja näin vähentää lopputuotteeseen pääsevien ohjelmointivirheiden vaaraa. [16]

```
1  List<Integer> myIntList = new LinkedList<Integer>();
2  myIntList.add(new Integer(2));
3  myIntList.add(new Integer(3));
4  myIntList.add(new Integer(5));
5
6  Iterator<Integer> myIntIt = myIntList.iterator();
7  while(myIntIt.hasNext()) {
8      Integer x = myIntList.iterator().next();
9      System.out.println(x);
10 }
```

Listaus 3.2: Kokonaislukusäiliö Genericsiä käyttäen

Listauksessa 3.2 esitetään kokonaislukujen tallentamiseen käytettävän säiliön luominen ja käyttö Javassa Genericsiä käyttäen. Listauksia vertaamalla huomaamme niiden erona ensimmäisellä rivillä olevan tyyppimäärityksen List<Integer>. Tällä määritetään listan sisältävän ainoastaan Integer-tyyppisiä olioita ja näin myös kääntäjä voi tarkastaa että listaan ei yritetä muun tyyppisiä olioita lisätä. Myös rivin 8

Iterator<Interger> on samalla tavalla tyypitetty, joten sen käyttö tarkastetaan käännösaikaisesti. Kolmas ero on ensimmäisen listauksen(3.1) kolmannella rivillä oleva tyyppimuunnos, jota ei tässä versiossa tarvita.

Genericsin lisääminen Javaan on parantanut geneerisen ohjelmakoodin laatua. Tuloksena on ohjelmakoodia, josta on entistä selvemmin nähtävissä ohjelmoijan aikeet. Säiliöiden ja iteraattoreiden tyypit näkyvät aina niiden esittelyn yhteydessä, ja tyyppimuunnokset säästyvät kohtiin joissa tyyppimuunnos on merkittävämmässä roolissa.

3.2.2 Reflektio

Javan reflektio-mekanismi on Genericsiä vanhempi tapa geneerisen ohjelmoinnin mahdollistamiseen. Se on huomattavasti Genericsiä joustavampi ja käyttöalueeltaan laajempi. Se mahdollistaa muun muassa vielä käännösaikana tuntemattoman tyyppisten olioiden luomisen, näiden ominaisuuksien tutkimisen ja käyttämisen, sekä taulukkojen ja lueteltujen tyyppien (enum) dynaamisen luomisen ja käyttämisen. [17]

Reflektion tyypillisiä käyttökohteita ovat ohjelmistojen laajentaminen jälkikäteen liitännäisillä sekä luokkien ja olioiden ominaisuuksien tutkiminen (introspection) kehitystyökaluissa niille annetuista näkyvyysmääritteistä huolimatta. Reflektion haittapuolina mainitaan suorituskyvyn heikentyminen, abstraktioiden rikkoutuminen ja mahdolliset ajoympäristön rajoitukset jotka voivat estää reflektion käytön esimerkiksi Java-sovelmissa (applet). [17]

Listauksessa 3.3 esitetään lyhyt esimerkki reflektion yksinkertaistetusta käyttötapauksesta jossa luodaan ensin luokasta instanssi ja sitten kutsutaan sen jäsenfunktiota. Huomionarvoista on, että riveillä 1–3 olevat merkkijonot (luotavan olion tyyppi, kutsuttava funktio, ja parametrit) voitaisiin lukea esimerkiksi ulkopuolisesta asetustiedostosta ohjelmakoodiin kirjoittamisen sijaan. Vertailun vuoksi listauksessa 3.4 esitetään vastaava esimerkki ilman reflektion käyttöä. Kuten nähdään, ohjelmakoodi on huomattavasti lyhyempää ja selkeämpää, mutta reflektion tarjoamat dynaamiset mahdollisuudet puuttuvat siitä täysin.

Reflektion suuri dynaamisuus ja monikäyttöisyys asettavat haasteita sitä käyttävälle ohjelmoijalle. Reflektiiviset operaatiot voivat epäonnistuessaan aiheuttaa useita erilaisia ajonaikaisia poikkeuksia. Kaikkien näiden ajonaikaisten poikkeusten käsittely on hoidettava huolellisesti jotta ohjelmisto toimisi vakaasti. Ohjelmoijan täytyy siis olla huomattavasti tarkempi välttääkseen vaikeasti selvitettävien ohjelmointivirheiden jäämisen koodiin. Lisäksi reflektiota käyttävien ohjelmiston osien testauksen täytyy olla huolellista, sillä kääntäjä ei voi ajonaikaisisesti ilmeneviä virheitä huomata.

```
1  String nimi = "com.foo.Hilavitkutin";
2  String funktio = "teeJotain";
3  String parametri = "Hello World!";
4
5  try {
6      Class c = Class.forName(nimi);
7      Object hv = c.newInstance();
8      hv.getMethod(funktio, String.class).invoke(hv, parametri);
9  } catch (Exception e) {
10     // Oikeassa ohjelmassa kaikki eri tyyppiset
11     // poikkeukset otettaisiin erikseen kiinni
12 }
```

Listaus 3.3: Olion luominen ja funktion kutsuminen reflektiota käyttäen

```
1  import com.foo.Hilavitkutin;
2
3  Hilavitkutin hv = new Hilavitkutin();
4  hv.teeJotain("Hello World!");
```

Listaus 3.4: Olion luominen ja funktion kutsuminen ilman reflektiota

3.2.3 Miksi reflektio?

Reflektiiviset olioiden luonnit ja funktiokutsut ovat täysin dynaamisen luonteensa vuoksi hitaampia kuin vastaavat ei-reflektiiviset versionsa, mutta reflektion käytön suhteellinen hinta pienenee huomattavasti sen mukaan, mitä monimutkaisempia funktioita sen kautta kutsutaan. Tämä johtuu siitä että reflektiivisen metodikutsun tekemiseen kuluva aika on huomattavasti pienempi kun kutsutun, mahdollisesti hyvinkin monimutkaisen, funktion suorittamiseen kuluva aika. Jos reflektiolla kutsutaan pientä, suorituskykykriittistä funktiota, voi reflektion vaikutus suorituskykyyn kasvaa merkittäväksi, sillä reflektiivinen metodikutsu on tavallista metodikutsua merkittävästi hitaampi. Todellisissa ohjelmistoissa vaikutus suorituskykyyn on kuitenkin useimmiten huomattavasti vähäisempi tai käytännössä jopa olematon.

Vaikka reflektiolla voi olla heikentävä vaikutus suorituskykyyn on sen käyttö tässä työssä välttämätöntä. Historiapalvelussa tarvitaan suurta geneerisyyttä, sillä sen on oltava konfiguroitavissa kuuntelemaan eri tyyppistä tietoa kääntämättä palvelun ohjelmakoodia uudestaan. DDS-kuuntelijan alustaminen vaatii tietotyyppikohtaisten luokkien instantioimista. Tästä johtuen pelkkä Generics ei mekanismina riitä, sillä kaikkia tyyppikohtaisia luokkia ei välttämättä palvelun kirjoitushetkellä ole edes vielä kirjoitettu.

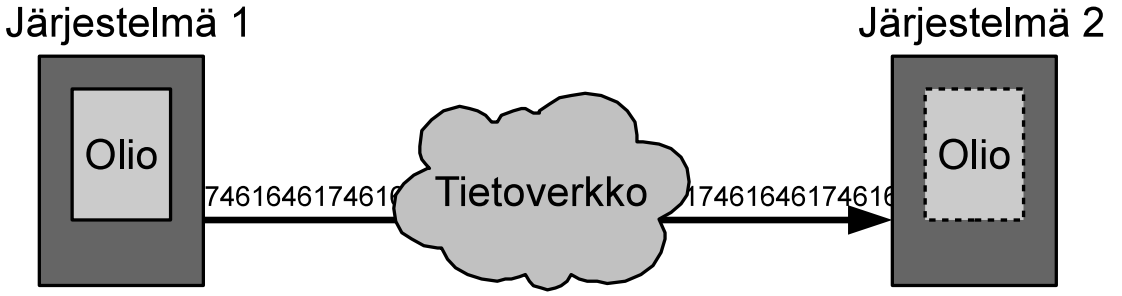
3.3 Sarjallistaminen

Tietorakenteiden sarjallistaminen on tärkeä asia hajautetuissa järjestelmissä. Tässä aliluvussa käsitellään sarjallistamista ensin yleisellä tasolla, tarkentaen käsittelyä sen jälkeen Javaan ja XStream-kirjastoon.

3.3.1 Yleistä

Ohjelmistoja kehitettäessä tietorakenteet ovat alusta asti olleet avainasemassa. Kaikkea ohjelmiston käsittelemää tietoa kuvaa jokin ohjelmointikielen tietorakenne. Nykyisissä oliokeskeisissä ohjelmointikielissä puhutaan usein olioista (object), siinä missä ei-oliokeskeiset kielet käyttävät vain termiä tietorakenne (struct). Tavallisesti ohjelmaa ajettaessa näitä tietorakenteita on järjestelmän muistissa useita, suuremmilla ohjelmilla useita tuhansia tai enemmänkin. On kuitenkin useita käyttötarkoituksia joihin tietorakenteiden pelkkä muistinvarainen tallentaminen ei riitä.

Sarjallistaminen tarkoittaa jonkin ohjelmointikielen tietorakenteen, esimerkiksi olion, muuttamista tavujonoksi. Tätä tavujonomuotoa voidaan käyttää esimerkiksi tallennettaessa rakenne levylle myöhempää käyttöä varten tai siirrettäessä se verkon yli toiseen järjestelmään. Kuva 3.4 hahmottelee sarjallistamisen käyttöä tiedon siirtämisessä kahden järjestelmän välillä. Sarjallistettu rakenne voidaan ladata takaisin alkuperäistä tietorakennetta vastaavaksi myöhempänä ajankohtana tai vaikka täysin eri järjestelmässä. Useat ohjelmointikielet tarjoavat mekanismin sekä sarjallistamiseen, että sarjallistetun tiedon lukemiseen takaisin järjestelmään.



Kuva 3.4: Esimerkki sarjallistamisen käytöstä olion siirtämisessä tietoverkon yli

Mahdollisia sarjallistamismuotoja on useita erilaisia ja ne vaihtelevat ohjelmointikielen mukaan. Useat muodot painottavat esityksen tiiviyttä, jotta sarjallistettu muoto pysyisi mahdollisimman pienenä. Kuitenkin monesti halutaan käyttää myös XML-muotoon sarjallistamista, jotta sarjallistettu muoto olisi helpommin ihmisen luettavissa ja varmemmin siirrettävissä erilaisten järjestelmien välillä. Sarjallistamisen tyypillisiä käyttökohteita ovat tiedon tallentaminen levylle säilytystä varten, hajautetun järjestelmän etäolioiden siirtäminen järjestelmien välillä, sekä tiedon- ja viestinvälitys tietoverkon avulla yhdistettyjen järjestelmien välillä.

3.3.2 Sarjallistaminen Javassa

Javassa, kuten monissa muissakin ohjelmointikielissä, on kattavat sarjallistamisominaisuudet. Oliot eivät kuitenkaan ole oletuksena sarjallistettavissa, sillä kaikkien olioiden sarjallistaminen ei olisi järkevää. Esimerkiksi säiettä kuvaava java.lang.Thread on tilaltaan sidottu sitä ajavan virtuaalikoneen (Java Virtual Machine) tilaan. Olio tulkitaan sarjallistettavaksi jos se toteuttaa rajapinnan java.io.Serializable joko suoraan tai jonkun kantaluokkansa kautta.

Varsinainen sarjallistettavien olioiden sarjallistaminen tapahtuu luokan java.io.ObjectOutputStream metodilla .writeObject(), ja sarjallistetun olion lataaminen luokan java.io.ObjectInputStream metodilla .readObject(). Ladattaessa sarjallistettua oliota täytyy olion määrittävä tavukooditiedosto olla saatavilla luokkapolussa, muuten lataaminen epäonnistuu.

3.3.3 XStream

Historiapalvelussa kerätty tieto täytyy luonnollisesti tallentaa johonkin myöhempää käyttöä varten. Javassa on sisäänrakennettuna kattavat sarjallistamisominaisuudet, jotka on tarkoitettu tietorakenteiden muuttamiseen muotoon, joka on helppoa tallentaa levylle tai siirtää verkon yli. Näiden käyttäminen vaatii että sarjallistettava olio toteuttaa rajapinnan java.io.Serializable. Valitettavasti DDS:n IDL-kääntäjä (katso aliluku 3.1.3) ei kuitenkaan aseta dataolioita toteuttamaan tätä rajapintaa luodessaan niitä IDL-kuvauksista. Vaihtoehdoiksi jäi siis sarjallistamisen toteuttaminen itse tai ulkoisen sarjallistamiskirjaston käyttäminen.

Sarjallistamisen toteuttaminen itse yleispätevällä tavalla on suurehko urakka, joten tässä työssä päädyttiin käyttämään ulkoista sarjallistamiskirjastoa, XStreamia. Kirjasto on käytön yksinkertaisuudeltaan verrattavissa Javan omaan sarjallistamismekanismiin, mutta siitä poiketen XStream ei vaadi sarjallistettavilta olioilta mitään erityisiä ominaisuuksia tai tietyn rajapinnan toteuttamista. Olioiden tallentamiseen ja lukemiseen käytetty rajapinta on periytetty Javan omasta java.io.ObjectOutputStream-luokasta, joten sen käyttäminen ei eroa XStreamin alustuksen jälkeen mitenkään Javan omasta sarjallistamismekanismista.

XStream käyttää oletuksena sarjallistamismuotona XML:ää, tarjoaa mahdollisuuden JSON:in (JavaScript Object Notation) käyttöön, ja mahdollistaa myös itse tehdyt sarjallistamismuodot. XML ei varmastikaan ole sarjallistamismuotona tehokkain mahdollinen, mutta se on XStreamissa tueltaan laajin ja viimeistellyin. Luvussa 5 tehtävien suorituskykymittausten perusteella arvioidaan onko syytä tutkia sarjallistamismuodon vaihtamista.

3.4 Spring

Spring Framework on laaja avoimen lähdekoodin ohjelmistokehys, joka tarjoaa paljon hyödyllisiä resursseja Enterprise-tason Java-sovellusten kehittämiseen. Se on erittäin modulaarisena kehyksenä laajuudestaan huolimatta kevyt, sillä kehittäjä voi vapaasti valita mitä osia Springistä haluaa käyttää. Keveys tulee esiin myös siinä, että Springin käyttäminen näkyy sovelluskoodissa melko vähän ja riippuvuuksia sovelluskoodista Springiin voidaan usein välttää. [18, luku 3.2.1]

Spring tarjoaa valtavasti ominaisuuksia noin kahdellakymmenellä moduulillaan, jotka jakautuvat Springin dokumentaation[19] mukaan pääasiassa kuuteen ryhmään: ydinluokat (Core Container), tiedonkäsittely (Data Access/Integration), web-ohjelmointi (Web), aspektiohjelmointi (AOP), instrumentointi (Instrumentation), ja testiluokat (Test). Tämän työn kannalta kuitenkin ainoastaan ydinluokat ovat relevantteja.

