TAMPEREEN TEKNILLINEN YLIOPISTO

SAMU NUUTAMO

ASKELTAVA SUORITUS HAJAUTETUSSA YMPÄRISTÖSSÄ

Diplomityö

Tarkastaja: prof. Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
Tietotekniikan osastoneuvoston
kokouksessa 9.9.2009

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
NUUTAMO, SAMU: Askeltava suoritus hajautetussa ympäristössä
Diplomityö, 49 sivua, 8 liitesivua
Huhtikuu 2010
Pääaine: Ohjelmistotuotanto
Tarkastajat: professori Hannu-Matti Järvinen
Avainsanat: askeltaja, askeltava suoritus, sekvensseri, sekvenssi, ohjelmistoarkkitehtuuri, rinnakkaisuus, hajautettu järjestelmä, hallinta, valvonta, ohjelmistokomponentti, suorituspolku

Viestinvälitykseen perustuvissa järjestelmissä on usein haasteena kehittää mekanismit proseduraalisten suorituspolkujen luomiseksi. Viestinvälityksen asynkronisuus ei suoraan salli tällaista, vaan tarvitaan erillinen komponentti, joka ottaa vastuulleen suorituksesta huolehtimisen. Askeltajakomponenttiin on luonnollista sisällyttää myös muuta toiminnallisuutta, jolla voidaan helpottaa koko järjestelmän tilan tarkkailua.

Tässä diplomityössä toteutetaan askeltajakomponentti olemassa olevaan tuoterunkoon. Askeltaja toimii tuoterunkoon liitettynä keskeisessä roolissa koko järjestelmässä. Sen vastuulla on erinäisten sekvenssien suorittaminen, suoritustilan tarkkailu ja kirjanpito. Viestiväylään perustuvassa asynkronisessa ohjelmistossa askeltaja tuo järjestelmään tietynlaista deterministisyyttä, koska suoritusketjut ovat toistettavissa ja jäljitettävissä. Lisäksi askeltaja mahdollistaa suorituksen tarkan hallinnan, jopa tarkemman kuin mihin proseduraalisissa ohjelmistoissa on totuttu.

Askeltajan olemassaolo vaikuttaa väistämättä järjestelmän arkkitehtuuriin. Perinteisissä sovelluksissa pyritään luomaan erilaisia kerroksia, jolloin ohjelmiston eri osien riippuvuudet ovat selkeitä suunnittelijoille. Diplomityön kuvaamassa tuoterungossa viestiväylän takia järjestelmään liittyneet komponentit ovat kuitenkin vertaisiaan, jolloin askeltajan olemassaolo toimii kerrosarkkitehtuurin tavoitetta vastaan. Tämä hankaloittaa komponenttien suhteiden selvittämistä, mutta toisaalta se myös sallii erittäin joustavia toteutusratkaisuja.

Hajautettu järjestelmä tuo myös muita haasteita komponentin toteutukselle. Esimerkiksi verkkoviiveet ja mahdolliset muutokset verkon topologiassa eivät saa jättää järjestelmää virheelliseen tilaan. Perinteisten, hajauttamattomien sovellusten tapauksessa tietyt sekvenssit ovat aina suoritettavissa. Hajautetussa järjestelmässä tämä ei kuitenkaan ole taattua.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Computer Technology
NUUTAMO, SAMU: Stepping execution in a distributed environment
Master of Science Thesis, 49 pages, 8 appendix pages
April 2010
Major: Software engineering
Examiner: professor Hannu-Matti Järvinen
Keywords: stepping execution, sequencer, sequence, software architecture, multi-threading, distributed system, managing, monitoring, software component, execution path

In systems based on a message bus the challenge usually is to develop mechanisms to create procedural execution paths. The asynchronous message dispatching does not allow this and an external component is needed to take responsibility of managing the execution. It is natural to also include other functionality that can ease the monitoring of the system state.

In this thesis a sequencer component is implemented to be part of an existing framework. Attached to the framework the sequencer works in a essential role. Its responsibilities are the execution of the sequences, the monitoring of the system state and logging. In an asynchronous system based on a message bus the sequencer delivers the promise of determinism, because every sequence can be traced and repeated. Furthermore, the sequencer enables the execution to be managed — even in greater detail than people are used to in procedural programs.

The presence of the sequencer inevitably affects the architecture of the system. In traditional programs it is customary to create different layers within the software so that dependencies inside the software are clear to the designers. In the framework described in this thesis all the components attached to the system are peers and therefore the presence of the sequencer breaks the layering principle. This complicates the understanding of the component relations but on the other hand allows very flexible solutions.

Distributed system brings other challenges to the implementation as well. For example network delays and possible changes in network topology must not leave the system in an erroneous state. With traditional, undistributed systems some sequences can always be executed. In a distributed system this is not guaranteed.

ALKUSANAT

Tämä diplomityö on tehty osana Patria Oyj:n tuotekehitysprojektia keväiden 2009 ja 2010 välisenä aikana. Vaikka diplomityön tarkoituksena on testata kirjoittajan kykyä itsenäiseen työskentelyyn, opin kirjoittamisen aikana myös arvostamaan tiimityöskentelyä sekä ideoista keskustelua.

Haluan esittää kiitokseni työn ohjaajana toimineelle Tero Kaipiolle hänen antamistaan hyvistä neuvoista, sekä työkavereilleni kannustuksesta ja työni oikoluvusta. Haluan kiittää myös työn tarkastajaa, professori Hannu-Matti Järvistä arvokkaista kommenteista ja ongelmakohtien esille tuomisesta.

Tampereella 9.3.2010

Samu Nuutamo

SISÄLLYS

LYHENTEET JA MERKINNÄT

ACE (Adaptive Communication Environment) Avoimen lähdekoodin ohjelmistokehys siirrettävien ohjelmistojen tuottamiseksi.

CP (Checkpoint) Piste, jossa jokin kokonaisuus on saatu suoritettua.

CORBA (Common Object Requesting Broker Architecture) Standardoitu ohjelmistoarkkitehtuurimalli, joka mahdollistaa eri ohjelmointikielillä toteutettujen komponenttien yhteistoiminnan samassa järjestelmässä.

FDR (Flight Data Recorder) Lentokoneen musta laatikko, joka tallentaa tietoa erinäisten järjestelmien toiminnasta.

SCS Sequencer Controlled Software

HW (Hardware) Laitteisto.

Keskeytysvektori Taulukko osoittimia keskeytyskäsittelijöihin. Joissain järjestelmissä taulukko voi sisältää hyppykäskyjä suorien osoittimien sijaan.

Mesh-verkko Vahvasti kytkeytynyt verkko, jossa solmut ovat liittyneet suoraan toisiinsa.

Sekvenssi Useista erillisistä tehtävistä koostuva suoritusketju.

TAO (The ACE ORB) ACE-tuoterungon CORBA-toteutus.

Tehtävä (*task*) Ohjelman toimintokokonaisuus, joka tähtää jonkin tietyn suorituspäämäärän savuttamiseen. Sisältää yhden tai useamman perustoiminnon.

Tehtävälista Ryhmä samaan sekvenssiin kuuluvia tehtäviä, jotka suoritetaan järjestyksessä.

Toiminto (*operation*) Pienin yksittäinen askeltajan hallitsema toimintokokonaisuus, monitoroinnin ja hallinnan erottelukyky (resolution).

UML (Unified Modeling Language) Graafinen mallinnuskieli, jota käytetään järjestelmien ja ohjelmistojen kehityksessä.

WDT (Watchdog Timer) Valvonta-ajastin. Ohjelmiston osa, joka tarkkailee suorituksen edistymistä ja huomaa lukkiutumiset.

XML (eXtensible Markup Language) Rakenteinen kuvauskieli puumaisen tiedon esittämiseen.

1. JOHDANTO

Monimutkaisten ohjelmistojen tapauksessa tulee usein tarve päästä valvomaan ohjelman suorituksen tilaa. Lisäksi saatetaan tarvita tietoa ohjelman suorituksesta, esimerkiksi siitä, mitä ohjelma aikoo tehdä seuraavaksi. Tällaisen toiminnallisuuden toteuttaminen vaikuttaa väistämättä ohjelmiston arkkitehtuuriin, ja se onkin vaikeaa lisätä ohjelmistoon jälkikäteen.

Ohjelmistoja on perinteisesti suunniteltu käyttäen apuna ohjelmistotuotannon v-mallia, joka sopii alkuperäiseen tarkoitukseensa loistavasti. V-malli itsessään on erittäin joustava kuvausmekanismi, jonka käyttöä ohjelmistojen kuvauksessa tämä diplomityö pyrkii laajentamaan. V-mallilla ei tässä työssä kuitenkaan viitata perinteiseen ohjelmistokehityksen v-malliin, vaan sen avulla kuvataan ohjelman suorituspolkua. Suorituspolkujen määrittely toimii osana ohjelmiston kehitystä ja ohjelmiston ylläpidon aikana lukija saa niiden avulla selkeän käsityksen siitä, mitä ohjelmisto tekee missäkin suoritusvaiheessa.

Järjestelmän suoritusta voidaan hallita käyttäen askeltajaa, joka hyödyntää v-mallien avulla määriteltyjä sekvenssejä ohjelman suorituspolun valinnassa. Yksinkertaistettuna voidaan ajatella, että askeltaja pitää kirjaa ohjelman tilasta ja hallitsee siirtymiä eri suoritustilojen välillä. Tilat siirtymineen muodostavat sekvenssejä, joista muodostuu eräänlainen kartta järjestelmän kaikista mahdollista suorituspoluista. Sekvenssien avulla voidaan kuvata monimutkaisiakin järjestelmiä helposti ymmärrettävässä muodossa.

Sekvenssiajattelua monimutkaistaa kuitenkin huomattavasti niiden mahdollisuus sisältää suhteita toisiin sekvensseihin, jolloin jo pelkästään sekvenssien keskinäisten suhteiden hallinnasta tulee työlästä. Säikeistettyjen ohjelmistojen tapauksessa tarvitaan lisäksi mekanismeja suorituksen synkronointiin eri säikeiden välillä, jotta suoritus etenee loogisesti oikeassa järjestyksessä, ja vain sallittuja polkuja pitkin. Sekvenssien tiloihin voidaan lisäksi liittää lisämääreitä, jotka ohjaavat toimintaa tietyissä erikoistilanteissa.

Edellä kuvattujen haasteiden voittamiseksi askeltajan toteutukseen vaaditaan tietyt mekanismit ja tietorakenteet, joihin tässä dokumentissa etsitään ratkaisuja. Lisäksi tämän diplomityön tavoitteina on selvittää v-mallin käyttöä ohjelman suorituksen kuvaamisessa, dokumentoida ja kuvata askeltajan toteutus niin kattavasti, että sen toiminnan ymmärtäminen on mahdollista myös asiaan perehtymättömältä henkilöltä, sekä tutkia askeltajan hyödyntämisen vaikutusta ohjelmiston arkkitehtuuriin. Lopulta askeltajasta toteutetaan prototyyppiversio osaksi olemassa olevaa tuoterunkoa. Motivaationa työn taustalla on vanhan käytössä olevan järjestelmän joustamattomuus, sekä sen vähäinen dokumentaatio.

Tässä diplomityössä perehtytään ensin ohjelmistojen hallintaan (luku 3) ja moni-

torointiin (luku 4) käymällä läpi niiden periaatteita ja eri toteutusmekanismeja. Kun perusteet on käsitelty, siirrytään kuvaamaan askeltajan toimintaa (luku 5) sekä sen toiminnallisia vaatimuksia yleisellä tasolla. Tämän jälkeen kuvataan järjestelmä sen toteutuksen kannalta (luku 6) käytettävien luokkien tarkkuudella sekä toteutetaan järjestelmä. Lopuksi arvioidaan askeltajakomponentin vaikutuksia järjestelmään sekä esitetään toteutuskelpoisia jatkokehitysideoita (luku 7).

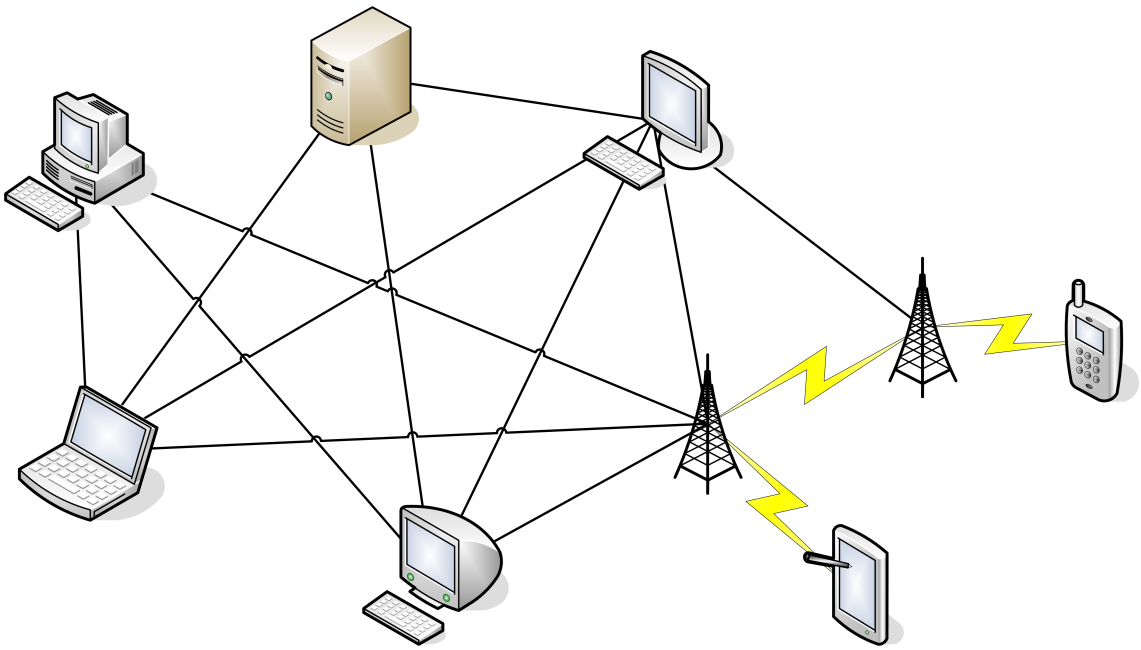
2. JÄRJESTELMÄN YLEISKUVAUS

Tässä luvussa kerrottu järjestelmän yleiskuvaus perustuu Timo Kuosmasen tekemään pro gradu -tutkielmaan [20] sekä Antti Viidanojan ja Lasse Ylisen diplomitöihin [12; 5]. Perustana olevaa järjestelmää ei pyritä kuvaamaan täydellisesti, vaan vain siltä osin, joka vaaditaan yleisen rakenteen ymmärtämiseksi. Kyseessä on laaja sovelluskehys, jonka syvällisempi tarkastelu tämän diplomityön puitteissa ei ole mahdollista.

2.1. Tuoterunko

Järjestelmän perustana on tuoterunko, josta voidaan erikoistaa vahvasti hajautettuja tilaaja–julkaisijamalliin perustuvia ohjelmistoja. Näiden ohjelmistojen avulla voidaan kerätä ja analysoida tietoa sekä suorittaa simulaatioita kerätyn datan perusteella. Analyysin tuloksista voidaan luoda yhteenvetoja käyttäjille. Järjestelmän päätarkoituksena on linkittyä erilaisiin antureihin ja laitteistoihin, joiden tuottamaa tietoa käsitellään edelleen käyttäjälle esittämistä varten. [20]

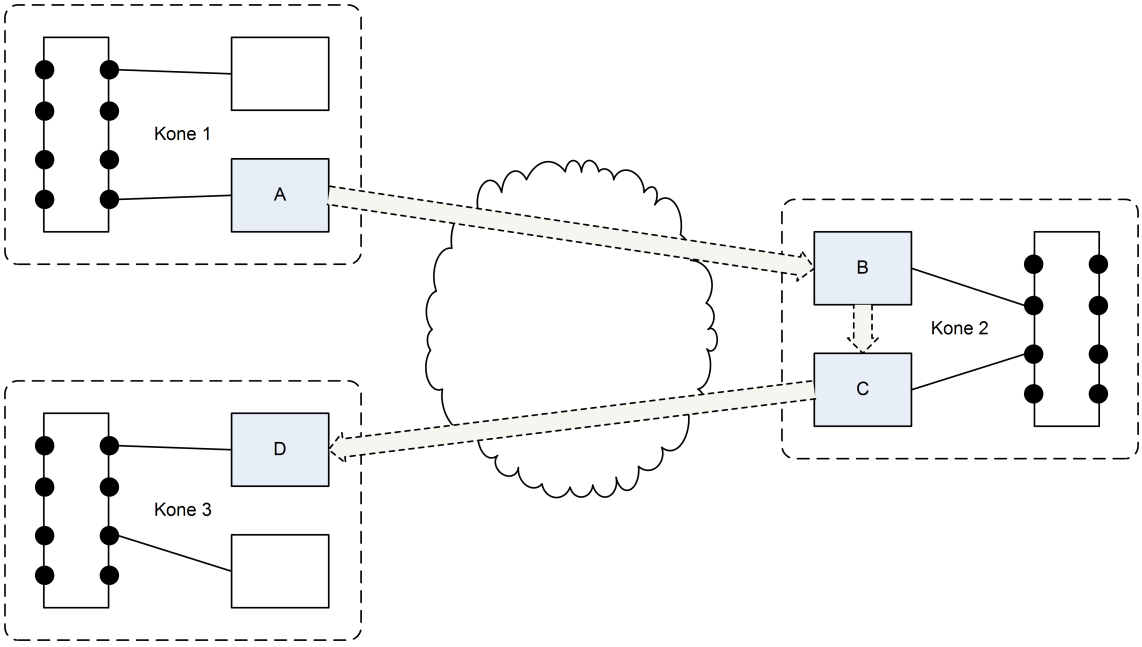
Kuva 2.1 esittää järjestelmän fyysistä rakennetta verkon näkökulmasta. Verkon solmut ovat järjestyneet mesh-verkon tavoin, eli niiden välillä voi olla suuri määrä yhteyksiä. Verkon fyysinen siirtotie ei rajoitu ethernet-tekniikkaan, vaan käytössä on lähiverkkotekniikan lisäksi muun muassa sarjaportti (RS-232) ja radiomodeemi.



Kuva 2.1: Esimerkki käytettävän verkon fyysisestä rakenteesta

Tuoterunko perustuu tietovuoarkkitehtuuriin [15, s. 132–136], jossa hajautetun järjestelmän eri solmut muodostavat välilleen tietovirtoja (kuva 2.2). Tietovirtojen

avulla järjestelmän eri osat vaihtavat tietoa keskenään. Kaikissa tapauksissa kyse ei kuitenkaan ole jatkuvasta virrasta, vaan myös yksittäisten viestien lähettäminen on mahdollista. Koska tietovirrat solmujen välillä liikkuvat viestiväylän päällä, noudattaa järjestelmä viestinvälitysarkkitehtuuria [15, s. 139–141]. Viestinvälitys mahdollistaa hajautetussa järjestelmässä kommunikaation verkon eri solmujen välillä käyttäen yksinkertaista rajapintaa.

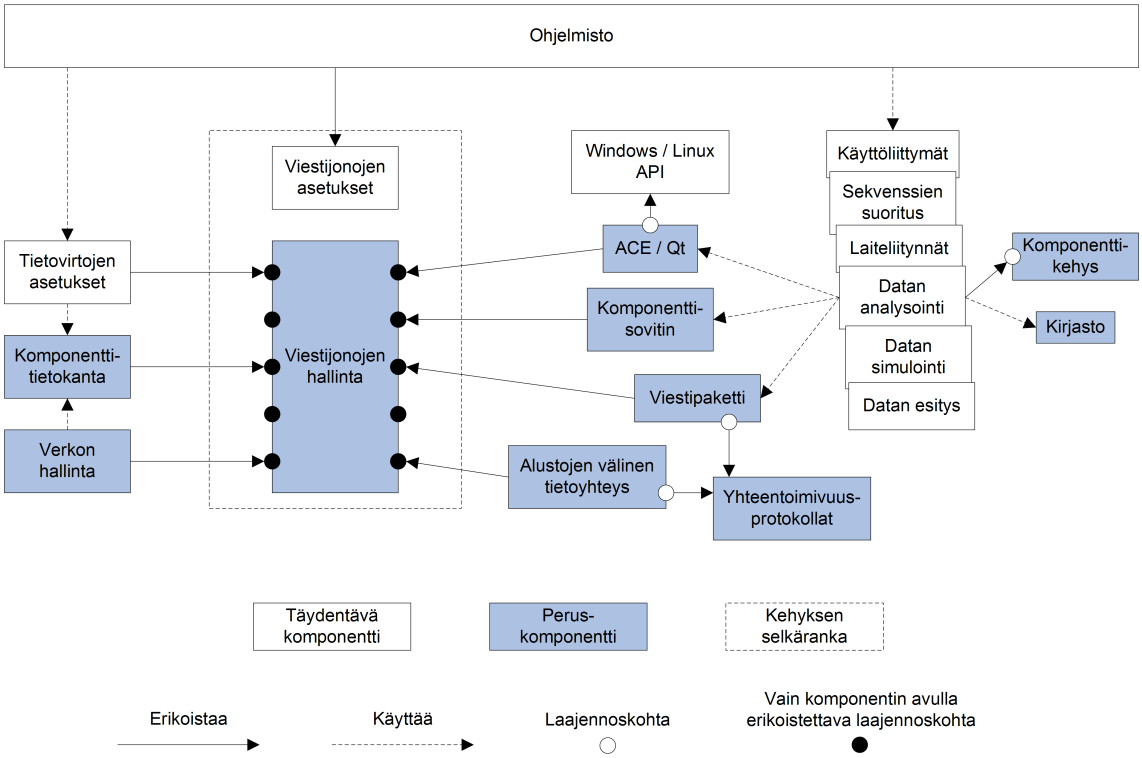


Kuva 2.2: Tietovuo järjestelmän komponenttien välillä [20, kuva 7.1]