Ydinluokkien ja ehkä koko Spring Frameworkin tunnetuin ominaisuus on hallinnan kääntämisenä (Inversion of Control) ja riippuvuuksien syöttämisenä (Dependency Injection) tunnettu kokonaisuus. Tätä kokonaisuutta käyttämällä siirretään sovelluksien luokkien keskinäisten riippuvuuksien määrittäminen luokilta itseltään ohjelmistokehyksen vastuulle. Luokat itsessään määrittelevät riippuvuutensa toisten luokkien sijasta rajapintoihin, joko rakentajan parametreina tai jäsenmuuttujina. Sovelluksen käynnistyessä ohjelmistokehys syöttää näihin riippuvuuksiin konfiguraationsa mukaisesti toisia sovelluksen olioita.

Näin saadaan sovelluksen luokkien keskenäistä riippuvuutta löyhennettyä ja voidaan tarvittaessa vaihtaa riippuvuudet toisiin luokkiin yksinkertaisesti ohjelmistokehystä konfiguroimalla. Näin helpotetaan mahdollista jatkokehitystä, luokkien uudelleenkäyttöä ja testattavuutta.

3.5 Käyttöliittymän komponentit

Koska tässä työssä käyttöliittymällä on vain melko pieni osuus ei näitä komponentteja esitellä tämän laajemmin. Järjestelmän kokonaisvaltaisessa kehityksessä niillä on kuitenkin hyvin merkittävä osuus, jonka johdosta ne ansaitsevat maininnan.

Java Swing ja JIDE Software

Swing on laaja käyttöliittymäkirjasto Javalle ja on käytännössä ensisijainen valinta tuotettaessa käyttöliittymiä Java-ohjelmistoihin. Se kuuluu osana Javan jakelupakettiin ja on erittäin hyvin laajennettava sekä mukautettava kokonaisuus, joka mahdollistaa suurten ja monipuolisten käyttöliittymien toteuttamisen. Järjestelmään liittyvien käyttöliittymien kehityksessä Swing toimii runkona, jonka päälle käyttöliittymät toteutetaan käyttäen Swingin omien komponenttien lisäksi JIDE Softwaren sekä LuciadMapin komponentteja.

JIDE Software puolestaan on yritys, joka tuottaa useita laajennuksia Swingiin komponenttikirjastojen muodossa. JIDE Softwaren sivuston[20] mukaan ne "keskittyvät runsaasti ominaisuuksia sisältäviin Swing-komponentteihin tarkoituksenaan yksinkertaistaa rikkaiden asiakassovellusten (rich-client applications) kehittämistä." Näitä komponentteja yhdessä käyttämällä voidaan nopeasti kehittää erittäin monipuolisia graafisia Java-sovelluksia Swingin päälle.

LuciadMap

LuciadMap on kehittäjäoppaansa[21] mukaan avoin ja oliokeskeinen ohjelmistorajapinta kehittäjille, jotka haluavat käsitellä ja esittää sovelluksissaan maantieteellistä tietoa. Se pyrkii kokonaisvaltaisen ratkaisun sijasta olemaan tehokas, monikäyttöinen ja laajennettava, ja on sen takia erittäin hyvin soveltuva tilannekuvajärjestelmän karttanäytön osaksi.

LuciadMap koostuu rajapinnoista ja luokista, joita käyttäen voidaan helposti toteuttaa mitä tahansa paikkaan liittyvää tietoa käyttävä Java-sovellus. Se soveltuu erityisen hyvin erittäin interaktiivisiin ja suorituskykyä vaativiin sovelluksiin, joista johtamisjärjestelmät ja reaaliaikainen tilannekuva ovat hyviä esimerkkejä. Muita esimerkkejä voidaan löytää esimerkiksi kiinteistöalalta, logistiikasta ja telekommunikaatiosta. [21]

4. PROTOTYYPPI

Osana työtä kehitettiin prototyyppi historian tallennus- ja toistopalveluista. Seuraavissa aliluvuissa käsitellään prototyypin kehitykseen vaikuttaneita vaatimuksia, sekä prototyypin arkkitehtuuria ja toimintaa.

4.1 Vaatimukset

Työssä toteutetaan prototyyppi historiapalvelusta. Historiapalvelun tarkoitus on toimia tilannekuvan historian tallennus- ja toistopalveluna, eli tallentaa tietoväylällä kulkevaa tilannekuvatietoa historiatiedoksi taltioonsa ja toistaa sitä myöhemmin käyttöliittymiin. Tämä saavutetaan toteuttamalla erikseen tallennuspalvelu ja toistopalvelu. Tallennuspalvelu tallentaa tietoväylältä saamaansa tietoa levylle, ja toistopalvelu toistaa tietoa tallenteista takaisin tietoväylälle.

Tietoväylällä kulkeva tilannekuvatieto koostuu tiettyihin DDS-väylän aiheisiin julkaistavista päivityksistä, jotka kuvaavat muun muassa tilannekuvan kohteiden sijaintia, nopeutta ja muita vastaavia tietoja. Historiapalvelu toteutetaan kuitenkin niin, että se pystyy tallentamaan ja toistamaan mitä tahansa väylällä kulkevaa tietoa.

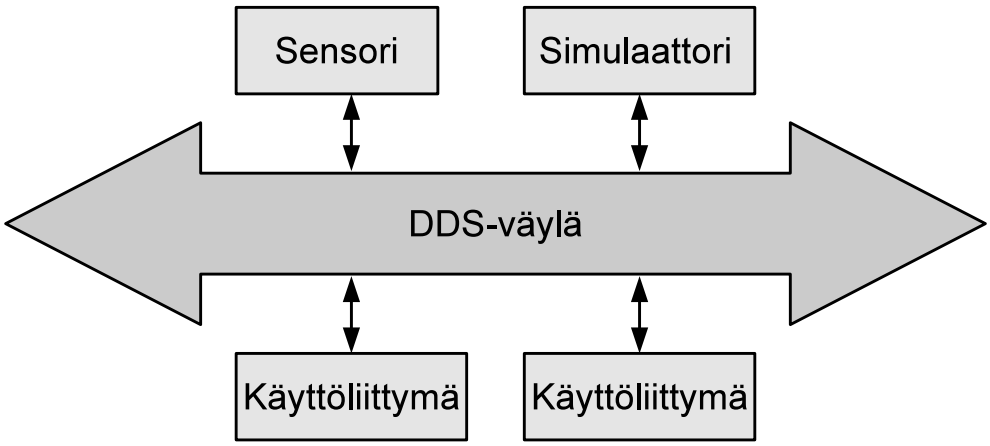
Historiapalvelulta vaaditaan mahdollisimman pientä vaikutusta muuhun järjestelmään. Historiatiedon tallennus ei saa vaikuttaa järjestelmän normaaliin toimintaan ja on voitava ottaa käyttöön muuttamatta olemassa olevia järjestelmän osia. Historiatiedon tallennuksen ja toiston käyttöönotto uusille tietotyypeille on oltava mahdollista ilman historiapalvelun koodin muokkausta. Historiapalvelu ei siis saa olla kooditasolla riippuvainen tallennettavan tiedon tyypistä.

Olemassaolevaa käyttöliittymää on laajennettava tukemaan historian toistoa. Sen on pystyttävä esittämään toistettavaa historiatietoa ja siihen on toteutettava komponentti toiston ohjausta varten. Tällä komponentilla on pystyttävä aloittamaan historian toisto halutusta kohtaa, pysäyttämään se, sekä valitsemaan toistolle haluttu nopeus. Komponentin tulee sopia mahdollisimman hyvin muuhun käyttöliittymään.

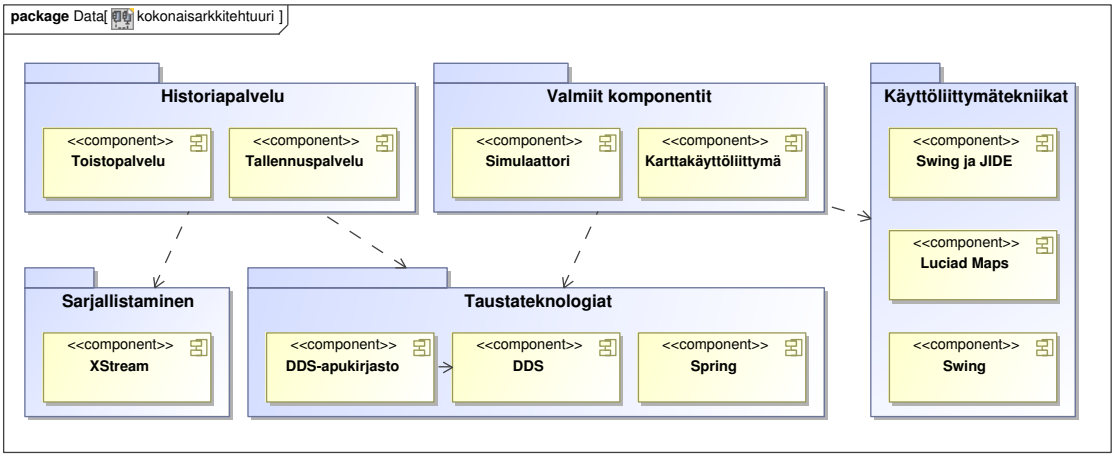
Palvelun toteutuskielenä on Java ja sen versiona Java Standard Edition 6. Tallennuspalvelun on pystyttävä tallentamaan väylällä kulkevaa seuranta- tai muuta tietoa historiatiedoksi hukkaamatta tietoa sen määrän ollessa normaalin rajoissa. Lisäksi toistopalvelun on pystyttävä toistamaan tätä tallennettua historiatietoa vähintään tallennusnopeutta vastaavalla nopeudella.

4.2 Arkkitehtuuri

Järjestelmä jonka osaksi historiapalvelu toteutettiin, on arkkitehtuuriltaan DDS-tietoväylään perustuva hajautettu tilannekuvajärjestelmä. Järjestelmässä on siis tietoväylään liittyviä tiedon tuottajia, ja tiedon kuluttajia. Pääasiallinen käyttö tietoväylälle on tiedon välittäminen tilannekuvassa näkyvistä kohteista, eli seurannoista. Kohdejärjestelmään on aikaisemmin toteutettu seurantatietoa tuottava simulaattori ja seurantatietoa tilannekuvana kartalla esittävä karttakäyttöliittymä. Kuva 4.1 esittää tämän kohdejärjestelmän arkkitehtuuria korkealla tasolla.



Kuva 4.1: Kohdejärjestelmän arkkitehtuuri



Kuva 4.2: Prototyypin kokonaisarkkitehtuuri

Kuva 4.2 esittää järjestelmän arkkitehtuuria komponenttien tasolla. Historiapalvelu koostuu historian tallennus- ja toistopalveluista. Näiden lisäksi toteutettiin myös DDS:n käyttöä helpottava apukirjasto ja kohdejärjestelmään kuuluvaa karttakäyttöliittymää kehitettiin tukemaan historiatiedon toistoa ja ohjausta. Seuraavat aliluvut kertovat tarkemmin prototyyppiin toteutetuista osista.

4.2.1 Seurantatiedon muoto

Järjestelmässä liikkuva seurantatieto koostuu kahdentyyppisestä tiedosta. Molemmilla tietotyypeillä on DDS-väylällä tyypin nimeä vastaava aihe. Ensimmäinen tietotyyppi on esitetty listauksessa 4.1 ja se kuvaa kohteiden perustietoja, jotka eivät muutu yhtä usein kun sijaintitieto. Niistä olennaisin on kohteen tyyppiä kuvaava APP-6A -symboliikan mukainen merkkijono. Tietotyyppiin voitaisiin liittää myös muuta hitaasti muuttuvaa tietoa.

```
1 module airpicture {
2 module dds {
3     struct TrackData {
4         long track_id;
5         string<15> app6a_code;
6     };
7 #pragma keylist TrackData track_id;
8 };
9 };
```

Listaus 4.1: Seurannan perustietojen tietotyyppi

Toinen tietotyyppi on esitetty listauksessa 4.2. Se kuvaa kohteen sijaintia, kurssia ja nopeutta. Tämä tieto on luonteeltaan jatkuvasti päivittyvää. Tähänkin tietotyyppiin voitaisiin liittää muuta sijaintiin liittyvää tietoa, mutta koska valtaosa väylän liikenteestä on todennäköisesti näitä sijaintipäivityksiä, kannattaa turhan tiedon lisäämistä välttää.

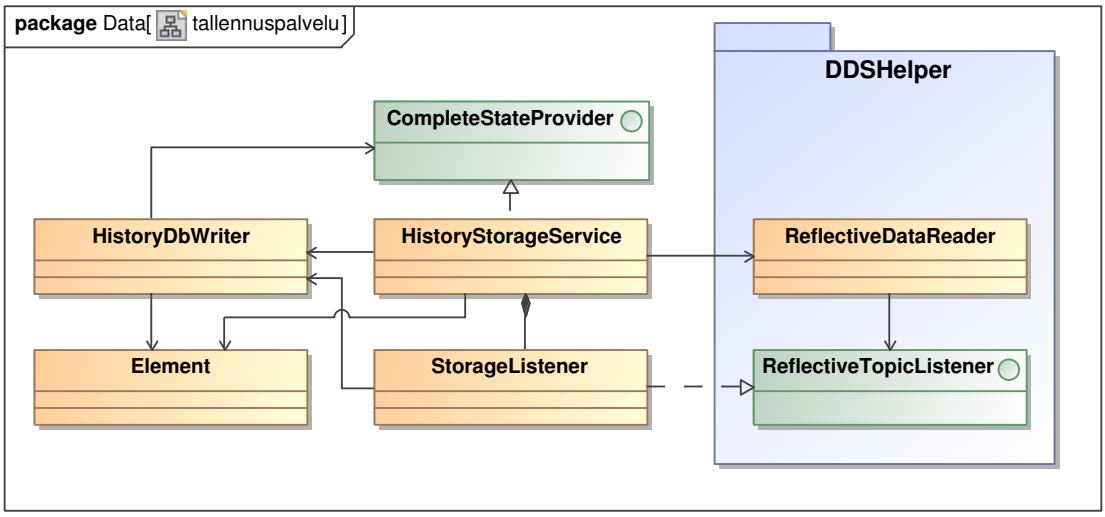
```
1 module airpicture {
2 module dds {
3     struct TrackPosition {
4         long track_id;
5
6         double latitude;
7         double longitude;
8         double speed;
9         double heading;
10    };
11 #pragma keylist TrackPosition track_id;
12 };
13 };
```

Listaus 4.2: Seurannan paikkatietojen tietotyyppi

4.2.2 Historian tallennuspalvelu

Historian tallennuspalvelu HistoryStorageService on toteutettu Springillä käynnistettävissä olevana luokkana, eikä vaadi kuin DDS-palvelun käynnissäolon. Sen ra-

kennetta on esitetty luokkakaaviona kuvassa 4.3. Tallennuspalvelun toteutus on jaettu kahteen osaan, HistoryStorageServiceen ja HistoryDbWriteriin, jotta tietokannan muoto ja käsittely saadaan eriytettyä tallennuspalvelun toteutuksesta.