Jotta järjestelmästä saataisiin mukautettava, toteuttaa se myös plugin–kehysmallin [15, s. 198–199]. Plugin–kehysmallin avulla tuoterunkoon voidaan liittää komponentteja, mikä mahdollistaa tuoterungon mukauttamisen helposti kohdesovelluksen tarpeita vastaavaksi. Käytännössä kaikki tuoterungon ulkopuolinen toiminnallisuus toteutetaan erilaisina komponentteina (kuva 2.3). Nämä kolme arkkitehtuuria yhdessä muodostavat tuoterungon selkärangan. Tässä diplomityössä käytetään tällaisesta järjestelmästä nimitystä *Sequencer Controlled Software* (SCS).

2.2. Ympäristö

Tuoterunko on suunniteltu käytettäväksi erilaisissa laitteisto- ja ohjelmistoympäristöissä. Sen ydin on toteutettu `C++`-ohjelmointikielellä käyttäen ilmaista, avoimen lähdekoodin The ADAPTIVE Communication Environment -ohjelmistokehystä (ACE) [1]. ACE sisältää tehokkaita oliopohjaisia luokkia, jotka kätkevät erilaiset käyttöjärjestelmien tarjoamat C-kieliset rajapinnat yhden yhtenäisen rajapinnan taakse. Tämä on tarpeellista heterogeenisen käyttöjärjestelmäympäristön yhtenäistämiseksi. Se tarjoaa muun muassa säikeistykseen, muistinhallintaan sekä verkko-



Kuva 2.3: Tuoterungon rakenne [20, kuva 7.2]

yhteyksiin liittyviä rajapintoja. ACE on toteutettu kymmenille eri alustoille, jolloin tuoterunko on helppo kääntää ajettavaksi myös jollekin täysin uudelle alustalle. Käyttöjärjestelmärajapintojen abstraktoinnin lisäksi ACE tarjoaa myös työkalut käännösvaiheeseen jokaiselle alustalle.

Tuoterungosta erikoistettuja sovelluksia tullaan suorittamaan ainakin Windows–, Linux– ja OS X –ympäristöissä. Myöskään muita käyttöjärjestelmäympäristöjä ei ole suljettu pois, joten ACE:n tarjoama joustavuus on tuoterungon kannalta välttämätöntä.

2.3. Komponentit

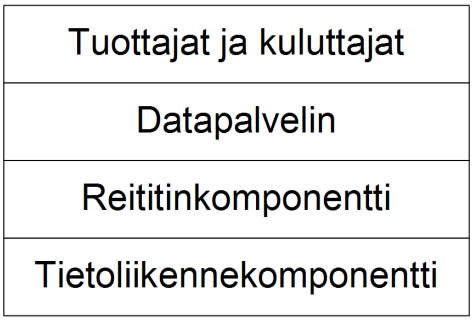
Tuoterungosta erikoistetaan ohjelmistoja luomalla komponentteja, jotka ovat dataa tuottavia tai käsitteleviä ohjelmistomoduuleja. Komponentit ovat toisistaan irrallisia ja ne ovat yhdistyneet toisiinsa viestinvälitysjärjestelmällä (kuva 2.2). Tämä mahdollistaa järjestelmän fyysisen hajauttamisen alla piilevästä verkosta riippumatta. Komponenttien sijainti ei ole rajattu yhden prosessin eri säikeisiin, vaan ne voivat myös olla eri prosesseissa tai jopa fyysisesti eri laitteissa. Käytännössä komponentit toteutetaan dynaamisesti ladattavina kirjastoina, joita voidaan liittää tuoterunkoon ajonaikaisesti.

Komponentit voivat tuottaa dataa, muokata sitä uudelleen tilattavaksi tai esit-

tää sitä käyttäjälle [12]. Tuoterunko itsessään on suunniteltu niin joustavaksi, ettei se keinotekoisesti pyri rajoittamaan sillä toteutettavia sovelluksia. Jokainen tuoterungosta erikoistettu sovellus kuitenkin sisältää tietyt peruskomponentit, jotka mahdollistavat esimerkiksi hajauttamisen. Komponenttien avulla voidaan tuoterungon päälle koostaa tarkasti loppukäyttäjän tarpeita vastaavia sovelluksia.

2.3.1. Tuottajat ja kuluttajat

Vaikka komponentit onkin toteutettu plugin–mallin mukaisina vertaismoduuleina, liikkuu tieto niiden välillä kerrosarkkitehtuurin mukaisesti. Komponentit voidaan siis ajatella pinona (kuva 2.4), joka kuvaa paremmin niiden välisiä suhteita. Pinon päällimmäisenä on tällöin jokin tietoa tuottava tai jalostava komponentti. Tuottajat ja kuluttajat ovat tilaaja–julkaisijamallin ilmentymiä. Näiden välillä esiintyy tietovirtoja, joilla välitetään tilauksia kuluttajille. Tietoa tuotetaan joko lukemalla jonkin anturin tuottamaa dataa tai jatkojalostamalla jonkin toisen komponentin tuottamaa dataa. Tuottaja voi halutessaan toimia myös tilaajana jollekin toiselle komponentille. Toisaalta komponentin ei tarvitse toimia tuottajana, vaan se voi pelkästään ottaa vastaan tietoa — esimerkiksi tulostaessaan sitä käyttäjälle.



Kuva 2.4: Järjestelmän komponentit esitettynä pinona

2.3.2. Datapalvelin

Datapalvelin on pinoajattelussa heti tuottajien ja kuluttajien alapuolella ja se toimii tilaajarekisterinä tilaaja–julkaisijamallille. Sen tehtävänä on jakaa tuottajien julkaisemaa tietoa kuluttajille, jotka ovat rekisteröityneet datapalvelimelle kyseisen tilauksen vastaanottajiksi. Rekisteröitymisen yhteydessä kuluttaja kertoo datapalvelimelle, minkä tyyppisestä datasta se on kiinnostunut, jonka jälkeen datapalvelin alkaa välittää tuotettua dataa kuluttajille. Myös tuottajat rekisteröityvät datapalvelimelle. [12; 5]

Koska tuottajat ja kuluttajat voivat sijaita fyysisesti eri koneilla, liittyy datapalvelimen toinen tärkeä toimenkuva verkkoyhteyksien hallintaan — se ylläpitää

verkkoyhteyksiä muodostamalla päästä–päähän-yhteyksiä eri verkkosolmujen välille. Datapalvelimen toiminnasta kerrotaan tarkemmin Lasse Ylisen diplomityössä [5].

2.3.3. Reititin

Koska datapalvelimen ja muiden komponenttien tietämys verkon fyysisestä rakenteesta rajoittuu kohdesolmujen tunnistelistaan, tarvitaan järjestelmään reititinkomponentti, joka pitää kirjaa verkon rakenteesta ja mahdollista poluista eri solmujen välillä. Reitittimen tehtävänä on valita paras reitti kohdesolmulle sekä tämän tiedon perusteella ohjata datapalvelimen lähettämät paketit oikeaan verkkoliitäntään. Reitin voi tehdä reitityspäätöksiä käyttäen apunaan myös palvelun laatuvaatimuksia sekä tietoa eri reittien metriikoista. Reitittimen toiminta kuvataan Antti Viidanojan diplomityössä [12].

2.3.4. Tietoliikennekomponentti

Alimmalla tasolla toimii tietoliikennekomponentti, joka hallinnoi yksittäisiä siirtoyhteystason yhteyksiä verkkoon. Riippuen alla piilevän fyysisen siirtoyhteyden tyypistä voi komponentti käyttää erilaisia moduuleita yhteyksien muodostamiseen. Esimerkiksi ethernet-verkossa voidaan käyttää TCP- tai UDP-moduulia. Tietoliikennekomponentin toiminta selvitetään tarkemmin Antti Viidanojan diplomityössä [12].

3. SUORITUKSEN HALLINTA

Askeltava suoritus tarkoittaa ohjelman hallintaa siten, että prosessin suoritus voidaan käskeä pysähtymään haluttuun aikaan halutussa pisteessä. Pysäytyksen jälkeen käyttäjällä on mahdollisuus päättää miten suoritusta jatketaan tai vaihtoehtoisesti käyttäjä voi tarkastella pysäytetyn prosessin tilaa. Askelluksella pyritään parantamaan ohjelman hallittavuutta sekä helpottamaan virheiden selvittämistä ja ohjelman suoritustilan tarkkailua.

3.1. Debuggerit

Esimerkkinä askeltavasta suorituksesta voidaan käyttää debuggereita, joiden avulla voidaan määritellä pysäyttämisen mahdollistavia pysäytyspisteitä (*breakpoint*). Debuggereita käytetään ohjelmavirheiden selvittämisessä joko ajonaikaisesti tai jälkikäteen niin sanottuja core-tiedostoja tarkastelemalla. Debuggereilla on pääsy ohjelman käyttämiin rekistereihin sekä pinoon, jolloin muuttujien arvot virheen sattuessa voidaan näyttää käyttäjälle. Korkeamman tason debuggereissa — esimerkiksi tulkattavien kielten yhteydessä — on rekisterien sisältö saatettu piilottaa käyttäjältä ja näkyvillä ovat vain muuttujien sisällöt.

Debuggerien pysäytyspisteiden toiminta perustuu IA-32-prosessoriarkkitehtuurilla siihen, että muistiin ladattuun ohjelmakoodiin sijoitetaan tilapäisesti `INT 3` (ansa, *trap to debugger*) -käsky paikkaan, jossa suoritus halutaan pysäyttää [9]. `INT 3` aiheuttaa ansan, joka ohjelman suorituksen pysäyttämisen lisäksi tuottaa POSIX-yhteensopivissa järjestelmissä signaalin `SIGTRAP`, jonka debuggeri kaappaa. Signaali on käyttöjärjestelmän mekanismi erilaisten tapahtumien ilmoittamiseksi prosesseille, jolloin debuggerin ei tarvitse käsitellä suoraan keskeytyksiä — eikä se olisi modernin käyttöjärjestelmän päällä toimivassa prosessissa edes mahdollista. Pysäyttämisen jälkeen korvattu käsky palautetaan takaisin paikoilleen, jolloin itse debuggerin toiminta on lähes täysin läpinäkyvää. Pysähtyneen ohjelman tilaa voidaan tutkia erittäin tarkasti — jopa rekisteritasolle asti — ja suoritusta voidaan myöhemmin jatkaa eteenpäin.

Debuggerit mahdollistavat ohjelman suorittamisen askeltaen — esimerkiksi yksi koodirivi kerrallaan eteenpäin tai suoraan seuraavaan pysäytyspisteeseen asti. Ne myös sallivat ohjelman tilan tutkimisen myös muiden, käyttäjästä riippumattomien signaalien tapauksessa. Esimerkiksi `SIGSEGV` (muistialueen ylitys) ja `SIGFPE` (aritmetiikkavirhe) ovat yleisiä signaaleja, joiden syyn selvittämisessä debuggereista on paljon hyötyä.

Askeltavan suorituksen perimmäisenä tarkoituksena on antaa käyttäjälle enemmän valtaa ohjelman suorituksen kontrolloinnissa. Perusideana on siis mahdollisuus

pysäyttää ohjelman suoritus tietyissä paikoissa, ja siten seurata ohjelman suoritusta. Lisäksi suoritusta tulee olla mahdollista jatkaa halutulla tavalla — joko ilman pysähdyksiä tai pysähtyen jokaisessa pysäytyspisteessä.

Pysäyttämisen ei kuitenkaan tarvitse koskea kokonaista prosessia, vaan pelkästään osaa sen säikeistä. Tämä eroaa debuggerien perinteisestä toimintaperiaatteesta, jossa koko prosessi pysäytetään kerralla — vaikkakin myös vain yhden säikeen pysäyttäviä debuggereita on kehitetty [14]. Monisäikeinen ympäristö on suuri haaste askellukselle, johon pelkästään matalan tason keinoin on hankala vastata. Tarvitaan siis korkeamman tason mekanismeja suorituksen hallintaan.

3.2. Menetelmiä suorituksen hallintaan

Tässä osiossa käydään läpi erilaisia menetelmiä suorituksen ohjaamiseen ja arvoidaan niiden soveltuvuutta ohjelman askellukseen.

3.2.1. Keskeytykset

Keskeytykset ovat matalimman tason mekanismi suorituksen ohjaamiseen. Niiden avulla voidaan pakottaa prosessori vaihtamaan suoritus käyttäjätilasta etuoikeutettuun tilaan ja samalla keskeyttämään käynnissä olevan ohjelman suoritus. Keskeytyksiä voi aiheutua laitteistosta tai niitä voidaan tuottaa ohjelmallisesti. Laitteistokeskeytyksillä ei voida kontrolloida ohjelman suoritusta sillä tarkkuudella kuin mitä askelluksessa tarvittaisiin, koska laitteisto ei tiedä tarkasti ohjelmakoodin tilaa. Toki on mahdollista asettaa ajastimia tuottamaan keskeytyksiä, mutta tämäkään ei ole sopiva mekanismi ohjelman askellukseen johtuen sen asynkronisuudesta.

Ohjelmistokeskeytyksiä voidaan tuottaa tarkalleen halutussa kohdassa ohjelman suoritusta (kohta 3.1). Nämä keskeytykset ”aktivoidaan” käyttämällä jotain suoritinarkkitehtuurille tyypillistä käskyä (esimerkiksi `INT`), joka siirtää suorituksen keskeytysvektorin kautta keskeytyskäsittelijään. Tavallisesti käyttöjärjestelmän toteuttamassa keskeytyskäsittelijässä voidaan suorittaa prosessien hallintaan liittyviä toimenpiteitä. Esimerkiksi toisen prosessin siirtäminen suoritukseen tai jonkin prosessin pysäyttäminen voivat olla tällaisia toimintoja.

Keskeytyksillä voidaan pysäyttää ohjelman suoritus helposti halutussa kohtaa, mutta puhtaasti keskeytyksiin perustuva askeltava suoritus vaatii huomattavaa tukea käyttöjärjestelmältä. Keskeytyksiä ei siis ole järkevää käyttää muualla kuin laiteohjelmissa, joissa ei ole valmista käyttöjärjestelmää ohjelman ja alustan välissä. Yleisesti käyttöjärjestelmät toteuttavat omat keskeytyskäsittelijänsä, jolloin keskeytykset eivät edes näy sovellusohjelmalle asti. Keskeytyksillä voidaan kuitenkin lähettää käyttöjärjestelmältä signaaleja ohjelmalle.

3.2.2. Käyttöjärjestelmän signaalit

Signaalit ovat periaatteellisesti hyvin lähellä keskeytyksiä, mutta ne sijaitsevat yhtä astetta korkeammalla tasolla. Molempien tarkoituksena on ilmoittaa tapahtumista asynkronisesti ja sallia käyttäjän reagoida näihin ilmoituksiin [18]. Signaalimekanismi toteutetaan käyttöjärjestelmän tasolla, jossa ne toimivat abstraktiona keskeytyksille. Kaikista keskeytyksistä ei aiheudu signaalia, eikä signaalien aiheuttaja ole välttämättä keskeytys.

Vaikka signaalien välitysmekanismi on asynkroninen, voi signaalin syy kuitenkin olla myös synkroninen. Tällainen tilanne voi olla kyseessä, jos signaali on aiheutunut ohjelmasta itsestään. Esimerkiksi `SIGFPE` ja `SIGSEGV` ovat signaaleita, joiden aiheutuminen voidaan jäljittää takasin tiettyyn yksittäiseen konekäskyyn. Toisaalta asynkroniset signaalit taas ovat ohjelmasta riippumattomia ja niitä aiheutuu esimerkiksi käyttäjän `kill`-kutsusta [19].

Vaikka signaalit ovatkin kätevä korkean tason mekanismi viestien välittämiseen prosesseille, on niiden käyttö hankalaa monisäikeisten ohjelmien tapauksessa. Koska käyttöjärjestelmä toimittaa signaalin ensimmäiselle halukkaalle säikeelle, ei signaaleja voida helposti kohdistaa mihinkään tiettyyn säikeeseen — tällainen epädeterministisyys ei ole kovinkaan toivottavaa. Signaalien hukkuminen on myös mahdollista, jos yksikään säie ei kykene käsittelemään signaalia. Tarvitaan siis vielä tätäkin korkeamman tason mekanismi askeltavan suorituksen toteuttamiseksi monisäikeiseen prosessiin.

3.2.3. Synkronointimekanismit

Rinnakkaisille ohjelmistoille on kehitetty useita mekanismeja prosessin sisäisen synkronoinnin hallintaan. Yksinkertaisin menetelmä on käyttää mutexeja, jotka mahdollistavat jaetun resurssin yhteiskäytön usean säikeen kesken. Ohjelmakoodi koostuu usein niin sanotuista kriittisistä alueista, joihin pääsemistä tulee rajoittaa. Yleensä kriittisille alueille päästetään vain yksi säie kerrallaan ja se suoritetaan yhtenä pilkkomattomana kokonaisuutena. Poissuljettavien säikeiden kesken jaettu mutex-olio tarjoaa tähän mahdollisuuden rajapinnastaan löytyvien `lock`- ja `unlock`-metodien avulla. Kriittiselle alueelle tultaessa säikeet kutsuvat `lock`-metodia, joka yrittää lukita mutexin. Jos lukko on jo varattu, jää säie odottamaan sen vapautumista. Kriittiseltä alueelta poistuttaessa lukko vapautetaan `unlock`-metodilla, jolloin jokin toinen mahdollisesti odottavista säikeistä vapautetaan jatkamaan suoritustaan. Tällöin kriittisellä alueella voi olla vain yksi säie kerrallaan suorituksessa. [11]

Hieman mutexia monimutkaisempi mekanismi on semafori, jonka avulla voidaan pysäyttää ja käynnistää säikeitä tarpeen mukaan — juuri sitä, mitä haluamme askeltavassa suorituksessa tehdä. Mutex voidaan toteuttaa semaforin avulla, ja se voi-

daankin myös käsittää vain semaforin erikoistukseksi. Semaforien toiminta perustuu `P`- ja `V`-operaatioihin, joilla käsitellään semaforin sisäistä laskuria. `P` vähentää laskurin arvoa yhdellä, ja jos laskurin arvo on vähennyksen jälkeen negatiivinen, estetään kyseisen säikeen suoritus. `V`-metodi taas kasvattaa laskurin arvoa yhdellä. Jos alkuarvolla nolla olevaa laskuria kasvatetaan, vapautetaan automaattisesti yksi odottavista säikeistä jatkamaan suoritustaan. Semaforeille voidaan lisäksi asettaa haluttu alkuarvo, joten niiden avulla voidaan synkronoida myös useamman kuin kahden säikeen suoritusta. Esimerkiksi alkuarvolla neljä voidaan helposti rajoittaa suoritus neljään yhtäaikaiseen säikeeseen. [11]

Toinen, huomattavasti semaforeja yksinkertaisempi mekanismi on binäärisemafori, joka on toteutettu esimerkiksi Python-ohjelmointikieleen tapahtumaobjektina (*event object*) [7]. Tapahtumaobjektin rajapinta koostuu funktioista, joilla voidaan asettaa sisäisen boolean-muuttujan arvo sekä `wait`-metodista. Säikeet, jotka kutsuvat objektin `wait`-metodia, estetään, kunnes sisäinen muuttuja saa arvon `true`. Jos muuttuja on jo valmiiksi `true`, palataan `wait`-kutsusta välittömästi. Tämän sisäisen muuttujan käsittelyyn rajapinta tarjoaa omat metodinsa. Tällainen toimintatapa on lähes ideaalinen prosessien askellukseen, koska sen avulla voidaan hallita suorituksen etenemistä yksinkertaisesti luomalla pisteitä, joissa tarkistetaan saadaanko suoritusta jatkaa. Tätä mekanismia ei kuitenkaan löydy suoraan yleisimmistä `C++`-kirjastoista, mutta onneksi se on mahdollista toteuttaa helposti käyttäen ehtomuuttujaa (*condition variable*) ja mutexia. Tapahtumaobjekti voidaankin käsittää vain kääreeksi (*wrapper*) näiden kahden synkronointiprimitiivin ympärille.