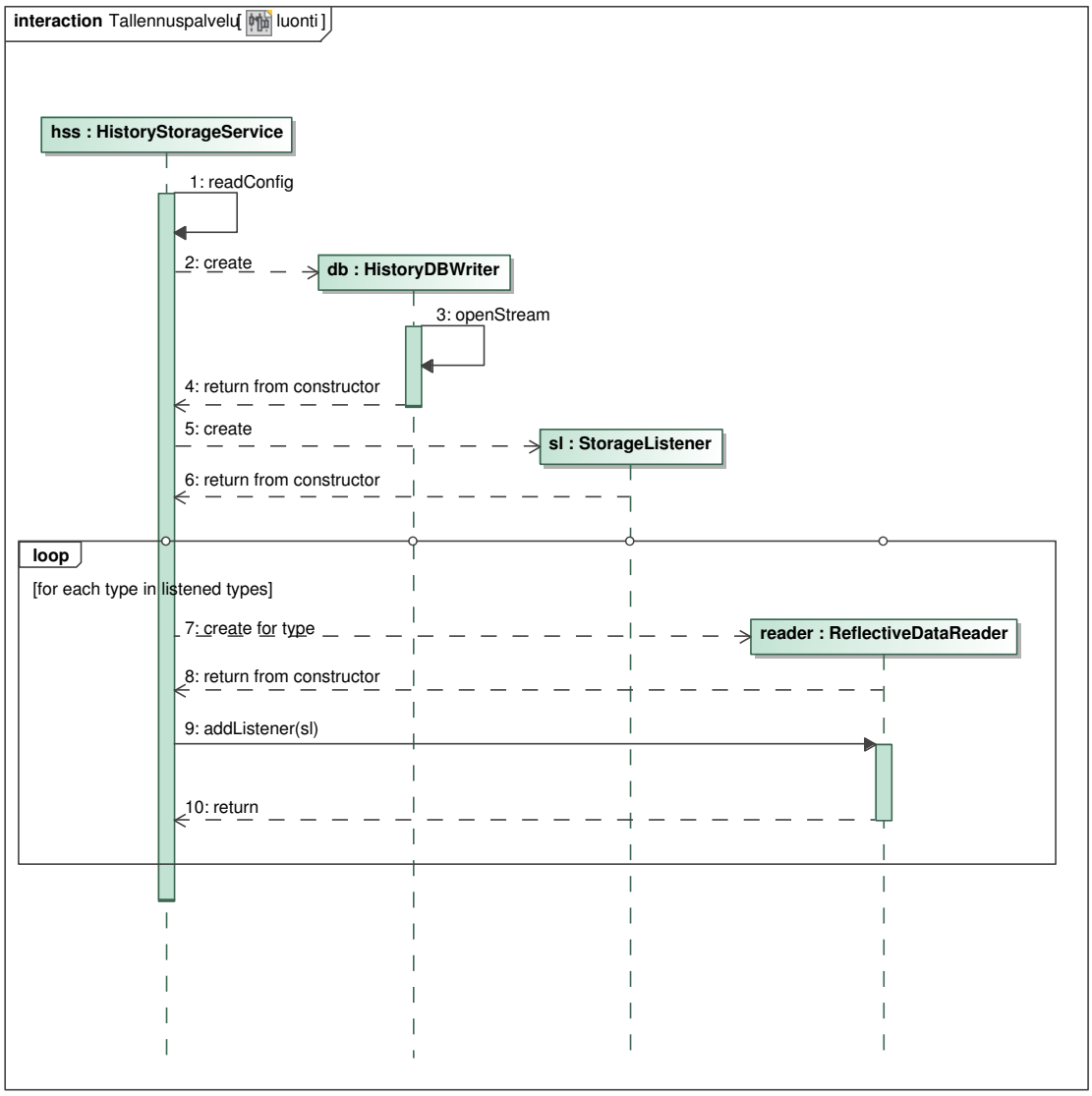
Kuva 4.3: Historian tallennuspalvelu

HistoryStorageService ei itsessään tee alustustoimenpiteiden lisäksi juuri muuta. Se luo HistoryDbWriterin, StorageListenerin ja riittävän määrän lukijoita kuunnellakseen asetustensa mukaisesti väylän liikennettä. Tämän jälkeen se ei tee muuta kun vastailee HistoryDbWriterin pyyntöihin CompleteStateProvider-rajapinnan kautta. Tätä alustussekvenssiä kuvataan kuvassa 4.4.

HistoryDbWriter tallentaa tiedot levylle myöhempää käyttöä varten käyttäen XStream-kirjastoa ja käyttää CompleteStateProvider-rajapintaa aina pyytäessään HistoryStorageServiceltä (joka edelleen pyytää sen ReflectiveDataReaderilta) jokaisen tiedoston alkuun kirjoitettavan vedoksen väylän tilasta. Tarkempi selitys tästä tallennusmuodosta on aliluvussa 4.3.2.

Muut luokat auttavat näitä kahta pääluokkaa tehtävissään. StorageListener on HistoryStorageServicen käyttämä tyyppi, jota se hyödyntää rekisteröidessään kuuntelijan ReflectiveDataReaderille. StorageListener välittää ReflectiveDataReaderilta tulevan tiedon suoraan HistoryDbWriterille kirjoitettavaksi. Element-luokkaa käytetään säilömään taltioitavat tietoalkiot yhdessä metatietonsa kanssa.

Historian tallennuspalvelu käyttää tässä työssä toteutettua DDS-apukirjastoa. Näin ollen palveluun ei tarvitse tehdä kooditason muutoksia haluttaessa muuttaa tallennettavia aiheita, vaan niiden valinta voidaan hoitaa XML-muotoisella asetustiedostolla. Tallennettavien aiheiden tietotyyppeihin liittyvät apuluokat (katso aliluku 3.1.3) täytyy löytyä Java-virtuaalikoneen luokkapolusta tai tallennus epäonnistuu. Näiden luokkien yhteistoimintaa selvitetään paremmin aliluvussa 4.3.

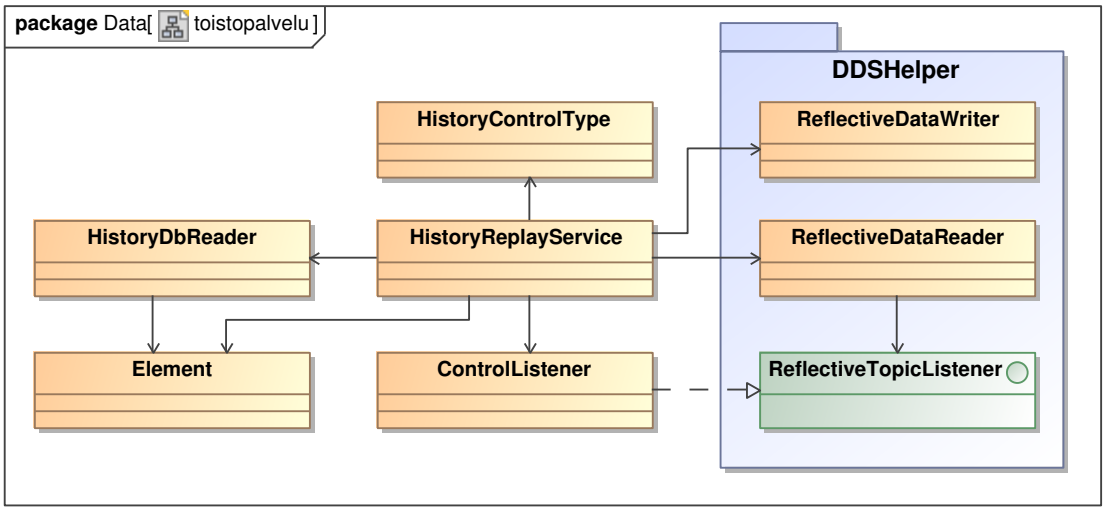


Kuva 4.4: Historian tallennuspalvelun käynnistyminen

4.2.3 Historian toistopalvelu

Historian toistopalvelu HistoryReplayService on myös Springillä käynnistettävissä oleva luokka, eikä vaadi toimiakseen muuta kuin DDS-palvelun käynnissäolon. Sen rakennetta on esitelty luokkakaaviona kuvassa 4.5. Kuten tallennuspalvelu, myös toistopalvelu on toteutukseltaan jaettu kahteen osaan, HistoryReplayServiceen ja HistoryDbReaderiin. Tämä varmistaa että tietokannan toteutusta voidaan helposti kehittää tulevaisuudessa.

Käynnistyessään HistoryReplayService luo ReflectiveDataReaderin ja tähän liittyvän kuuntelijan, ControlListenerin, toistonohjauksen kuuntelemista varten. Tämän jälkeen se jää odottamaan viestejä historiantoistoa kaipaavalta käyttöliitty-



Kuva 4.5: Historian toistopalvelu

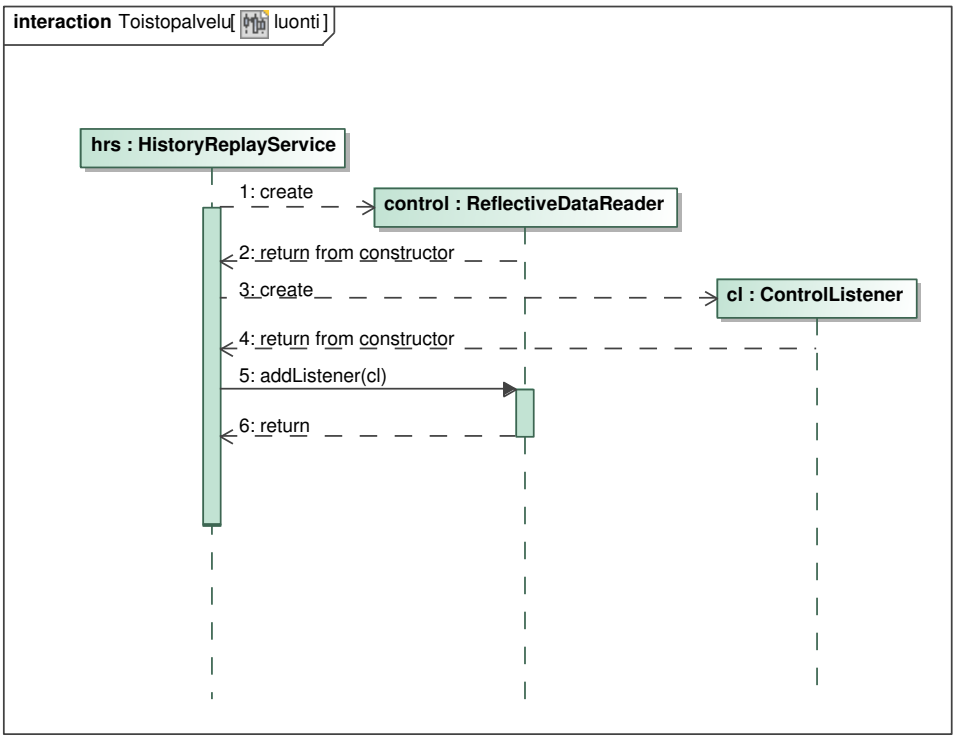
mältä. Tämä alustussekvenssi esitetään kuvassa 4.6. HistoryDbReaderin tehtävä on ladata toistettavaa tietoa tarpeen mukaan levyltä käyttäen XStream-kirjastoa. Tarkempi kuvaus käytetystä tallennusmuodosta löytyy aliluvusta 4.3.2.

Muut luokat tukevat näitä kahta pääluokkaa tehtävissään. HistoryControlType on toiston ohjauksen tietotyyppi ja ControlListener-luokkaa käytetään HistoryReplayServicen rekisteröityessä kuuntelemaan toiston ohjausviestejä ReflectiveDataReaderin kautta. Saapuneet ohjausviestit välitetään HistoryReplayServicelle joka käsittelee ne asianmukaisesti. Tästä käsittelystä ja toistosta kerrotaan tarkemmin aliluvussa 4.4. Element-luokka säilöö taltioidut tietoalkiot yhdessä metatietonsa kanssa.

Historian toistopalvelu käyttää samaa DDS-apukirjastoa kuin tallennuspalvelu, ja pysyy näin riippumattomana tallennetun tiedon sisällöstä tai sen tyypistä. Se ei tarvitse edes XML-muotoista asetustiedostoa, vaan riittää että toistettaviin tyyppeihin liittyvät apuluokat(katso aliluku 3.1.3) löytyvät toistopalvelun luokkapolusta.

4.2.4 Apukirjasto DDS:n käyttöön

Liitteen 1 mukaan toteutetusta Hello World -esimerkistä huomataan, että DDS:n rajapintaa suoraan käyttävä sovellus on rajapinnan suoraviivaisesta luonteesta johtuen yksinkertaisimmillaankin melko pitkä. Sovelluksessa on suoritettava rivikaupalla alustustoimenpiteitä, ja jokainen päivitettävä instanssi on rekisteröitävä erikseen ennen kun päivityksiä voidaan lähettää. Koska ongelma on DDS:n rajapinnan luonteessa, se on olemassa kaikissa ohjelmointikielissä ja ympäristöissä joissa DDS-rajapinta on käytettävissä. Ongelmaa on pyritty ratkaisemaan kehittämällä parempia ohjelmointikielikohtaisia rajapintoja DDS-standardin tulevassa versiossa V1.3, mutta sen julkaisua odotellaan yhä.



Kuva 4.6: Historian toistopalvelun käynnistyminen

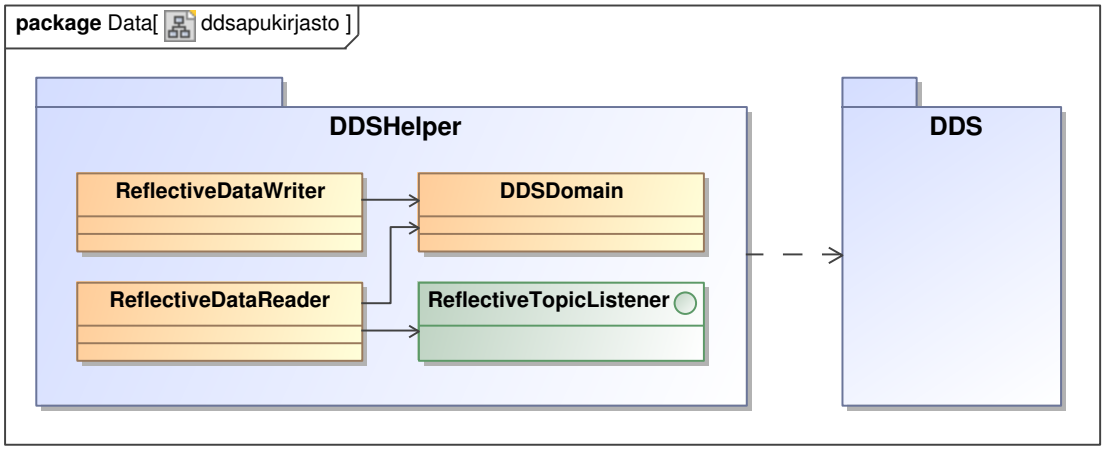
DDS:n ohjelmointikielikohtaisia rajapintoja kehitetään useissa julkisissa projekteissa [22, 23, 24], joista luultavasti tulee aikanaan osa julkistettavaa standardia. Nämä yksinkertaistavat rajapinnan käyttöä valtavasti, ja hyödyntävät ohjelmointikielten geneeriseen ohjelmointiin liittyviä ominaisuuksia selkeyttääkseen rajapintaa. Näillä esimerkkisovellus voitaisiin kutistaa muutamaan riviin.

Kuten aliluvussa 3.1.3 kuvataan, jokainen tietotyyppi jota halutaan käyttää DDS:n kanssa kuvataan IDL-kuvauksella, josta käännetään IDL-kääntäjällä joukko kohdekielen luokkia. Nämä käännetyt luokat ovat vahvasti tyyppisidonnaisia, joten niiden käyttäminen palvelussa suoraan luo käännösaikaisen riippuvuuden palvelusta tiedon tyyppiin. Näin toteutettua ohjelmistoa ei voisi laajentaa jälkikäteen tukemaan uusien tietotyyppien käyttöä väylällä kääntämättä sen ohjelmakoodia uudestaan.

Yrityksessä on jo aiemmin toteutettu Javalla apukirjasto, joka käärii DDS:n sisäänsä käytön selkeyttämiseksi. Kirjasto hyödynsi Javan Generics-ominaisuuksia suoraviivaistaakseen rajapinnan käyttöä. Se ei kuitenkaan sovellu suoraan tässä prototyypissä käytettäväksi, johtuen historiapalvelun tarpeesta olla kooditasolla riippumaton tallennettavan tiedon tyypistä. Reflektion käyttäminen Genericsin sijaan ratkaisee tämän riippuvuusongelman. Tässä työssä toteutettu apukirjasto on toteutettu yhteensopivuussyistä pitkälti aikaisemman apukirjaston mallin mukaan.

Edellistä toteutusta mallina käyttäen toteutettiin apukirjastosta versio joka käyt-

tää sopivien apuluokkien luomiseen Javan reflektio-ominaisuuksia. Reflektiota käyttäen voidaan DDS-väylällä käytettävät tietotyypit(katso aliluku 3.1.3) ja aiheiden nimet valita käännöksen jälkeen historiapalvelun asetustiedostoilla. Tämä on historiapalvelun kannalta merkittävä parannus aikaisempaan toteutukseen, joka olisi vaatinut uuden koodin kirjoittamista jokaista tallennettavaa tyyppiä varten ja palvelun uudelleenkääntämistä tämän jälkeen.



Kuva 4.7: DDS-apukirjasto

Kuva 4.7 esittelee olennaisimmat apukirjaston osat. DDSDomain abstrahoi DDS-toimialueeseen liittyvät toiminnot, kuten käytettävän osion. ReflectiveDataWriter abstrahoi kirjoittamiseen liittyvät toiminnot, ReflectiveDataReader puolestaan lukemiseen liittyvät toiminnot. ReflectiveDataReaderille voidaan rekisteröidä tarkkailijamallin mukaisesti ReflectiveTopicListener-rajapinnan toteuttavia luokkia tarkkailemaan uuden tiedon saapumista.

```
1 import com.foo.dds.ReflectiveDataWriter;
2
3 ReflectiveDataWriter dw = new ReflectiveDataWriter("HelloTopic",
4     "com.foo.model.HelloType");
5 dw.write(new com.foo.model.HelloType("Hello World!"));
```

Listaus 4.3: Kehitetyn DDS-apukirjaston käyttö: tuottajakomponentti

Listauksista (4.3, 4.4) huomataan, että tätä kirjastoa käyttämällä DDS-sovellus kutistuu muutamaan riviin, ja käännösaikainen riippuvuus väylällä käytettävistä tietotyypeistä rajoittuu ainoastaan tiedon luontiin ja käsittelyyn. Toisin sanottuna apukirjaston koodissa ei käännösaikana viitata väylällä kulkeviin tietotyyppeihin. Historiapalvelussa tiedon luonti ja käsittely tapahtuu sarjallistamisen kautta. Sarjallistaminen käyttää myös reflektiota, joten koko historiapalvelu voi olla käännösaikaisesti riippumaton käytetyistä tietotyypeistä.

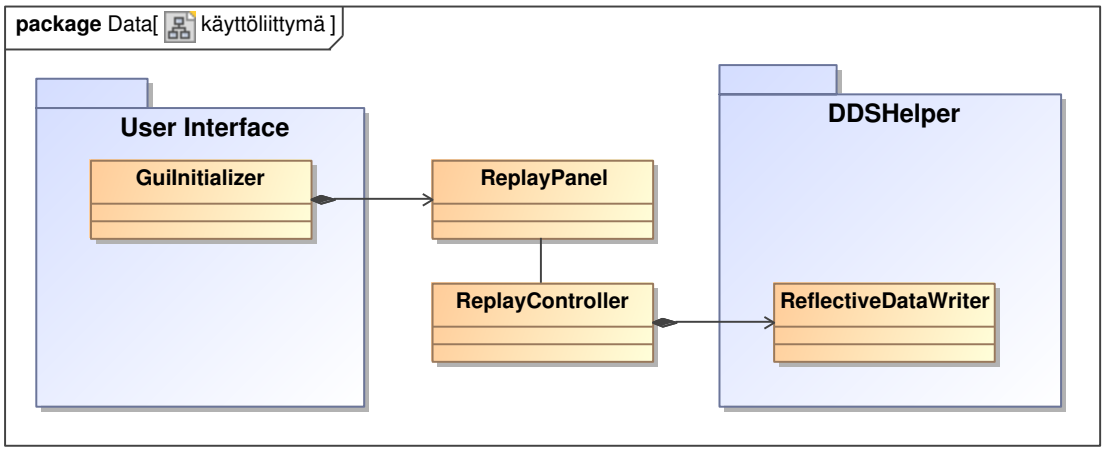
```
1  import com.foo.dds.ReflectiveDataReader;
2
3  ReflectiveDataReader dr = new ReflectiveDataReader("HelloTopic",
4      "com.foo.model.HelloType");
5  dr.registerListener(new ReflectiveTopicListener(){
6      @Override
7      void updateReceived(Object instance, SampleInfo info) {
8          try {
9              System.out.println((com.foo.model.HelloType) instance);
10         } catch (Exception e) {
11             // Alkio on tyyppi oli odottamaton -> ei huomioida alkiota
12         }
13     }
14 });
```

Listaus 4.4: Kehitetyn DDS-apukirjaston käyttö: tilaajakomponentti

Reflektio Javassa on kuitenkin dynaamisuutensa takia joiltain osin hitaampaa kuin olioiden suora rakentaminen ja suorien metodikutsujen käyttö. DDS:n operaatiot ovat kuitenkin suhteellisen raskaita, joten reflektion vaikutus tehokkuuteen on todennäköisesti melko pieni. Sen vaikutuksia suorituskykyyn tutkitaan tarkemmin osana lukua 5.

4.2.5 Karttakäyttöliittymä

Yrityksessä on aiemmin toteutettu yhtenä kohdejärjestelmän osana karttakäyttöliittymä, joka näyttää ilmatilannekuvaa ja tietoa siinä näkyvistä seurannoista. Tämä karttakäyttöliittymä on toteutettu käyttäen mm. Springiä (aliluku 3.4), Javan Swing-kirjastoa yhdessä JIDE-komponenttien kanssa (aliluku 3.5) ja Luciadin karttakomponenttia(aliluku 3.5).



Kuva 4.8: Käyttöliittymään lisätyt komponentit

Kuvassa 4.8 kuvataan käyttöliittymään tehtyjä lisäyksiä luokkakaavion muodossa. Käyttöliittymään lisättiin Swing-komponentti ReplayPanel toiston ohjausta varten ja ReplayController hoitamaan kommunikaatiota ohjauskomponentin ja toistopalvelun välillä. Käyttöliittymän rakentavaa GuiInitializer-luokkaa muokattiin lisäämään ReplayPanel osaksi käyttöliittymää. Toistopalvelun kanssa kommunikoidaan käyttäen samaa DDS-väylää, jota käytetään myös seurantatiedon siirtoon. ReplayControllerin käynnistymisestä sovelluksen käynnistymisen yhteydessä huolehtii Spring (katso aliluku 3.4).

4.2.6 Simulaattori

Järjestelmän osaksi on jo aiemmin toteutettu simulaattori, joka lähettää simuloitua seurantatietoa DDS-väylälle. Simulaattorille voidaan asettaa haluttu simuloitujen kohteiden määrä ja sekunnissa tapahtuvien päivitysten määrä. Päivitykset lähetetään halutun osion seurantatietoon liittyviin aiheisiin ja ovat muodoltaan samankaltaisia kun todellinen järjestelmään tuleva seurantatieto.

Simulaattorilla voidaan myös vaikuttaa paljon siihen, minkälaisia simuloidut kohteet ovat. Kohteiden tyyppiä ja sijaintia voidaan rajoittaa helposti. Nämä ominaisuudet eivät kuitenkaan ole tässä työssä olennaisia, sillä kaikki tieto tallennetaan sen sisällöstä riippumatta.

4.3 Historiatiedon tallentaminen

Historian tallennuspalvelu on käynnistymisensä jälkeen valmis tallentamaan väylällä liikkuvaa tietoa myöhempää käyttöä varten. Seuraavissa aliluvuissa käsitellään tiedon syntymistä, ja päätymistä tallennuspalvelun kautta levylle talteen.

4.3.1 Tiedon syntyminen ja kerääminen

Tallennettava tieto voi periaatteessa syntyä missä vaan. Tyypillisessä toimintaympäristössä raakatietoa tuottaa esimerkiksi tutka. Rakennetun prototyypin ympäristössä tiedon luomisesta vastaa simulaattori, joka simuloi tilannekuvan seurantojen liikkumista, sekä seurantoihin liittyvän lisätiedon, kuten seurattavan kohteen tyypin, päivittymistä. Näin syntynyt tieto kirjoitetaan DDS-väylälle seurantatietoa välittäviin aiheisiin (tiedosta tarkemmin aliluvussa 4.2.1). Järjestelmän normaalissa käytössä tätä tietoa voidaan seurata ja tutkia karttakäyttöliittymää käyttäen.

Historian tallennuspalvelu liitetään järjestelmään DDS-väylän kautta samalla tavalla kun mikä tahansa sovellus liitettäisiin. Tämän jälkeen se aloittaa automaattisesti asetustensa mukaisten aiheiden tallentamisen myöhempää toistoa varten. Tietolähteen ei tarvitse millään tavoin ottaa huomioon tallennuspalvelun olemassaoloa.

Järjestelmän maantieteellisesti hajautettu luonne(katso aliluku 2.3) voi aiheuttaa verkkokatkoja, jotka voitaisiin huomioida ylläpitämällä useampaa tallennuspalvelua, ja yhdistämällä niistä saadut tallenteet myöhemmin yhteyksien palattua. Tässä työssä ei kuitenkaan oteta kantaa tallennetiedostojen yhdistämiseen.

4.3.2 Tiedon tallentaminen

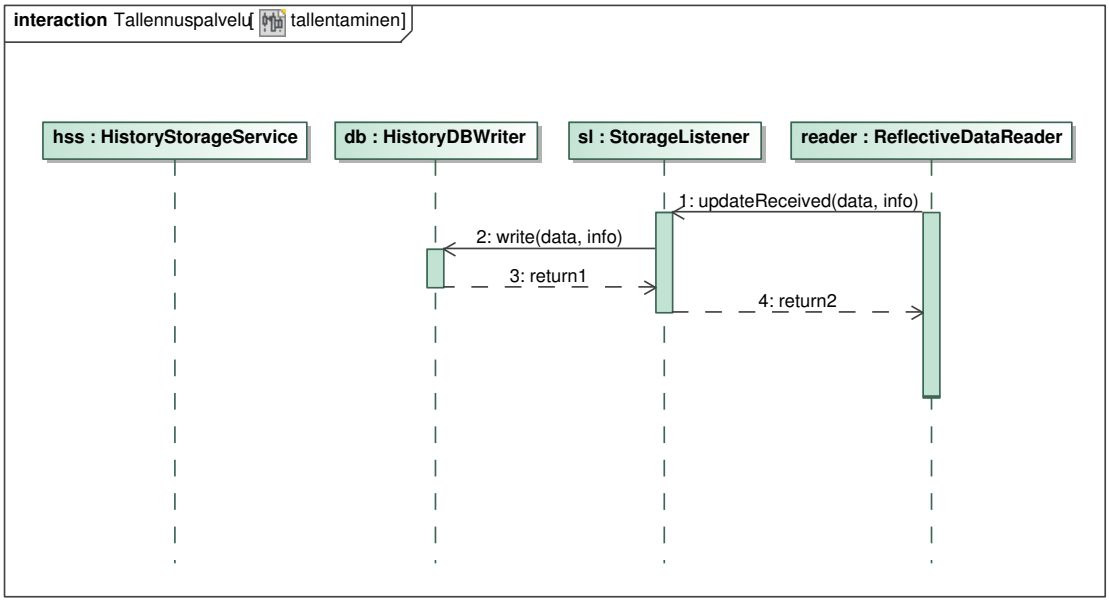
Käynnistyessään tallennuspalvelu lukee asetustiedostoistaan tallennettavaksi haluttujen aiheiden nimet ja luo jokaista aihetta kohden DDS-apukirjastoa käyttäen lukijan. Jokaiselle lukijalle rekisteröidään kuuntelijaksi instanssi StorageListeneristä. Kuuntelija antaa vastaanottamansa päivitykset HistoryDbWriterille tallennettavaksi. Kuva 4.9 esittää lukijan, kuuntelijan ja HistoryDbWriterin yhteistoiminnan sekvenssikaaviona.

Tiedon saapuessa HistoryDbWriterin käsiteltäväksi se sarjallistetaan ensin käyttäen XStream-kirjastoa(katso aliluku 3.3.3). Sarjallistamisen jälkeen näytteet tallennetaan saapumisjärjestyksessä tiedostoihin palvelimen levyllä, asetustiedostolla määriteltävissä olevaan hakemistoon. Yhteen tiedostoon tallennetaan oletuksena korkeintaan yhden päivän verran tai asetustiedoston mukainen määrä tietoa. Yhden tiedoston tultua täyteen, jatketaan tallentamista automaattisesti seuraavaan. Tiedostot nimetään tallennusajankohdan mukaan muotoon YYYYMMDDhhmmss, sekunnin tarkkuudella.