Kun mutexit hoitavat poissulkemisen rajoittamalla pääsyä dataan, sallivat ehtomuuttujat säikeiden synkronoitua datan arvon perusteella [17]. Ehtomuuttuja on tarkkailija-suunnittelumallia [6] mukaileva mekanismi, joka samalla myös automatisoi mutexien käyttöä. Mutexeilla estetään kilpailutilanne (*race condition*) tapahtumaobjektin sisällä. Koska säie jää odottamaan ehtomuuttujaa kriittisen alueen sisälle, suorittaa ehtomuuttuja automaattisesti mutexin lukitsemis- ja avaamisoperaatioita — esimerkiksi `wait`-operaation yhteydessä.

Pseudokoodilla kuvattu esimerkkitoteutus `EventObject`-luokasta löytyy liitteestä 1. Sen toiminta on kuvattu sekvenssikaavion avulla liitteessä 2. Sekvenssikaavio kuvaa myös ehtomuuttujan automaattisesti suorittamat mutex-operaatiot. Kuten `EventObject`-luokan rajapinnasta huomataan, on sen käyttö erittäin yksinkertaista. Työsäikeissä vaaditaan ainoastaan kutsu tapahtumaobjektin `wait`-metodiin, ja kontrollisäikeessä taas kutsutaan sen `clear`- ja `set`-metodeita.

3.3. Askelluksen toteutusmekanismi

Askelluksen toteuttamiseksi tarvitaan kaksi komponenttia: askeltaja ja syklinen este. Askeltaja hallitsee prosessin suoritusta määrittämällä pisteitä, joiden kohdalla säikeet on mahdollista pysäyttää. Syklinen este taas mahdollistaa säikeiden suorittamisen synkronoidusti — siten, ettei mikään säie pääse ”karkaamaan” muilta.

3.3.1. Askeltaja

Askelluksen perustaksi sopii parhaiten tapahtumaobjektiin perustuva mekanismi. Siinä ei kuitenkaan ole suoraan mahdollisuutta kontrolloida askellusta halutulla tavalla, joten kyseiseen luokkaan täytyy tehdä pieniä muutoksia. Liitteessä 3 on toteutus, joka on tehty `C++`-kielellä käyttäen POSIX-järjestelmien `pthread`-kirjastoa.

Jotta rajapinta vastaisi paremmin luokan käyttötarkoitusta, annetaan kutsulle `set` uusi nimi, `go`. Tämän funktion merkitys hieman muuttuu nimen vaihtuessa, mutta toiminta on sama kuin alun perinkin. Funktiokutsu `go` sallii suorituksen edistyä ilman esteitä. Lisäksi `clear`in nimeksi vaihdetaan `stop`, koska pysäyttäminen kuvaa paremmin sen käyttötarkoitusta ja toimintaa — `stop` nimensä mukaisesti pysäyttää askelluksen seuraavaan odotustilaan.

Näiden muutosten lisäksi rajapintaan lisätään uusi funktio `step`, joka sallii kaikkien odottavien säikeiden suorituksen edistyä yhden askeleen verran. Sen toteutus on lähes identtinen `go`n kanssa, mutta sisäisen muuttujan arvoksi asetetaankin `false`. Tämä muuttaa toimintaa `wait`-metodin sisällä, jossa sisäisen muuttujan avulla tarkastetaan, tarvitseeko kutsujan odottaa.

Tapahtumaobjekti mahdollistaa ohjelman suorittamisen askeltaen. Se ei kuitenkaan huomioi askelten synkronointia, vaan riippuen käyttöjärjestelmän vuorontajan (*scheduler*) toiminnasta saattaa jokin säie olla monta askelta muita edellä. Tähän haasteeseen on ratkaisuna esimerkiksi Java-ohjelmointikielen ja Boost-kirjaston toteuttama syklinen este (*cyclic barrier*) [4; 2].

3.3.2. Syklinen este

Kuvitellaan tilanne, jossa ryhmä retkeilijöitä on suunnitellut vaellusreitin muutamalla välietapilla. Jokaisella välillä kukin henkilö saa vaeltaa omaa tahtiaan, mutta levähdyspaikoilta lähdetään aina yhtä aikaa. Syklinen este mahdollistaa tällaisen synkronoinnin. Nimessä ”syklinen” viittaa siihen, että kyseistä objektia voidaan käyttää uudelleen. Tässä kohdassa käsitellään `C++`-kielellä tehtyä ja pthread-kirjastoa käyttävää esimerkkiä syklisen esteen toteutuksesta, joka löytyy liitteestä 4.

Myös syklisen esteen toiminta perustuu tapahtumaobjektin tavoin ehtomuuttujan hyödyntämiseen. Kummankin luokan tapauksessa `wait`-metodin kutsuja jätetään odottamaan jotain tapahtumaa. Tapahtumaobjekti vapautetaan manuaalisella

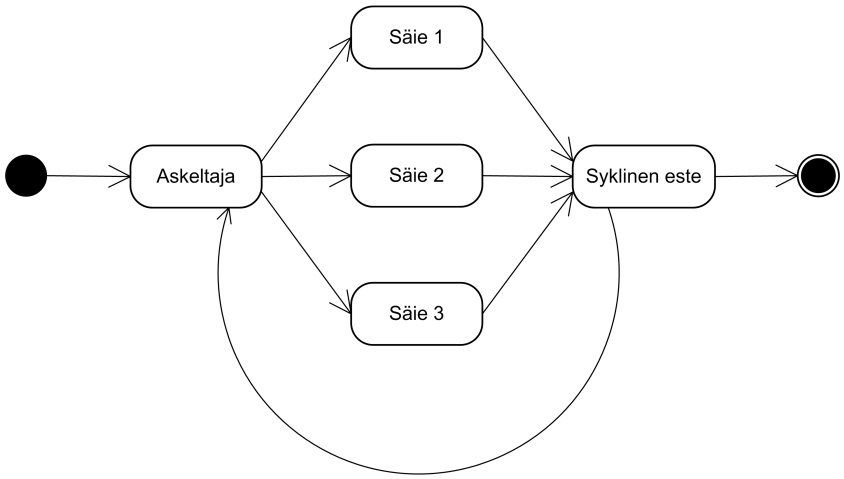
kutsulla sen jäsenmetodiin, kun taas syklisen esteen vapauttaminen on automaattista kaikkien säikeiden saavutettua saman pisteen suorituksessa.

Ehtomuuttujaa käytetään säikeiden odotuttamiseen ja herättämiseen. Mutexilla taas suojataan kriittistä aluetta, jossa käsitellään odottavien säikeiden määrän sisältävää laskuria. Vaikka toteutus onkin periaatetasolla yksinkertainen, pientä monimutkaisuutta siihen tuo `generation`-muuttujan käyttö. Kyseisellä muuttujalla pidetään kirjaa syklisen esteen ”järjestysnumerosta”, jotta säikeet saadaan pysähtymään varmasti samaan pisteeseen. Jos kyseistä muuttujaa ei olisi, pysähtyisivät säikeet aina myös ollessaan muista säikeistä jäljessä. Tämän kaltainen tilanne voisi syntyä, jos säikeet käynnistettäisiin eri aikaan eikä niiden aloitusta synkronoitaisi.

Vaikka toteutus on hiukan tapahtumaobjektia monimutkaisempi, on luokan käyttö kuitenkin helpompaa. Sen rajapinnasta löytyy ainoastaan yksi funktio, `wait`, jota kutsumalla säie jää odottamaan muita säikeitä. Kun rakentajan parametrina annettu määrä säikeitä on kutsunut kyseistä funktiota, vapautetaan kaikkien suoritus jatkamaan eteenpäin.

3.3.3. Synkronoitu askellus

Yhdistämällä tapahtumaobjekti ja syklinen este saadaan toiminnallisuus, joka askelluksen lisäksi mahdollistaa myös säikeiden keskinäisen synkronoinnin (kuva 3.5). Kun molempien olioiden `wait`-metodia kutsutaan peräkkäin voidaan varmistua siitä, että säikeet ovat sekä askellettavissa ja että ne suorittavat jokaisen askeleen synkronisesti. Metodien kutsujärjestyksellä ei tässä tapauksessa ole merkitystä, sillä lopputulos on molemmissa sama. Järjestyksen tulee kuitenkin olla yhdenmukainen kaikissa säikeissä.



Kuva 3.5: Askeltajan ja syklisen esteen yhteistoiminta säikeiden suorituksen hallinnassa. Kuva ei huomioi säikeiden elinikää.

Aiempia esimerkkiluokkia hyödyntävä testiohjelma löytyy liitteestä 5. Kyseisessä ohjelmassa on mahdollista hallita suorituksen edistymistä askeleen tarkkuudella säikeiden keskinäinen synkronointi säilyttäen. Säikeiden dynaaminen luominen ei kuitenkaan ole mahdollista. Hallinta tapahtuu komentoriviltä käyttäen käskyjä `go`, `step` ja `stop`. Nämä komennot pääohjelma muuttaa suoraan kutsuiksi askeltajalle. Työsäikeiltä vaaditaan kutsu sekä `Controller`- että `Barrier`-olioiden `wait`-metodeihin.

3.4. Askellus hajautetussa ympäristössä

Säikeistetyssä — mutta hajauttamattomassa — ohjelmistossa askeltamisesta tulee haasteellista, koska synkronoinnin toteutus säikeiden välillä on huomattava toteutustekninen ja ylläpidollinen rasite. Hajautettu ympäristö ei kärsi säikeistyksen ongelmista.

Viestinvälitys helpottaa askellusta huomattavasti. Se abstrahoi monimutkaiset asiat yksinkertaisen viestiväylän taakse, joka vaikuttaa ohjelman toteuttamiseen positiivisesti. Koska viestiväylään liittyneet komponentit ovat itsenäisiä laskentayksiköitään, mahdollistaa viestiväylä tehtävien yksinkertaisen hallinnan ilman monisäikeistyksen perinteisiä haasteita.

4. PROSESSIEN MONITOROINTI

Prosessien monitoroinnilla tarkoitetaan toimintaa, jossa prosessin suorituksen edistymistä valvotaan joko passiivisesti tai aktiivisesti. Monitoroinnin tarkoituksena on pitää kirjaa prosessin suorituksen tilasta ja löytää poikkeamia sen toiminnasta, esimerkiksi mahdollisten virhetilanteiden selvittämistä varten. Monitoroinnilla pystytään myös reagoimaan toimintojen loogisesti väärään järjestykseen.