Tiedon kanssa yhdessä tallennetaan sen kanssa saapunut SampleInfo-instanssi, joka sisältää DDS-väylän kertomaa metatietoa tietopaketista. Tästä metatiedosta olennaisin on SampleInfon sisältämä tiedon saapumisen aikaleima. Toistopalvelu käyttää tätä aikaleimaa ajoittaessaan tiedon toiston vastaamaan tallennettua. Lisäksi tallennetaan tiedon tyypin nimi. Toistopalvelu käyttää tyyppinimeä luodakseen oikeanlaisen DDS-kirjoittajan. Muuta metatietoa tiedosta on vaikeata tallentaa yleispätevästi, koska tiedon sisältöä ei tiedetä ennalta.

Koska väylän tieto on luonteeltaan muutosviestejä, ei pelkästään tallennuksen aloittamisen jälkeen saapuneiden viestien tallentaminen riitä kertomaan järjestelmässä olevan tiedon täydellistä tilaa. Tilaa muodostettaessa täytyy olla saatavilla myös ennen tallennushetkeä tapahtuneiden päivitysten tulokset. Tämä ilmenisi ongelmana esimerkiksi tilanteessa jossa tietyn seurannan sijaintiin saadaan päivitys 10 sekuntia ennen tallennuksen aloittamista ja seuraavan kerran vasta 30 sekuntia tallennuksen aloittamisen jälkeen. Tällöin kohteella ei olisi ensimmäiseen 30 sekuntiin lainkaan paikkatietoa, sillä ennen tallennuksen aloittamista tapahtunut päivitys ei olisi nähtävissä tallennetun tiedon toistossa.

Tämä voidaan ratkaista siten että jokaisen tiedoston alkuun tallennetaan vedos tallennettavien aiheiden jokaisen kohteen tuoreimmasta näytteestä tiedoston luontihetkellä. Käytännössä tiedostoon siis tallennetaan väylällä olevan voimassaolevan



Kuva 4.9: Historian tallennus tallennuspalvelussa

tiedon kokonaistila. Näin saadaan aloitettua toisto ehjästä tilasta, joten toistokin toimii oikein.

Tiedostot pakataan ja puretaan lennossa GZIP-muotoon käyttäen java.util.zip -pakkauksen luokkia GZIPOutputStream ja GZIPInputStream. [25] Tällä on huomattava pienentävä vaikutus tietokannan tilan käyttöön, mutta myöskin jonkinlainen vaikutus palvelun suoritinajan käyttöön. Pakkauksen vaikutuksia suorituskykyyn tutkitaan tarkemmin luvussa 5.

4.4 Historiatiedon esittäminen

Historiatiedon toistopalvelu lukee tiedon taltiosta, ja toistaa sen väylälle toistokäyttöliittymän käskyjen mukaisesti. Seuraavissa aliluvuissa käsitellään toistopalvelun ohjaamista ja itse toistoa.

4.4.1 Tiedon tilaaminen historian toistopalvelulta

Toistopalvelun ohjausta varten on DDS-väylällä oma aihe, jolla käytetään toiston ohjaamiseen suunniteltua tietotyyppiä. Listauksessa 4.5 esitetty tietotyyppi sisältää toimintokoodin, toistajan id:n ja lisätietokentän, jota voidaan käyttää muun muassa toistonopeuden määrittämiseen.

Karttakäyttöliittymään on toteutettu komponentti toistopalvelun ohjausta varten. Sillä voidaan lähettää käskyjä toistopalvelulle. Nämä käskyt kulkevat DDS-väylää pitkin toistopalvelulle, joka aloittaa tai lopettaa toiston käskyjen mukaan.

```
1  module history {
2  module control {
3  module dds {
4      struct HistoryControlType {
5          long long sessionId;
6          char action;
7          long long timestamp;
8          string extrainfo;
9      };
10 #pragma keylist HistoryControlType sessionId
11 };
12 };
13 };
```

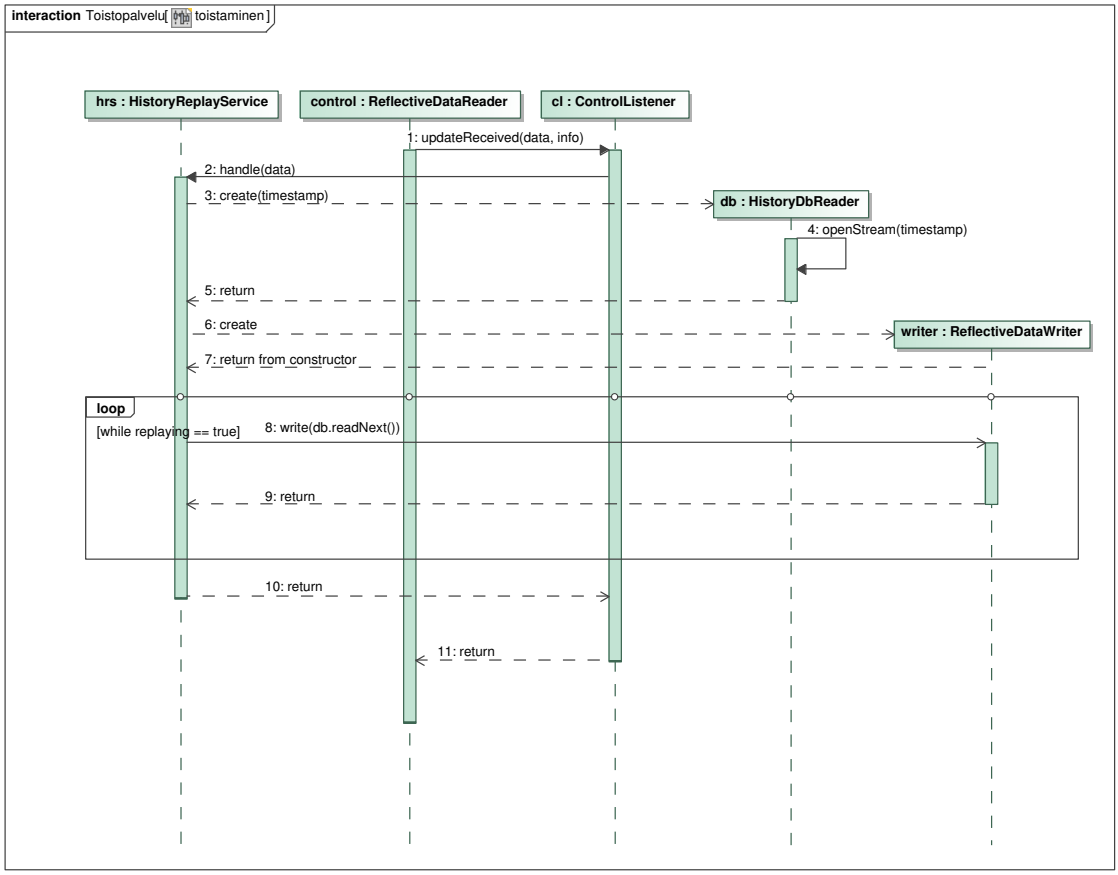
Listaus 4.5: Toistonohjauksen tietotyyppi

Jokaisella käyttöliittymällä on historian toistoa varten oma osio järjestelmässä, jotta samanaikaiset toistajat saadaan pidettyä erillään toisistaan. Osion tunnuksena käytetään käyttöliittymän DDS-tilaajakomponentin yksikäsitteistä id:tä (katso aliluku 3.1.3).

Kuvassa 4.10 esitetään toiston alkaminen sekvenssikaaviona. Toistokäskyn saavuttua käsiteltäväksi toistopalvelu luo instanssin HistoryDbReaderista, joka etsii pyydetyn ajankohdan sisältävän tiedoston ja avaa sen luettavaksi. Tämän jälkeen luodaan ReflectiveDataWriter, jonka avulla toistopalvelu aloittaa toistamaan historiaa sitä pyytäneen sovelluksen kuuntelemaan osioon. Historian toisto tapahtuu tiedosto kerrallaan Javan ja XStreamin virtaominaisuuksia käyttäen. Tiedostovirran sisältämä pakattu virta puretaan lennosta, ja sen sisältämät sarjallistetut tietorakenteet ladataan virrasta yksitellen muistiin toistoa varten. Näytteet toistetaan saman nimisiin aiheisiin joista ne on tallennettu ja toisto ajoitetaan tiedon tallennusajan mukaan niin, että näytteiden väliset aikaerot säilyvät ennallaan. Sekvenssikaaviosta on yksinkertaistuksen vuoksi jätetty pois aliluvussa 4.3.2 kuvattu alkutilanteen lataus ja toisto, sillä sen toteutus on hyvin samankaltainen muun toiston kanssa.

Alkutilanteen latauksen ratkaisemaa ongelmaa vastaava ongelma syntyy myös, jos halutaan aloittaa toisto jostain muusta kohdasta kun tiedoston alusta. Luonnollisin tapa ratkaista tämä ongelma olisi ladata alkutilanne toistopalvelun muistiin ja lukea tallennetta pyydettyyn toiston aloitushetkeen asti analysoiden jokaisen tallennetun päivityksen sisältöä niin, että voidaan muodostaa väylän kokonaistila toiston aloitushetkellä. Tämä vaatisi kuitenkin toistopalveluun lisää logiikkaa päättelemään mitkä päivitykset kohdistuvat mihinkin tietoon. Koska toistopalvelun ei muuten tarvitse sisäisesti muodostaa kokonaiskuvaa väylän tilasta, toisi tämä ratkaisutapa merkittävästi lisää monimutkaisuutta toistopalveluun. Siksi jätämme tämän ratkaisutavan tulevaa kehitystä varten, ja tyydymme toteutusta yksinkertaistavaan ratkaisuun.

Toteutusta yksinkertaistava tapa ratkaista kesken tiedostoa aloitettavan toiston



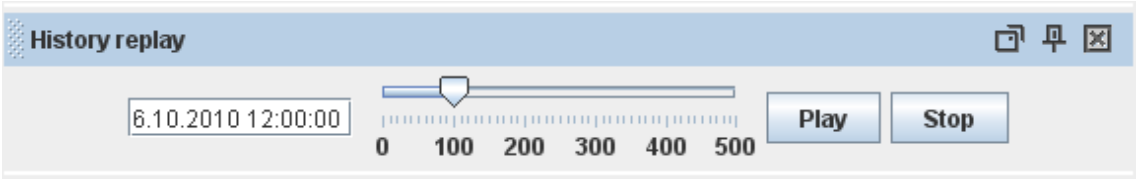
Kuva 4.10: Historian toisto toistopalvelussa

ongelma on pakottaa toisto alkamaan aina tiedoston alusta. Tämän jälkeen toistetaan tallennetta "pikakelauksella", eli mahdollismman nopeasti päivitysten välistä aikaeroa huomioimatta, aina pyydettyyn aloitusaikaan asti. Väylällä kulkevan tiedon ja näin ollen myös tietoa tarkkailevan karttakäyttöliittymän näkökulmasta tämä nopeutettu toisto näyttää vastaavalta kuin elokuvan pikakelaus. Tällä tavoin saadaan kuitenkin väylällä oleva tieto vastaamaan pyydetyn toiston aloitushetken tilaa, ja toistoa voidaan jatkaa siitä hetkestä eteenpäin normaalisti.

4.4.2 Käyttöliittymä

Historian toistoa varten käyttöliittymä täytyy käynnistää erilliseen historiatilaan, jossa se ottaa vastaan reaaliaikaisen tiedon sijasta historiapalvelulta saamaansa tietoa. Tämä tapahtuu käytännössä muuttamalla käyttöliittymän asetustiedostoja niin, että se kuuntelee DDS-väylällä reaaliaikaisen osion sijasta omaa historiantoisto-osiotaan. Lisäksi käyttöliittymästä voidaan ohjata historiantoistoa.

Toiston ohjausta varten toteutettiin Javan Swing-kirjastolla komponentti kontrollereineen jo olemassaolevaan karttakäyttöliittymään. Kuvassa 4.11 esitellään yk-



Kuva 4.11: Kuva toiston ohjauskomponentista

sinkertainen käyttöliittymä, jossa on painikkeet toiston aloittamiselle ja lopettamiselle, tekstikenttä toiston alkuajan määrittämistä varten, sekä liukusäädin toiston nopeuden valintaa varten. Toiston nopeus valitaan prosentteina tallennusnopeudesta, 100% ollessa oletusvalinta.

Painettaessa Play-nappia kontrolleri rakentaa HistoryControlType-olion ja asettaa siihen muuttujan sessionId arvoksi käyttöliittymän DDS-tilaajakomponentin yksikäsitteisen tunnisteen (katso aliluku 3.1.3), muuttujan action arvoksi merkin 'p' kuvaamaan toistoa, muuttujan timestamp arvoksi halutun toiston alkuajankohdan ja muuttujan extraInfo arvoksi halutun toistonopeuden. Painettaessa Stop-nappia asetetaan vain sessionId samoin kun Playn tapauksessa ja action 's' -merkiksi. Toiston aloituksen jälkeen käyttöliittymässä näkyy tallennettua tietoa, alkaen määritellystä toiston alkuajasta ja toistuen määritellyllä nopeudella reaaliaikaan verrattuna.

5. PROTOTYYPIN SUORITUSKYKY

Tärkeä osa työtä oli prototyypin suorituskyvyn testaus ja testitulosten arviointi. Seuraavissa aliluvuissa käydään ensin läpi testimenettelyt, sen jälkeen itse testit ja viimeisenä arvioidaan saatuja tuloksia.

5.1 Suorituskyvyn testaus

Testien pääasiallisena tarkoituksena oli selvittää historiapalvelun suorituskykyisyyttä ja sitä onko tällainen historiapalvelu suorituskyvyn näkökulmasta järkevää, tai edes mahdollista, toteuttaa. Samalla tutkittiin myös järjestelmään liittyvien alustakomponenttien vaikutusta suorituskykyyn ja pyrittiin näin löytämään mahdollisia pullonkauloja järjestelmän suorituskyvystä.

5.1.1 Testijärjestelmä

Testijärjestelmään kuului kaksi konetta, jotka oli kytketty toisiinsa sadan megabitin Ethernet-verkolla. Koneiden tärkeimmät ominaisuudet on esitetty taulukossa 5.1.

Taulukko 5.1: Testijärjestelmän koneet

Nimi	Käyttöjärjestelmä	Prosessori	Keskusmuisti
Kone 1	Linux	2 x Intel Xeon 2 GHz	4GB
Kone 2	Solaris	2 x UltraSPARC T2+ 1,2 GHz	32GB

5.1.2 Testimenettelyt

Testejä varten DDS-palvelu asetettiin käyttämään 128 megatavua muistia. Tämän todettiin riittävän reilusti kaikkien testitapausten suorittamiseen ongelmitta. Testit joissa käytettiin DDS-väylää suoritettiin niin että julkaisijaa ja tilaajaa ajettiin eri koneilla.