> ”Process monitoring is a continuous realtime task of recognizing anomalies in the behavior of a dynamic system and identifying the underlying faults.”
>
> — Dvorak, Kuipers [13]

Yksinkertainen esimerkki monitoroinnista on varsinkin sulautetuissa ohjelmistoissa yleisesti käytetty vahtikoira-ajastin (*watchdog timer*, WDT). Sen tarkoituksena on palauttaa ohjelman suoritus alkutilaan tilanteessa, jossa ohjelma ei enää reagoi normaalilla tavalla annettuihin syötteisiin [8]. Vahtikoiran aktivoinnin jälkeen prosessin tulee ilmoittaa sille tietyin väliajoin olevansa vielä toiminnassa, tai ohjelma katsotaan jumiutuneeksi. Vahtikoiran toteutus ei kuitenkaan ole täysin triviaalia kaikissa järjestelmissä. Se vaatii tukea laitteistolta, koska pelkästään ohjelmistollisesti valvonnasta ei saada riittävän tehokasta — pelkkä ohjelmalaskurin nollaus ei välttämättä riitä. Vahtikoira-ajastin liitetäänkin usein suoraan kiinni suorittimen reset-linjaan. [8]

Esimerkki huomattavasti kehittyneemmästä monitoroinnista on lentokoneissa käytettävä musta laatikko (*flight data recorder*, FDR), joka nauhoittaa jopa tuhansien lentokoneen järjestelmien tilan tarkasti mahdollistaen jopa toimintojen lähes täydellinen jäljittämisen [3]. Tästä on korvaamaton apu onnettomuustilanteiden selvittämisessä, koska sen avulla onnettomuustutkijat saavat tarkkaa tietoa lentokoneen järjestelmien toiminnan lisäksi lentäjien ohjausliikkeistä ja lentokoneen antureiden mittaustuloksista (lentokorkeus, ilmanopeus, kohtauskulma, ...).

Sovelluksen sisäinen monitorointi on mahdollista toteuttaa niin yksi- kuin monisäikeisessäkin ympäristössä. Jos kuitenkin järjestelmässä on käytössä vain yksi säie, asettaa se pieniä rajoitteita monitoroinnille. Tällöin monitorointia täytyy esimerkiksi suorittaa samalla prioriteetilla alkuperäisen toiminnallisuuden kanssa, jolloin monitorointifunktion suoritus saattaa häiritä itse toiminnallisuuden suoritusta. Monisäikeisessä ympäristössä valvontaa voidaan tehdä pienemmällä prioriteetilla, jolloin se ei vaikuta negatiivisesti prosessin suorituskykyyn. Riippumattomuuden ansiosta on säikeistyksen etuna mahdollisuus prosessoida saatua dataa samalla, kun uusia toimintoja jo suoritetaan.

Prosessien monitorointia on tämän diplomityön käyttämässä merkityksessä käsitelty kirjallisuudessa vähän, eikä eri menetelmille löydy vakiintuneita nimiä. Siksi tämän työn yhteydessä monitorointi on jaettu kahteen alaluokkaan: tilaperustaiseen ja tapahtumaperustaiseen monitorointiin. Käsittely rajataan myös koskemaan vain monisäikeistä ympäristöä.

4.1. Tilaperustainen monitorointi

Tilaperustaisella monitoroinnilla voidaan tarkoittaa mekanismia, jossa prosessin sisällä toimiva valvontasäie kysyy valvottavalta komponentilta sen tilaa aina tietyin väliajoin. Tällainen toimintatapa sopii varsinkin tilanteisiin, jossa on tärkeää säilyttää väli monitorointitapahtumien välillä vakiona. Tilaperustaisessa monitoroinnissa katsotaan järjestelmän kokonaistilan seuraamisen olevan yksittäisten tapahtumien kirjaamista tärkeämpää.

Monimutkaisten järjestelmien tilaperustaisessa valvonnassa voi haasteita kuitenkin tulla seurattavien arvojen suuresta määrästä ja lyhyestä seurantavälistä. Monien järjestelmien pieni tallennuskapasiteetti rajoittaa tallentavan monitoroinnin aikaväliä. Menetelmää voidaan optimoida tallentamalla vain muuttuneet arvot, mutta sekään ei ratkaise tilaongelmaa lopullisesti. Parempi vaihtoehto on käyttää lentokoneen mustan laatikon tyyppistä menetelmää, jossa vanhat mittaukset pudotetaan automaattisesti pois tilan loppuessa. Tämän käyttökelpoisuus kuitenkin riippuu täysin monitoroinnin sovelluskohteesta.

4.2. Tapahtumaperustainen monitorointi

Toinen tapa valvoa suorituksen tilaa on käyttää aktiivista monitorointia, jossa valvottava komponentti itse ilmoittaa tapahtumien sattuessa tilastaan muualle. Monitoroinnin rajaaminen koskemaan vain uusien tapahtumien kirjaamista voi pienentää sen kokonaisvaikutusta ohjelman suorituskykyyn huomattavasti. Tapahtumaintensiivisissä ympäristöissä voi tästä kuitenkin muodostua suuri kuorma monitoroinnille. On siis tarkkaan valittava kumpaa tekniikkaa käytetään monitorointiin. Lisäksi on myös mahdollista yhdistää molempien mekanismien parhaat puolet yhteen järjestelmään, jossa tilaperustaista monitorointia hyödynnetään jatkuvasti muuttuvien suureiden valvonnassa, kun taas tapahtumapohjaisesti tallennetaan harvakseltaan muuttuvien tapahtumien tiedot.

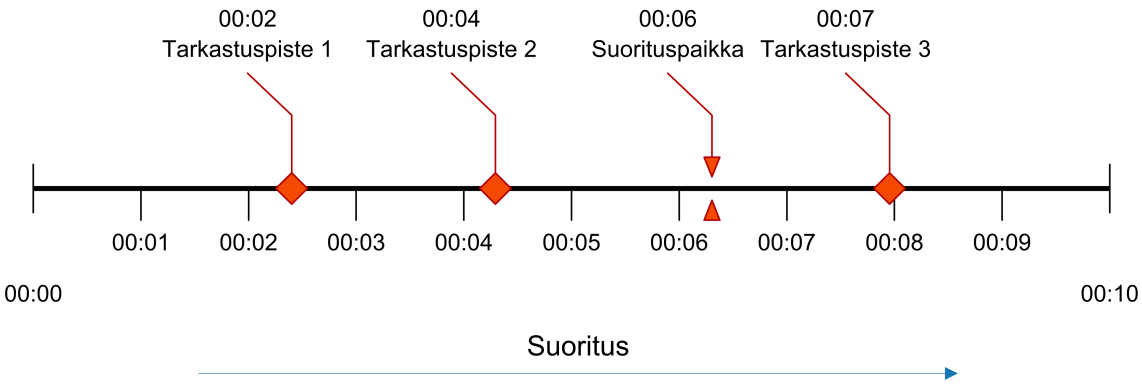
4.3. Menetelmiä prosessin monitorointiin

4.3.1. Intuitiivinen tapa

Tarkoituksena on kehittää sovellus, jonka tila on aina selvitettävissä. Edellä mainituista mekanismeista tapahtumaperustainen monitorointi sopii sovelluksen valvontaan paremmin, koska säikeet suorittavat toimintojaan epäsäännöllisesti ja suhteellisen harvakseltaan. Tilaperustainen monitorointi sopii paremmin esimerkiksi erinäisten anturien valvontaan, eikä sitä kannata hyödyntää tällaisten ohjelmistojen tapauksessa.

Tapahtumaperustaisuus myös mahdollistaa valvonnan tason määrittämisen. Koska ei ole järkevää tarkastella ohjelman tilaa liian tarkasti, kannattaa keskittyä tutkimaan toiminnallisuuskokonaisuuksien suoritusta, jolloin valvonta ei muodostu liian raskaaksi operaatioksi. Mitä suurempia kokonaisuuksia valvonnassa käsitellään, sitä pienemmäksi myös monitoroinnin aiheuttama kuorma muodostuu.

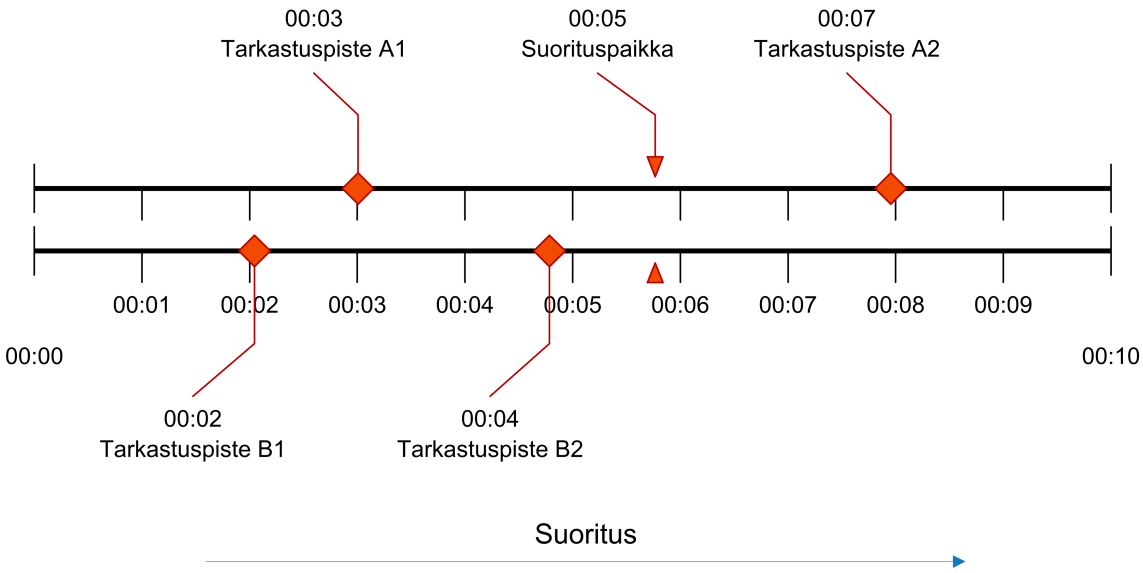
Tapahtumakokonaisuuksien erottelussa otetaan käyttöön tarkastuspisteen (*checkpoint*) käsite. Se kuvaa pistettä suorituksessa, jossa jokin tietty toiminto on saatu suoritettua kokonaisuudessaan loppuun asti. Tämä mahdollistaa monitoroinnin rajoittamisen vain kiinnostaviin tiloihin. Sen avulla voidaan helposti ryhmittää myös ohjelman suoritus loogisiin kokonaisuuksiin.



Kuva 4.6: Tarkastuspisteet ja suorituspaikka yhden säikeen tapauksessa

Yksisäikeisessä ohjelmassa monitorointi on helppo toteuttaa tarkastuspisteiden avulla, koska se ei edellytä tietoa säikeiden keskinäisestä tilasta (kuva 4.6). Yksinkertaisimmillaan monitorointi tapahtuu siten, että valvottava prosessi tallentaa viimeisimmän suorittamansa toiminnon johonkin tietorakenteeseensa. Koska prosessi itse tietää toimintojen järjestyksen, voidaan tästä tiedosta päätellä myös mitä suoritetaan seuraavaksi.

Monisäikeisessä ympäristössä (kuva 4.7) tilanne ei ole aivan näin yksinkertainen. Koska suoritus voi olla yhtä aikaa eri säikeissä monen eri suorituspisteen välillä,



Kuva 4.7: Tarkastuspisteet ja suorituspaikka usean säikeen tapauksessa

vaaditaan kirjanpitoon hiukan monimutkaisempaa toiminnallisuutta. Teoriassa asiaa voisi yksinkertaistaa kasvattamalla abstraktiotasoa, jolloin siirryttäisiin monitoroimaan mahdollisesti korkeammalta tasolta löytyviä selkeitä toimintokokonaisuuksia. Tämä ei kuitenkaan ole hyväksyttävä ratkaisu, koska kaikkia toimintoja ei ole mahdollista käsitellä suurempina kokonaisuuksina järkevästi.

On siis välttämätöntä käsitellä jokaista säiettä erikseen. Tämä voidaan toteuttaa samaan tapaan kuin yksisäikeisessäkin ympäristössä, mutta tilannetietoon täytyy liittää tieto siihen liittyvästä suoritussäikeestä. Haasteita aiheutuu kuitenkin siitä, ettei säikeiden elinikä ole välttämättä etukäteen tiedossa. Tähän voidaan ottaa ratkaisuksi säikeiden rekisteröinti, jossa säie ilmoittaa käynnistymisestään ja sammumisestaan komponentille, joka pitää kirjaa suorituksen tilasta. Jos säie ei itse sisällä yhtään tarkastuspistettä, ei myöskään rekisteröintiä tarvita.

Tällainen mekanismi prosessin monitorointiin on mahdollista lisätä ohjelmistoon myös jälkikäteen. Se ei välttämättä vaadi suuriakaan muutoksia ohjelmiston rakenteeseen, mutta on kuitenkin operaationa työläs ja vaikea ylläpidettävä. Jos monitoroinnilta halutaan joustavuutta, tarvitaan täysin erilainen lähestymistapa, joka on otettu huomioon jo ohjelmiston arkkitehtuurisuunnittelusta lähtien.

4.3.2. Monitorointikomponentti

Prosessin suoritusta voidaan valvoa myös täysin erilaisesta lähtökohdasta. Edellä kuvatussa tekniikassa säikeet itse ilmoittavat valvojalle tilansa, mutta tämä voidaan kääntää myös täysin päälaelleen. Jos koko järjestelmä rakennetaankin monitorointikomponentin päälle, on sillä täydellinen tilannekuva ohjelman suorituksesta. Havaitaan, että tällaiseen monitorointikomponenttiin on helppoa ja jopa luonnollista

sisällyttää myös suorituksen hallinnan vaatima toiminnallisuus. Tällöin se myös tietää aina, missä tilassa ohjelma kullakin hetkellä on.

Monitorointiin ja askeltajaan perustuva arkkitehtuuri mahdollistaa laskentapolkujen muuttamisen dynaamisesti. Polkujen määrittelyssä tulee linkittää tilat toisiinsa sekä määrittää virhetilanteissa käytettävät paluureitit. Näiden tietojen avulla askeltaja osaa ohjata suorituksen etenemistä ja monitori voi saada hyvän kuvan prosessin tilasta. Yksisäikeisessä ympäristössä tällainen mekanismi on yksinkertainen toteuttaa, mutta kohdeympäristön ollessa monisäikeinen ei asia ole kuitenkaan yhtä yksinkertainen.

Monisäikeisessä ympäristössä haasteet ovat samat kuin aiemmissakin menetelmissä. Järjestelmän pohjana oleva tuoterunko perustuu kuitenkin viestinvälitysarkkitehtuuriin, joka tarjoaa tähän yksinkertaisen ratkaisun. Koska ohjelmistotuote koostuu viesteillä kommunikoivista komponenteista, on viestien lähetys ainut keino käynnistää toimintoja ohjelmistossa. Samaan aikaan voi olla useita toimintoja yhtäaikaisesti ajossa, jolloin on järkevää osoittaa kullekin tehtäväjonolle oma hallinta- tai monitorointisäikeensä. Tällöin jokainen säie hallitsee oman tehtäväjononsa suoritusta autonomisesti, kuitenkin ottaen vastaan suoritusta ohjaavia viestejä. Monitori pystyy jaksottamaan suorituksen käyttäen lähetettyjä tehtävänaloitusviestejä tarkastuspisteinään.

Sisäisesti komponentit voivat käynnistää useita säikeitä, jos suoritus sitä vaatii. Jokainen komponentti itse huolehtii sisäisesti rinnakkaistuksestaan, eikä tämä muodostu ongelmaksi komponenttien välillä. Rinnakkaisuuden hallinta komponenttien sisällä ei kuitenkaan näy askeltajalle mitenkään, ja askeltajan näkökulmasta onkin yhdentekevää miten komponentti toteuttaa toimintonsa suorittamisen. Rinnakkaisuuden hallinta ei siis muodostu ongelmaksi askeltajan toteutuksessa.

Jos askeltajan olemassaoloa ei ole huomioitu jo arkkitehtuurisuunnittelusta lähtien, on sen lisääminen jälkikäteen ohjelmistoon todella työlästä. Järjestelmän tuoterunko on kuitenkin suunniteltu modulaariseksi, jolloin uusien komponenttien toteuttaminen on helppoa. Askellus voidaan toteuttaa järjestelmään komponenttina, joka on mahdollista tarpeen vaatiessa myös poistaa käytöstä. Komponentin toteutus on huomattavasti yksinkertaisempaa kuin askellusmekanismin toteutus monoliittiseen järjestelmään — jälkimmäisessä tapauksessa se vaatisi luultavasti huomattavan määrän refaktorointia sekä uudelleen suunnittelua.

Hajautetun ympäristön tuoma etu on selkeä verrattuna puhtaasti säikeistetyn ohjelmiston askellukseen. Abstraktiotason kasvu yksinkertaistaa toteutusta suorituskyvyn kustannuksella, mutta rinnakkaisuusongelmien yksinkertaistamiseksi tällainen on helposti perusteltavissa. Askeltajakomponentti myös mahdollistaa paremman joustavuuden, koska yksittäisiä tehtäväkokonaisuuksia ei tarvitse määrittää käännösaikana.

5. ASKELTAJA

Ohjelmistojen kasvaessa riittävän monimutkaisiksi muuttuu suorituksen valvonta ja virhetilanteiden selvittäminen hankalaksi. Tähän ratkaisuksi kehitettiin askeltaja, joka pitää kirjaa ohjelman suorituksen tilasta sekä valvoo ja hallitsee sen edistymistä. Askeltajalla voidaan selvittää, missä tilassa ohjelma milläkin hetkellä on, ja mitä se aikoo suorittaa seuraavaksi. Tämä helpottaa virheiden selvittämistä, koska ohjelman tilan tulkitseminen yksinkertaistuu huomattavasti.

Askeltajakomponentti hallisee ohjelman loogista suorituspolkua. Se määrää toimintojen suoritusjärjestyksen sekä niiden ajankohdan. Askeltaja tuo siis hajautettuun tuoterunkoon proseduraalisuutta. Askeltaja käsittelee suorituksessa sekvenssejä, joilla kuvataan järjestelmän eri tehtäviä. Näistä kuvauksista se muodostaa suorituspolkuja, jotka askeltaja suorittaa käyttäjän haluamalla tavalla.

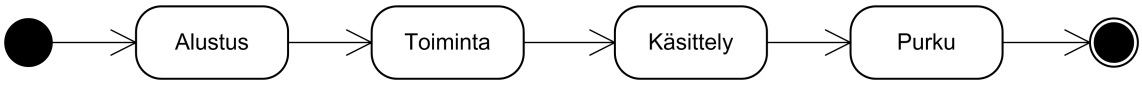
Esimerkki askeltajan mahdollisista sovelluskohteista on erinäisten testien ajaminen kohdejärjestelmässä. Usein näihin testeihin liittyy alustustoimenpiteitä, jotka suoritetaan ennen varsinaisen testin ajamista sekä lopuksi suoritettavia purkutoimenpiteitä. Testiajo voi sisältää useita testitapauksia. Alustustoimenpiteissä voidaan esimerkiksi ladata jokin komponentti järjestelmän käyttöön ja purkutoimenpiteissä vastaavasti poistaa se käytöstä. Askeltajan tehtävänä on hallita ohjelman suoritusta siten, että näitä tehtäviä voidaan suorittaa rinnakkaisesti ja toisistaan riippumatta.

Askeltaja ohjaa suoritusta dynaamisesti määritettyjen suorituspolkujen avulla, ja se myös mahdollistaa suorituksen kontrolloinnin yhden askeleen tarkkuudella. Koska kyseessä on vahvasti hajautettu ja rinnakkainen järjestelmä, täytyy askeltajan pystyä hallitsemaan useita eri suorituspolkuja samanaikaisesti — vaikka ne suoritettaisiin fyysisesti eri laitteissa.

5.1. Sekvenssipolut

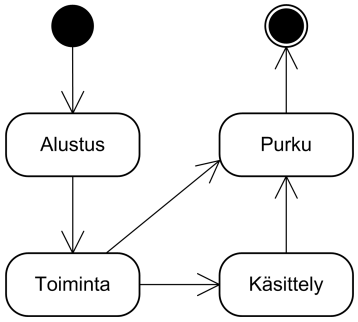
5.1.1. Tilakone

Koska askeltajan tarkoituksena on hallita ohjelman suoritusta tilojen avulla, on luonnollista kuvata sen toimintaa tilakoneilla. Kuva 5.8 esittää yksinkertaista järjestelmää, joka suorittaa tietyn sekvenssin erinäisiä toimintoja. Toimintojen järjestys on ennalta määrätty ja tilakoneesta on selvästi eroteltavissa suorituksen eri vaiheet.