Ensin testattiin historian tallennuspalvelun kapasiteettia niin, ettei se ollut liitettynä lainkaan DDS-väylään. Tämä testi suoritettiin molemmilla testikoneilla. Tämän jälkeen testattiin pelkästään DDS-väylän suorituskykyä käyttäen reflektiivistä ja ei-reflektiivistä versiota DDS-apukirjastosta (katso aliluku 4.2.4). Nämä testit suoritettiin käyttäen molempia koneita vuoroin tuottajan ja tilaajan rooleissa. Näin

saatiin muodostettua käsitys sovelluksen osien suorituskyvystä eri testikoneilla, jonka pohjalta koneiden roolit valittiin viimeiseen testiin sopivasti.

Viimeisessä testissä testattiin historiapalvelun tallennuskapasiteettia niin, että se oli kytkettynä normaalisti DDS-väylälle ja tallennettavat näytteet saapuivat sille väylän kautta. Tämä testi suoritettiin niin, että edellisissä testeissä nopeammaksi osoittautunut kone toimi tiedon tuottajana, ja hitaampi tiedon tilaajana ja tallentajana. Roolit valittiin näin päin, jotta saataisiin luotettavasti testattua tallennuksen suorituskykyä, ilman että tarvitsee huolehtia tiedon tuottajan mahdollisista suorituskykyrajotteista.

5.1.3 Historiataltion tallennuksen suorituskyvyn testaus

Ensin testattiin historiataltion tallennuskapasiteettia yksinään, ilman DDS-väylän aiheuttamaa vaikutusta tuloksiin. Testiä varten kirjoitettiin pääohjelma, joka kirjoittaa mahdollisimman nopeasti tietoa historiataltioon käyttäen HistoryDbWriter-luokkaa (katso aliluku 4.2.2). Lisäksi pääohjelma piti kirjaa tallennetun tiedon määrästä ja suorituksen päättyessä laski keskiarvon tallennettujen näytteiden määrästä sekuntia kohden.

Testiä ajettiin molemmilla testikoneilla 120 sekunnin ajan, HistoryDbWriterin käyttäessä pakattua ja pakkaamatonta taltiota. Odotettu tulos olisi että pakattu taltio olisi molemmilla koneilla jonkin verran hitaampi, ja kone 1 olisi suuremman kellotaajuutensa vuoksi konetta 2 nopeampi. Lopulliset keskiarvot on esitetty taulukossa 5.2. Tuloksista ei laskettu erikseen hajontaa, mutta sen havaittiin testien aikana olevan suhteellisen pientä.

Taulukko 5.2: HistoryDbWriterin tallennussuorituskyky tallennettuina päivityksinä sekunnissa

	pakattu taltio (päivityksiä/sekunti)	pakkaamaton taltio (päivityksiä/sekunti)
Kone 1	8550.96	189.55
Kone 2	1165.19	1255.74

Tuloksissa huomataan yksi merkittävä poikkeus odotuksista, korostettuna harmaalla pohjavärillä taulukossa. Koneella 1 pakkaamattoman taltion tulos on merkittävästi heikompi kun pakatulla taltiolla ja myös merkittävästi heikompi kun koneen 2 vastaava tulos. Tämän työn puitteissa ei saatu selville mistä tämä selkeästi heikompi tulos johtuu. Tulosta ei osata selittää, sillä tätä testiä ajaessa ei koneessa havaittu mitään syytä huonoon suorituskykyyn. Suoritinkuorma pysyi matalana ja myöskään levyjärjestelmälle aiheutuva kuorma ei vaikuttanut merkittävältä. Kaikissa muissa testitapauksissa yhden ytimen suoritinkäyttö nousi nopeasti testin aloittamisen jälkeen sataan prosenttiin, ja pysyi siellä koko testin ajan. Todetaan siis

tämä testitulos kelvotottomaksi tarkemmin tuntemattomasta syystä, hylätään se ja tutkitaan kolmea muuta tulosta.

Koneella 2 pakkaamattoman taltion tulos on noin 8% parempi. Tämä on hieman odotettua vähemmän, mutta kuitenkin helposti uskottavissa. Koneen 1 tulos pakatun taltion tallentamisessa oli yli 7-kertainen koneen 2 tulokseen verrattuna. Tämä on odotettua suurempi ero, sillä koneen 1 kellotaajuus on vain noin kaksinkertainen koneeseen 2 verrattuna. Ilmeisesti koneen 2 suoritinarkkitehtuuri ei sovellu yhtä hyvin yksisäikeisten ohjelmien ajamiseen.

Näiden tulosten perusteella voidaan todeta että vaikka pakkaamaton taltio on nopeampi, nopeusero ei ole kovinkaan merkittävä. Molemmissa tapauksissa yhden ytimen prosessorinkäyttö kohosi sataan prosenttiin, josta voidaan päätellä että suurin osa prosessorikuormasta johtuu jostain muusta kuin taltion pakkaamisesta. Tämä kuorma aiheutuu todennäköisesti XStream-kirjaston XML-sarjallistamisesta, sillä muita huomattavasti kuormittavia operaatioita ei tallennukseen liity.

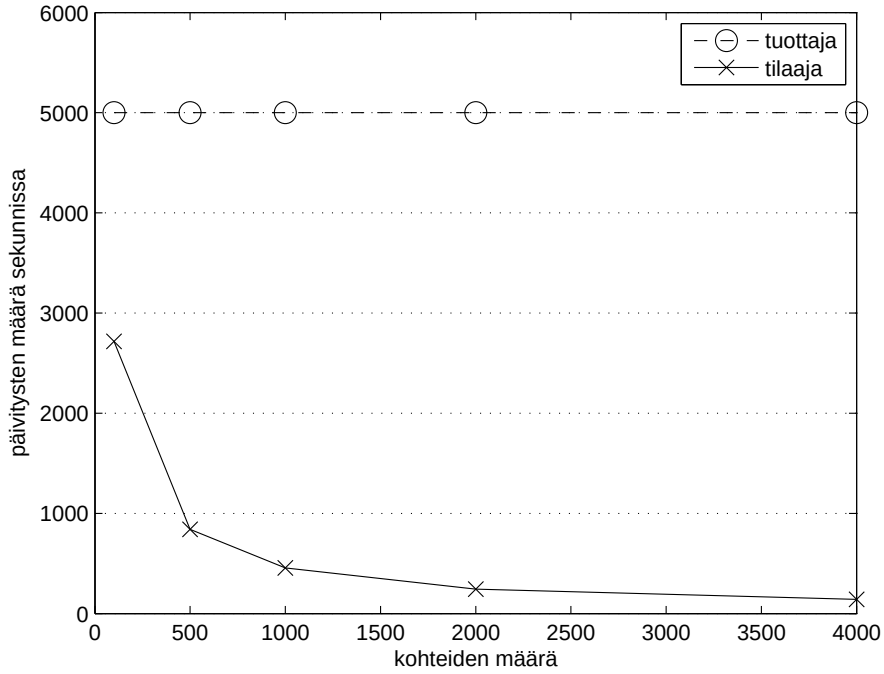
Huomioiden pakkauksen aiheuttaman hyvin vähäisen lisäkuorman ja lisäksi pakkaamattoman taltion toisella testikoneella aiheuttamat selittämättömät ongelmat, voidaan sanoa että taltion pakkauksen käyttäminen on perusteltua kaikissa testeissä.

5.1.4 DDS-komponentin suorituskyvyn testaus

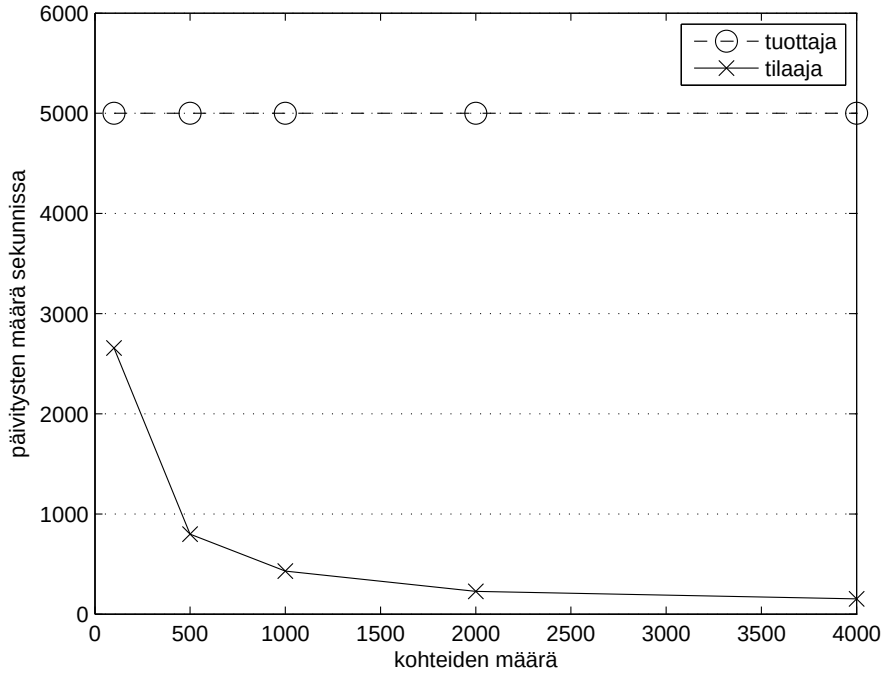
DDS-komponentin testauksessa käytettiin erillisiä tuottaja- ja tilaajakomponentteja. Testi ajettiin useaan kertaan eri kohdemäärillä, sillä kohteiden määrä vaikuttaa huomattavasti tilaajakomponentin suorituskykyyn. Tuottajakomponentti päivitti silmukassa jokaista DDS-väylällä olevaa kohdetta mahdollisimman nopeasti. Tilaajakomponentti vastaanotti päivityksiä mahdollisimman nopeasti ja heitti vastaanottamansa tiedon pois. Molemmat komponentit pitivät kirjaa keskimäärin sekunnin aikana käsittelemistään päivityksistä. Näin pyrittiin saamaan selville suurin mahdollinen päivitysmäärä joka sekunnin aikana onnistuneesti läpäisee kunkin DDS-komponentin kullakin testikoneella. DDS-rajapinnan suorituskykyä testattiin erikseen reflektiivisellä ja ei-reflektiivisellä toteutuksella. Testit ajettiin ensin niin että kone 1 toimi tuottajana ja kone 2 tilaajana ja sen jälkeen roolit vaihtaen.

Testauksen tulokset esitellään kuvissa 5.1–5.5. Koneen 1 viivasuorat tulokset sen toimiessa tuottajan roolissa johtuvat siitä, että kirjoitusnopeudelle jouduttiin asettamaan rajoitus. Ilman rajoitusta tiedon tuottamisnopeus oli liian suuri tilaajana toimivalle koneelle 2 ja siinä ajettu testisovellus kaatui ennen loppuun pääsemistä. Kohdemäärien ja toteutusten välisiä eroja kirjoittajan roolissa voidaan kuitenkin arvioida tuloksista, joissa kone 2 toimi tuottajana.

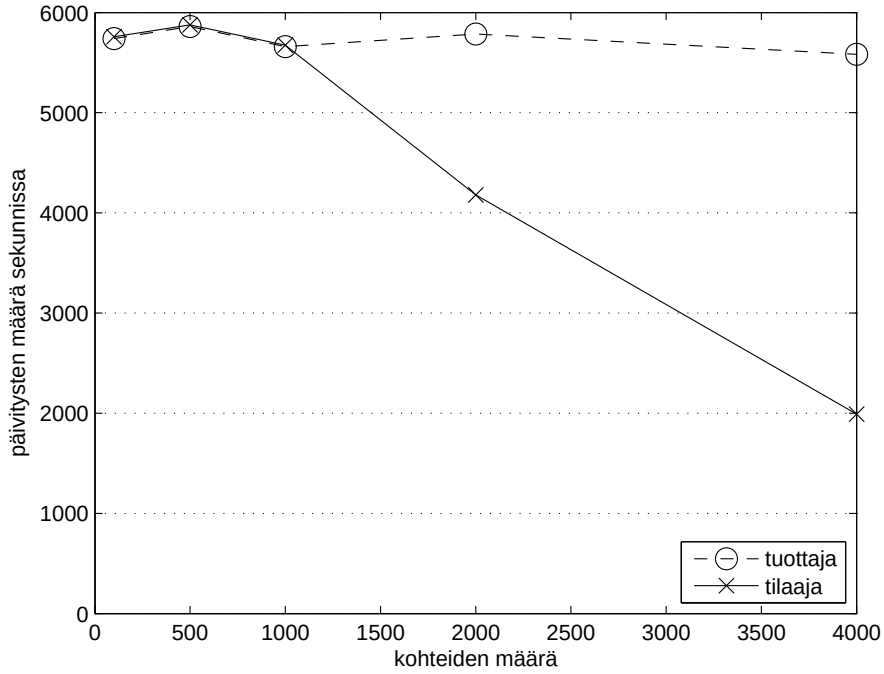
Kun vertaillaan kuvassa 5.5 näkyvän erotuksen avulla reflektiivisen ja ei-reflektiivisen version tuloksia toisiinsa, huomataan eron olevan suhteellisen pieni. Kummassakaan roolissa ei voida selvästi sanoa toisen toteutuksen olevan toista nopeampi,



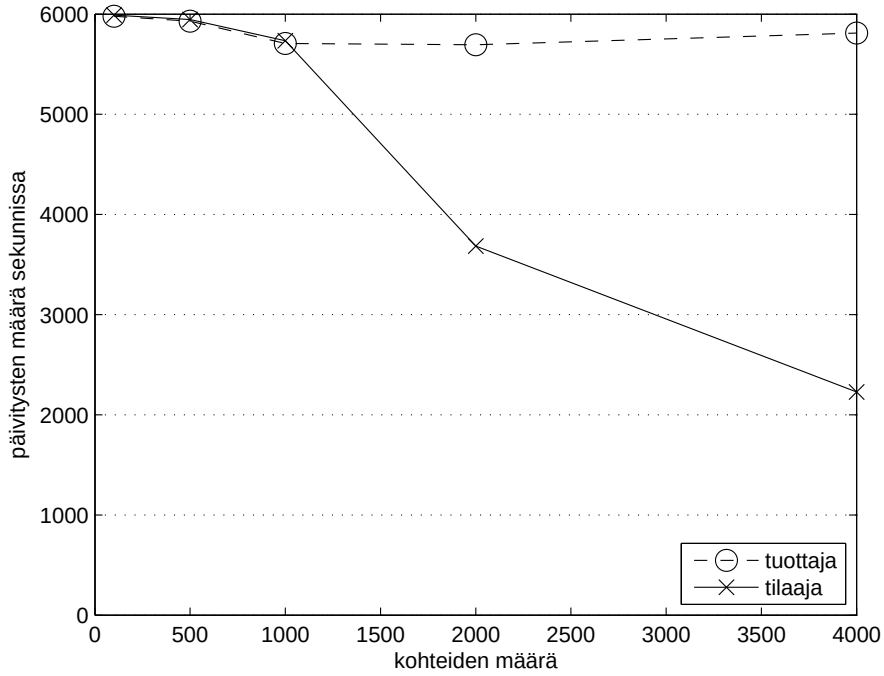
Kuva 5.1: Reflektiivisen DDS-komponentin suorituskyky, tiedon kulkusuunta koneelta 1 koneelle 2



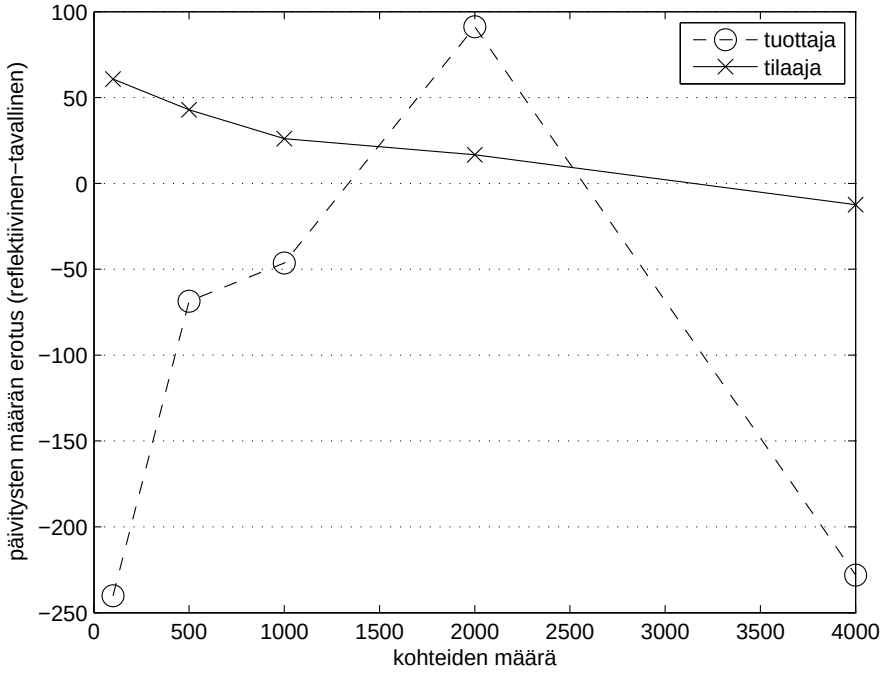
Kuva 5.2: Ei-reflektiivisen DDS-komponentin suorituskyky, tiedon kulkusuunta koneelta 1 koneelle 2



Kuva 5.3: Reflektiivisen DDS-komponentin suorituskyky, tiedon kulkusuunta koneelta 2 koneelle 1



Kuva 5.4: Ei-reflektiivisen DDS-komponentin suorituskyky, tiedon kulkusuunta koneelta 2 koneelle 1



Kuva 5.5: Reflektiivisen ja ei-reflektiivisen DDS-komponentin suorituskykyjen erotus, laskettuna koneen 2 tuloksilla

vaan tulokset vaihtelevat molempiin suuntiin riippuen näytteiden määrästä. Tällä perusteella voidaan sanoa että toteutusten välillä ei ole sellaista merkittävää eroa, joka selkeästi esittäisi toisen toteutuksista paremmassa valossa.

Tuloksista huomataan myös että kone 1 oli riittävän nopea tuottamaan riittävästi näytteitä kone 2:n käsiteltäväksi kaikilla testatuilla kohdemäärillä, kun taas kone 2 pystyi pitämään koneen 1 kiireisenä ainoastaan kun kohteiden määrä nostettiin yli tuhanteen. Näiden tulosten pohjalta voidaan valita historiapalvelun tallennuksen suorituskyvyn testausta varten tiedon tuottajan rooliin nopeampana kone 1 ja tilaajan/tallentajan rooliin kone 2. Näin voidaan varmistua siitä, että tiedon tuottajan kapasiteetin puute ei vaikuta tallentajan testauksen tuloksiin.

5.1.5 Historiapalvelun tallennuksen suorituskyvyn testaus

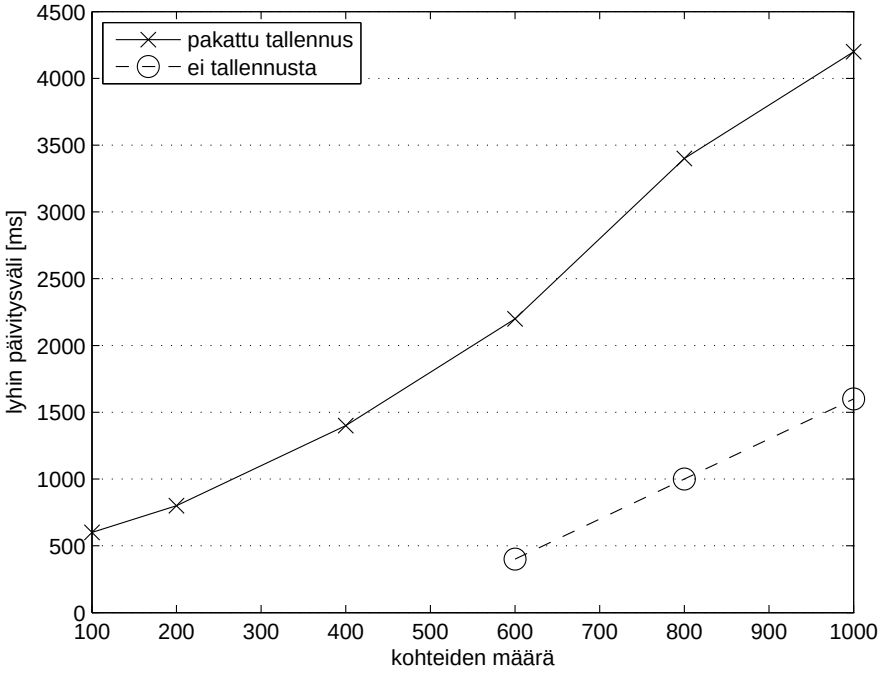
Näillä testeillä pyrittiin arvioimaan sitä aikaa, joka tallennuspalvelulta kuluu tietyn näytemäärän kirjoittamiseen levylle. Testaus suoritetiin epäsuoralla tavalla, sillä suora mittaustapa vaatisi kaksisuuntaisen viestintäkanavan varsinaisen informaatiokanavan rinnalle testin koordinointiin. Kanavaa pitkin osapuolet varmistuisivat yhden päivityssyklin saapumisesta perille ennen seuraavan lähettämistä. Tämä saattaisi vääristää jonkun verran tuloksia, ja olisi ennen kaikkea epärealistisempi kuin valittu mittaustapa. Todellisessa järjestelmässä väylälle kirjoittajat eivät voi tietää

mitään konkreettista lukijoiden tai väylän kapasiteetista, vaan kirjoittavat väylälle päivityksiä säännöllisin aikavälein mahdollisesta väylän tukkiutumisesta ja tietohukasta huolimatta.

Historiapalvelun tallennuskapasiteettia testattiin lähettämällä järjestelmään purskeittain päivityksiä useasta samanaikaisesta kohteesta. Kohteiden määrää ja purskeiden lähettämisen aikaväliä vaihdeltiin testitapauksen mukaan. Jokaista testiä ajettiin kunnes 50 päivityspursketta oli lähetetty ja vastaanotettu. Testit suoritettiin myös tallennus pois päältä kytkettynä, jotta voitaisiin paremmin vertailla tuloksia muiden testitapausten kanssa.

Testeissä selvitettiin pienin purskeiden lähettämisen aikaväli, jolla yli 95% päivityksistä saadaan tallennettua. Tämä 5% häviikki sallitaan tässä tapauksessa, sillä toteutetun tallennuspalvelun tiedonkäsittely ei ole yksisäikeisyytensä vuoksi tallennuksen kannalta optimaalinen. Myös mittaustuloksia tutkiessa havaittiin, että 100% tallennuksen vaatiminen olisi vääristänyt testin tuloksia merkittävästi huonontaen tuloksia erityisesti pienillä näytemäärillä.

Syy epäoptimaalisuuteen on se, että tallennuspalvelun rinnakkaistumaton toteutus yhdistettynä kirjoituksen puskuroimiseen aiheuttaa hetkittäin suurempia viiveitä tallennuksessa. Näiden merkitys korostuu erityisesti lyhyillä päivitysväleillä, kun hetkellisen viiveen aikana tuottaja ehtii siirtyä jo seuraavan päivityserän lähettämiseen. Tästä ongelmasta päästäisiin varmasti toteuttamalla tallennuspalvelun kirjoitusosuus omana säikeenään.



Kuva 5.6: Historiapalvelun suorituskyky tallennettaessa

Testien tuloksia nähdään kuvassa 5.6. Kuvassa X-akselilla on päivitettävien kohteiden määrä ja Y-akselilla päivitysväli. Käyrille on merkitty lyhin päivitysväli, jolla vaatimusten mukaisesti vähintään 95% lähetetyistä päivityksistä saadaan tallennettua. Ylempi käyrä kuvaa tallennuspalvelun käyttäytymistä tallennuksen ollessa normaalisti päällekytkettynä. Alempi käyrä kuvaa palvelun käyttäytymistä tallennuksen ollessa kytkettynä pois päältä.

Käyristä nähdään käyttäytymisen olevan melko lähellä lineaarista käyttäytymistä kohteiden määrän suhteen. Tämä oli odotettavissa oleva tulos, sillä kohteiden määrä lisää lineaarisesti vastaanotto- ja tallennusoperaatioiden määrää. Alemmalla käyrällä suoritetaan ainoastaan toinen näistä operaatioista, vastaanotto ilman tallennusta, joten sen kulmakerroin on hieman matalampi. Käyrien välinen erotus syntyy tallennuksessa tehtävistä suhteellisen raskaista operaatioista: olioiden sarjallistamisesta ja tietovirran pakkaamisesta. Historiataltion tallennuksen suorituskyvyn testauksesta saatujen tulosten perusteella voidaan sanoa tämän eron johtuvan suurimmaksi osaksi sarjallistamisesta.

5.2 Tulosten arviointi

Historiapalvelun suorituskyky vaikuttaisi kokonaisuudessaan olevan jo tällaisenaan varsin hyvällä tasolla. Historiapalvelun tallennuksen suorituskyvyn testauksen perusteella (kuva 5.6) nähdään että tallennuksen päälle kytkeminen pidentää tuhannen näytteen arvioidun vastaanottoajan noin kolminkertaiseksi. Tämä on huomattava hidastuminen, mutta suorituskyky on kuitenkin riittävä prototyyppivaiheessa olevalle järjestelmälle.

Ensimmäisessä testissä testattiin taltion pakkauksen vaikutusta taltion kirjoituksen nopeuteen. Vaikutus todettiin kuitenkin suhteellisen pieneksi, joten ilmeisesti sarjallistaminen on merkittävin tekijä taltion kirjoituksen nopeudessa ja siten luonnollinen kehityskohde taltion kirjoituksen nopeuttamiselle.

Kahden ensimmäisen testin perusteella voidaan arvioida että tallennuksessa kuluu yhtäläinen aika taltion kirjoittamiseen ja tiedon vastaanottamiseen silloin kun kohteiden määrä on noin viisisataa. Historiapalvelun tallennuksen suoriuskyvyn testauksen tulokset puolestaan osoittavat tämän rajan olevan jossakin hieman tuhannen kohteen yläpuolella. Näytteiden määrän ollessa tätä korkeampi tiedon vastaanottamiseen kuluva aika hallitsee testitulosta, ja näytteiden määrän ollessa pienempi taltion kirjoittaminen hallitsee testitulosta. Tämä vaikuttaa johtuvan siitä, että DDS-alustassa tiedon vastaanottaminen hidastuu näytteiden määrän kasvaessa.

6. YHTEENVETO JA JOHTOPÄÄTÖKSET

Tässä luvussa käydään läpi työn merkittävimmät vaiheet, sen tuomat vaikutukset järjestelmän kokonaisarkkitehtuuriin, hylätyt toteutusvaihtoehdot ja jatkokehitysajatukset.

6.1 Prototyypin tarkastelu

Työssä toteutettiin prototyyppi historiapalvelusta, joka kykenee tallentamaan ja toistamaan historiatietoa osana hajautettua tilannekuvajärjestelmää. Lisäksi muokattiin olemassaolevaa käyttöliittymää niin että se soveltuu historiantoiston ohjaukseen, ja toteutettiin uusi versio yrityksessä aiemmin toteutetusta DDS:n käyttöä yksinkertaistavasta apukirjastosta.

Prototyypille asetetuista vaatimuksista tärkeimpiä olivat vaatimus mukautettavuudesta minkä tahansa tietotyypin käsittelemiseen ja vaatimus siitä, että tallennuspalvelun käyttöönotto ei saa aiheuttaa muutoksia muihin järjestelmän komponentteihin. Nämä toteutuivat uuden DDS-apukirjaston myötä erittäin hyvin: historiapalvelu pystyy käsittelemään mitä tahansa tietotyyppiä, johon liittyvät luokkatiedostot sillä on saatavillaan. Muihin komponentteihin tarvittavat muutokset rajoittuvat käyttöliittymän kokoonpanon muuttamiseen historiatiedon käyttämiseksi, tietolähteitä ei tarvitse muokata lainkaan.

Suorituskyky jäi välttävälle, mutta prototyyppijärjestelmän ollessa kyseessä kuitenkin varsin riittävälle tasolle. Merkittävin tekijä historiapalvelun suorituskyvylle on sarjallistamisen raskaus. Suuremmilla kohdemäärillä kuitenkin myös DDS-alustan suorituskyky näyttää kasvavan merkitykselliseksi tekijäksi. Historiapalvelun prototyyppitoteutus kuitenkin vihjaa että tällainen historiapalvelu on mahdollista toteuttaa riittävän suorituskykyisenä kohtuullisella panostuksella.

6.2 Vaikutukset kokonaisarkkitehtuuriin

Osana työtä toteutettu reflektiivinen DDS-apukirjasto osoittautui suorituskyvyltään käytännössä yhtä nopeaksi kun aikaisempi ei-reflektiivinen versio. Tästä suorituskykyisyydestä ja reflektiivisen version helpommasta käytöstä ja joustavuudesta johtuen apukirjasto otettiin jo prototyypin toteutuksen aikana käyttöön järjestelmään liittyvässä testaustyökalussa. Pian tämän jälkeen myös järjestelmän muut osat siirtyivät käyttämään toteutettua reflektiivistä versiota apukirjastosta.

On kuitenkin syytä muistaa että reflektion käytön joustavuudella on hintansa kääntäjän tekemien tarkastusten puuttumisessa. Tästä johtuen DDS-apukirjastoa käyttävien luokkien testauksessa on syytä kiinnittää erityistä huomiota testien kattavuuteen ja niiden säännölliseen ajamiseen. Monet virheet jotka aiemmin olisi huomattu heti käännösvaiheessa, jäävät reflektion myötä myöhempien vaiheiden huomattavaksi.

6.3 Hylätyt toteutusvaihtoehdot

Toteutusvaihtoehtoja joita pohdittiin, mutta päädyttiin lopulta hylkäämään oli muutamia. Näistä merkittävimpiä olivat DDS:n verkkoprotokollan purkaminen ja käyttäminen tallennuksen lähteenä DDS:n oman rajapinnan sijasta, DDS-apukirjaston toteuttaminen ilman reflektiota pelkällä Generics-mekanismilla ja Javan oman sarjallistamismekanismin käyttäminen.

Näistä DDS:n verkkoprotokollan purkaminen hylättiin tällä erää potentiaalisesti erittäin suuritöisenä, epävarmana ja eri järjestelmien väliseltä yhteensopivuudeltaan epäilyttävänä. Apukirjaston toteuttaminen pelkällä Generics-mekanismilla hylättiin kun työn alkuvaiheessa selvisi että se ei riittäisi täyttämään vaatimusta minkä tahansa tietotyyppien tuen lisäämisestä palveluun jälkikäteen. Javan sarjallistamismekanismin käyttäminen hylättiin, sillä se olisi vaatinut DDS:n IDL-kääntäjän muokkaamista jotta sen tuottamat luokat olisi saatu toteuttamaan java.io.Serializable -rajapinta. Ilman tämän rajapinnan toteuttamista ei Javan sarjallistamismekanismi kykene näitä olioita sarjallistamaan.

6.4 Jatkokehitysajatukset

Jatkokehitysajatuksia järjestelmän parantamiseksi jäi useita. Vaikka DDS:n verkkoprotokollan tallentaminen suoraan hylättiinkin tällä kertaa, se voisi silti olla mielenkiintoinen tutkimuskohde. Toinen vastaavaan tulokseen pyrkivä kehitysmahdollisuus olisi DDS:n rajapinnan ohittaminen, eli toimittajan DDS-toteutuksen sisäisten luokkien käyttäminen suoraan DDS-rajapinnan ohi. Näillä voitaisiin mahdollisesti päästä eroon siitä, että historiapalvelulla täytyy olla saatavilla tallennettavien ja toistettavien tietotyyppien IDL-kuvauksista käännetyt apuluokat.

Haluttaessa aiheiden kuuntelun voisi automatisoida niin, että historiapalvelu asettuu automaattisesti kuuntelemaan kaikkia ymmärtämiään aiheita väylällä. Tiedon tallentamisvaiheeseen voitaisiin toteuttaa suodatus, ja näin karsia turhaa tietoa jo tallennusvaiheessa. Reflektiota käyttäen voitaisiin saapuvia viestejä suodattaa jonkun kentän arvon perusteella. Lisäksi tallennettavasta tiedosta voitaisiin tallentaa tyypin nimen ja saapumisaikaleiman lisäksi myös muuta tietoa. Näiden toteuttaminen yleispätevällä tavalla voi kuitenkin olla melkoisen haastavaa.

Suorituskyvyn kannalta selkein kehityskohde on sarjallistamisen tehostaminen. Nykyisellään siihen kuluu merkittävin osa tallennuspalvelun suoritinkäytöstä. Uusien tallennusmuotojen tutkiminen voisi tuoda parannusta tähän. Sarjallistamisen kehittämisen yhteydessä voitaisiin myös tutkia tallenteiden yhdistämistä. Toinen suorituskyvyn parantamiseen pystyvä kehityskohde olisi rinnakkaisuuden suurempi hyödyntäminen, sillä nykyisellään järjestelmän hitaimmat osiot eivät rinnakkaistu lainkaan. Historiapalvelua toteutettaessa tuli myös ilmi vihjeitä siitä, että DDS-väylältä lukemisen muuttaminen kuuntelija-mekanismista säännöllisin aikavälein lukemiseen saattaisi tarjota parempaa suorituskykyä.

Tallennusmuodossa olisi myös selvästi kehittämisen varaa: nykyisellään tiedostokoko täytyy pitää melko pienenä, koska muuten toistopalvelun tiedoston pikakelaamiseen kuluva aika toistoa aloitettaessa kasvaa sietämättömän suureksi. Tallennusmuodon olisi syytä tukea paremmin toiston aloittamista mielivaltaisesta paikasta.

Käyttöliittymää parantamalla voitaisiin myös parantaa käyttökokemusta merkittävästi. Kehityksen aikana tapahtuneessa manuaalisessa testailussa erittäin tarpeelliseksi lisäykseksi tulevaisuudessa havaittiin kello, joka kertoisi missä ajanhetkessä historian toisto sillä hetkellä kulkee.

Kaikkiaan mahdollisuuksia jatkokehitykselle ja lisätutkimuksille jäi mukavasti. Kun suorituskykyä saadaan kehitettyä parempaan suuntaan, toiston alkamiseen liittyvät haasteet ratkaistua sekä käyttökokemusta parannettua, voidaan todella sanoa järjestelmän toteuttavan kaikki asetetut tavoitteensa.

LÄHTEET

[1] Kari, M., Hakala, A., Pääkkönen, E., Pitkänen, M. *Puolustusjärjestelmien kehitys - Sotatekninen arvio ja ennuste 2025, osa 2*. Helsinki 2008, Edita Prima Oy. 280 s.

[2] Coulouris, G., Dollimore, J., Kindberg, T. *Distributed systems - concepts and design*. 4th edition. England 2005, Addison-Wesley Publishers; Pearson Education. 944 s.

[3] About SETI@home [WWW]. [viitattu 29.12.2010]. Saatavissa: http://setiathome.berkeley.edu/sah_about.php

[4] Folding@home - Main [WWW]. [viitattu 29.12.2010]. Saatavissa: http://folding.stanford.edu/English/Main

[5] Folding@home - Client statistics by OS [WWW]. [viitattu 15.12.2010]. Saatavissa: http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats

[6] Ajankohtaista: Pohjois-Karjalan rajavartiosto tutkii mahdollista alueloukkausta [WWW]. Rajavartiolaitos, 21.11.2008. [viitattu 25.12.2010]. Saatavissa: http://www.raja.fi/

[7] Ilmavoimat - Hävittäjälentolaivue 31 [WWW]. [viitattu 1.2.2010]. Saatavissa: http://www.ilmavoimat.fi/index.php?id=920

[8] Kuusisto, O. *Viestintäominaisuuksien kehittäminen johtamisjärjestelmässä*. Diplomityö. Tampere 2008. Tampereen teknillinen yliopisto. 72 s.

[9] OMG specification formal/2007-01-01. *Data Distribution Service for Real-time Systems*, v1.2. 2007, Object Management Group. 260 s. Saatavissa: http://www.omg.org/.

[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J.. *Design Patterns: Elements of Reusable Object Oriented Software*. Massachusetts 1994, Addison-Wesley Professional. 416 s.

[11] Data Distribution Intro [WWW]. [viitattu 16.12.2010]. Saatavissa: http://www.omgwiki.org/dds/content/data-distribution-intro.

[12] OMG specification formal/2008-01-04. *Common Object Request Broker Architecture (CORBA) Specification*, Version 3.1. 2008, Object Management Group. 540 s. Saatavissa: http://www.omg.org/

[13] Salmivesi, T. *Tiedon hajautus johtamisjärjestelmässä ja DDS-teknologia*. Diplomityö. Lappeenranta 2010. Lappeenrannan teknillinen yliopisto. 61 s.

[14] Joshi, R. *A Comparison and Mapping of Data Distribution Service (DDS) and Java Message Service (JMS)*. 2006, Real-Time Innovations. 44 s.

[15] Peltola, T. *Reaaliaikavälikerroksen arviointi johtamisjärjestelmäkäyttöön*. Diplomityö. Tampere 2008. Tampereen teknillinen yliopisto. 57 s.

[16] G. Bracha. *Generics in the Java Programming Language* [WWW]. [viitattu 25.12.2010]. Saatavissa: http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

[17] The Reflection API (The Java^{TM} Tutorials) [WWW]. Oracle. [viitattu 25.12.2010]. Saatavissa: http://java.sun.com/docs/books/tutorial/reflect/index.html

[18] Kamppinen, T. *Palvelupohjainen taktisen grafiikan piirtotyökalu johtamisjärjestelmäympäristössä*. Diplomityö. Tampere 2009. Tampereen teknillinen yliopisto. 54 s.

[19] Spring Documentation [WWW]. [viitattu 25.12.2010]. Saatavissa: http://www.springsource.org/documentation/

[20] JIDE Software - Products [WWW]. [viitattu 25.12.2010]. Saatavissa: http://www.jidesoft.com/products/

[21] LuciadMap v9.0.19 - Developer's Guide. Luciad NV, 2009.

[22] dds-psm-cxx - Project Hosting on Google Code [WWW]. [viitattu 20.11.2010]. Saatavissa: http://code.google.com/p/dds-psm-cxx/

[23] simd-cxx - Project Hosting on Google Code [WWW]. [viitattu 20.11.2010]. Saatavissa: http://code.google.com/p/simd-cxx/

[24] datadistrib4j - Project Hosting on Google Code [WWW]. [viitattu 20.11.2010]. Saatavissa: http://code.google.com/p/datadistrib4j/

[25] Qusay H. Mahmoud. *Compressing and Decompressing Data Using Java* [WWW]. Oracle 2002. [viitattu 16.12.2010]. Saatavissa: http://java.sun.com/developer/technicalArticles/Programming/compression/

[26] Hello World Application | Data Distribution Service Portal [WWW]. [viitattu 25.12.2010]. Saatavissa: http://portals.omg.org/dds/content/hello-world-application

LIITE 1: DATA DISTRIBUTION SERVICE -ESIMERKKIKOODIT

Hello World

Lainattu pienin muokkauksin lähteestä [26].

Lähettäjä:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h> // needed for sleep()
4  #include <dds_cpp.h>
5  using DDS;
6  int main(int argc, char *argv[])
7  {
8      int domainId = 0;
9      int sample_count = 100;
10     DomainParticipant *participant = NULL;
11     Publisher *publisher = NULL;
12     Topic *topic = NULL;
13     DataWriter *writer = NULL;
14     HelloWorldDataWriter *helloWriter = NULL;
15     HelloWorld instance;
16     InstanceHandle_t instance_handle = HANDLE_NIL;
17     const char *type_name = NULL;
18     int count = 0;
19     participant = TheParticipantFactory->create_participant(
20         domainId, PARTICIPANT_QOS_DEFAULT, NULL /* listener */,
21         STATUS_MASK_NONE);
22     publisher = participant->create_publisher(
23         PUBLISHER_QOS_DEFAULT, NULL /* listener */,
24         STATUS_MASK_NONE);
25     type_name = HelloWorldTypeSupport::get_type_name();
26     HelloWorldTypeSupport::register_type(
27         participant, type_name);
28     topic = participant->create_topic(
29         "Example HelloWorld",
30         type_name, TOPIC_QOS_DEFAULT, NULL /* listener */,
31         STATUS_MASK_NONE);
32     writer = publisher->create_datawriter(
33         topic, DATAWRITER_QOS_DEFAULT, NULL /* listener */,
34         STATUS_MASK_NONE);
35     helloWriter = HelloWorldDataWriter::narrow(writer);
36     strcpy(instance.name, "MyName");
37     instance_handle = HelloWorld_writer->register_instance(instance);
38     /* Main loop */
```

```
39      for (count=0; count < sample_count; ++count) {
40          sprintf(instance.msg, "Hello World! (count %d)", count);
41          printf("Writing: %s", instance.msg);
42          helloWriter->write(*instance, instance_handle);
43          sleep(1);
44      }
45      participant->delete_contained_entities();
46      TheParticipantFactory->delete_participant(participant);
47  }
```

Vastaanottaja:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h> // needed for sleep()
4  #include <dds_cpp.h>
5  using DDS;
6  /* Listener used to receive notifications on data updates */
7  class HelloWorldListener : public DataReaderListener {
8    public:
9      virtual void on_data_available(DataReader* reader);
10 };
11 void HelloWorldListener::on_data_available(DataReader* reader)
12 {
13     HelloWorldDataReader *HelloWorld_reader = NULL;
14     HelloWorldSeq data_seq;
15     SampleInfoSeq info_seq;
16     HelloWorld_reader = HelloWorldDataReader::narrow(reader);
17     retcode = HelloWorld_reader->take(
18         data_seq, info_seq, LENGTH_UNLIMITED,
19         ANY_SAMPLE_STATE, ANY_VIEW_STATE, ANY_INSTANCE_STATE);
20     for (i = 0; i < data_seq.length(); ++i) {
21         if (info_seq[i].valid_data) {
22             printf("From %s: %s\n", data_seq[i].name, data_seq[i].msg);
23         }
24     }
25     retcode = HelloWorld_reader->return_loan(data_seq, info_seq);
26 }
27 int main(int argc, char *argv[])
28 {
29     int domainId = 0;
30     DomainParticipant *participant = NULL;
31     Subscriber *subscriber = NULL;
32     Topic *topic = NULL;
33     HelloWorldListener *reader_listener = NULL;
34     DataReader *reader = NULL;
35     ReturnCode_t retcode;
```

```
36      const char *type_name = NULL;
37      participant = TheParticipantFactory->create_participant(
38          domainId, participant_qos,
39          NULL /* listener */, STATUS_MASK_NONE);
40      subscriber = participant->create_subscriber(
41          SUBSCRIBER_QOS_DEFAULT, NULL /* listener */, STATUS_MASK_NONE);
42      type_name = HelloWorldTypeSupport::get_type_name();
43      retcode = HelloWorldTypeSupport::register_type(
44          participant, type_name);
45      topic = participant->create_topic(
46          "Example HelloWorld",
47          type_name, TOPIC_QOS_DEFAULT, NULL /* listener */,
48          STATUS_MASK_NONE);
49      /* Create data reader listener */
50      reader_listener = new HelloWorldListener();
51      reader = subscriber->create_datareader(
52          topic, DATAREADER_QOS_DEFAULT, reader_listener,
53          STATUS_MASK_ALL);
54      /* Main loop. Does nothing. Action taken in listener */
55      for (count=0; count < sample_count; ++count) {
56          printf("HelloWorld subscriber sleeping for %d sec...\n",
57                  receive_period.sec);
58          sleep(10);
59      }
60      participant->delete_contained_entities();
61      TheParticipantFactory->delete_participant(participant);
62  }
```