Kuva 5.8: Yksinkertainen tilakone

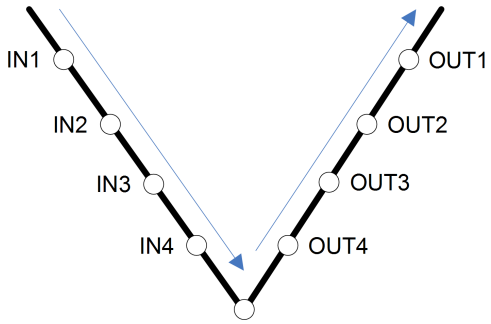
Näin yksinkertainen tilakone ei kuitenkaan vastaan todellista tilannetta, jossa yksinkertaisessakin tilakoneessa voi olla useita haaraumia suorituspolussa. Virhetilanteiden käsittely on yleinen tapaus, jossa suorituksen tulee päästä lopputilaan, muttei tavanomaista reittiä pitkin. Virhetilanteissa on luonnollista, että virheen sattuessa siirrytään purkamaan ohjelman suoritusta ja ohitetaan kaikki virhekohdan ja purkukohdan väliset toiminnot (kuva 5.9).



Kuva 5.9: Tilakone, jossa on otettu myös virheistä toipuminen huomioon

Kun virhetilanteiden siirtymät otetaan mukaan tilakoneisiin, huomataan että ne on järkevää piirtää v-mallia jäljitellen — tällöin kuvasta voidaan helposti erotella alustustoimet niihin liittyvistä purkutoimenpiteistä.

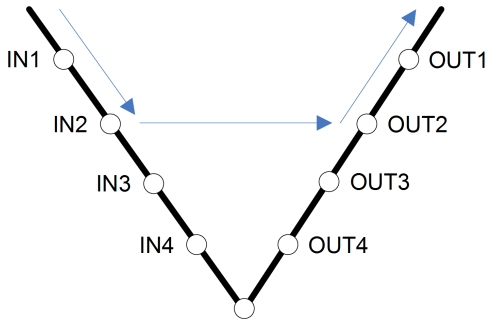
5.1.2. V-malli



Kuva 5.10: V-mallin perustapaus

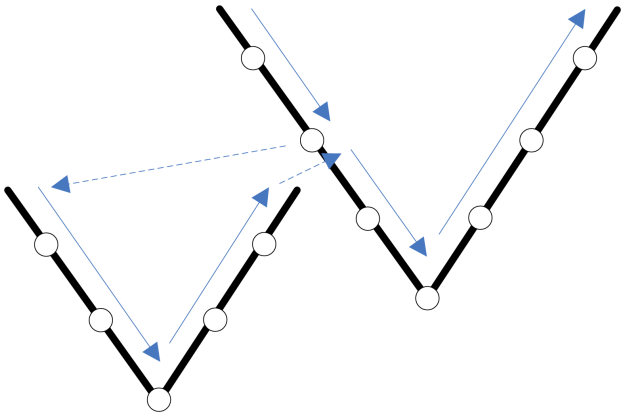
Ohjelman suoritus voidaan ajatella koostuvan v-mallisista poluista (kuva 5.10), joiden vasen (laskeva) sivu kuvastaa alustustoimenpiteitä, ja oikea (nouseva) sivu taas kuvastaa purkutoimenpiteitä. V-mallin pohjalta on tarkoitus suorittaa toimenpiteitä, jotka vaativat näiden alustus- ja purkutoimenpiteiden suorittamisen. Tällaistä polkua — joka voi esimerkiksi kuvata yhden monesta toiminnosta koostuvan tehtävän suoritusta — kutsutaan sekvenssiksi. [16]

Jos jossakin alustustoimenpiteessä tapahtuu virhe, hypätään siihen linkittyneeseen purkutoimenpiteeseen (kuva 5.11). Tällöin voidaan varmistua siitä, että kaikki



Kuva 5.11: V-malli, joka kuvaa virhetilannetta alustusvaiheessa

alustettu myös puretaan. On myös tärkeää säilyttää tehtävien järjestys siten, ettei myöhemmin suoritettu alustustoimenpide hyppää pidemmälle purkutoimenpiteissä kuin jokin aikaisemmin suoritettu alustus. Tilannetta voidaan kuvitella piirtämällä toimintojen välille kuvitteelliset linkkiviivat, jolloin niiden ei tule leikata toisiaan. Siispä virhetilanteen sattuessa mahdollisesti vuorossa olleet alustustoimenpiteet ohitetaan, ja siirrytään suoraan purkamaan jo suoritettuja alustustoimenpiteitä.

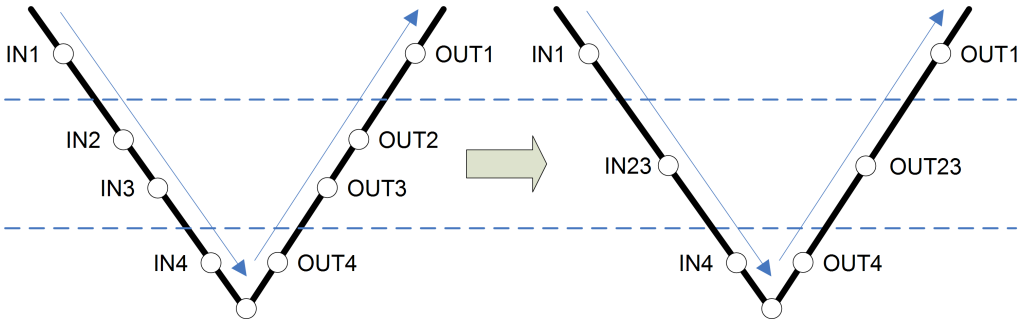


Kuva 5.12: V-mallien riippuvuus toisistaan

Sen lisäksi, että eri toimintoja voi sekvenssipolussa olla vaihteleva määrä, voi toiminto linkittyä johonkin toiseen sekvenssipolkuun (kuva 5.12). Nämä toiminnot toimivat hyppyinä jonkin toisen sekvenssipolun alkutilaan, ja kyseisen sekvenssin suorituksen päätyttyä palataan takaisin alkuperäisen sekvenssin tilaan, josta jatketaan suoritusta normaalisti eteenpäin. Jos tällaisessä *alisekvenssissä* tapahtuu virhe, voidaan myös pääsekvenssi määritellä epäonnistuneeksi, ja toimia samoin kuin perinteisen tehtävän epäonnistuttua.

Toiminto kuvaa yhtä sekvenssin etappia, joka sisältää tietyn loogisen tapahtuman. Tällainen tapahtuma voi olla esimerkiksi jonkin komponentin alustus. Toiminnon ei kuitenkaan tarvitse olla pelkästään yksittäinen tapahtuma, vaan se voi olla myös oma sekvenssinsä. Käytännössä tällainen toiminto onkin oikeasti hyppy toisen sekvenssin alkuun. Pääsekvenssi näkee alisekvenssin vain yksittäisenä toimintona

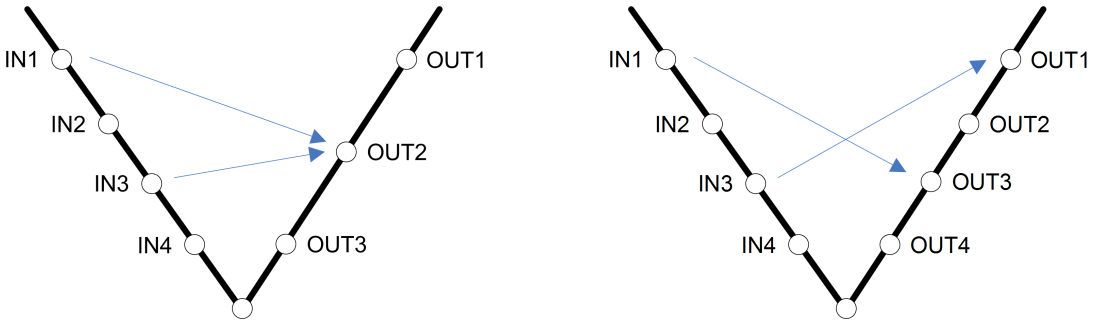
joka on suoritettu kokonaan vasta sitten, kun suoritus on palannut alisekvenssin polulta takaisin.



Kuva 5.13: Toimintojen ryhmitys

Periaatteessa yksi perustoiminto voisi koostua myös useista peräkkäisistä toiminnoista olematta kuitenkaan puhtaasti sisäkkäinen sekvenssi. Kuten kuvasta 5.13 nähdään, toimintojen ryhmittäminen ei tuo abstraktiotason kasvamisen lisäksi suurtakaan lisäarvoa sekvenssin kuvaamiselle, mutta saattaa vaikeuttaa myöhemmin kuvattavaa toimintojen optimointia.

Toisin kuin ohjelmistotuotannon v-mallissa, ei sekvenssipolun molemmissa jaksoissa tarvitse olla yhtenevää määrää toimintoja. Jokaisesta alustussekvenssin toiminnosta on linkki johonkin purkutoimintoon, mutta näitä ei tarvitse olla sama määrä. Yksi tai useampia alustustoimintoja voi siis linkittyä yhteen purkutoimintoon (kuva 5.14a). Linkkien määrää yhden toiminnon suhteen ei myöskään ole rajoitettu, joten tarvittaessa kaikki alustustoiminnot voidaan linkittää vain yhteen ainoaan purkutoimintoon. Yksi alustustoiminto ei kuitenkaan voi linkittyä useaan purkutoimintoon.



(a) Oikein *(b)* Väärin

Kuva 5.14: Linkit toimintojen välillä

Kuten kuvasta 5.14b nähdään, tulee huolehtia siitä etteivät linkit leikkaa toisiaan. Leikkauksen tapahtuessa muuttuu toimintojen järjestys sellaiseksi, että suorituksessa saatetaan purkaa jotain alustamatonta tai vaihtoehtoisesti jättää jotain

kokonaan purkamatta. Tämä ei tietenkään ole haluttu tilanne, ja tälle rajoitteelle on olemassa myös toteutustekninen peruste.

Käytännössä linkkiajattelu toteutetaan tasoina, jossa jokaiselle toiminnolle on määrätty ennalta jokin numeerinen tasoarvo. Arvo kasvaa sitä suuremmaksi, mitä lähemmäs sekvenssin pohjaa lähestytään, ja pienenee taas purkuvaiheessa. Tämän arvon perusteella askeltaja osaa siirtää suorituksen alustuksista oikeaan purkutoimenpiteeseen. Tasot ovat tehtäväkohtaisia, eikä ylemmän tason tehtävillä ole tietoa niiden sisältämien alitehtävien tasoista. Virheenkäsittely tapahtuu siis yksittäisten tehtävien sisällä, ja tehtävien sisäinen virhepolitiikka päättää raportoidaanko virheistä ylemmälle tasolle. On myös mahdollista käsitellä virheet siten, että ylemmän tehtävän näkökulmasta ei alitehtävässä edes tapahtunut mitään virhettä.

5.2. Askeltajan toiminta

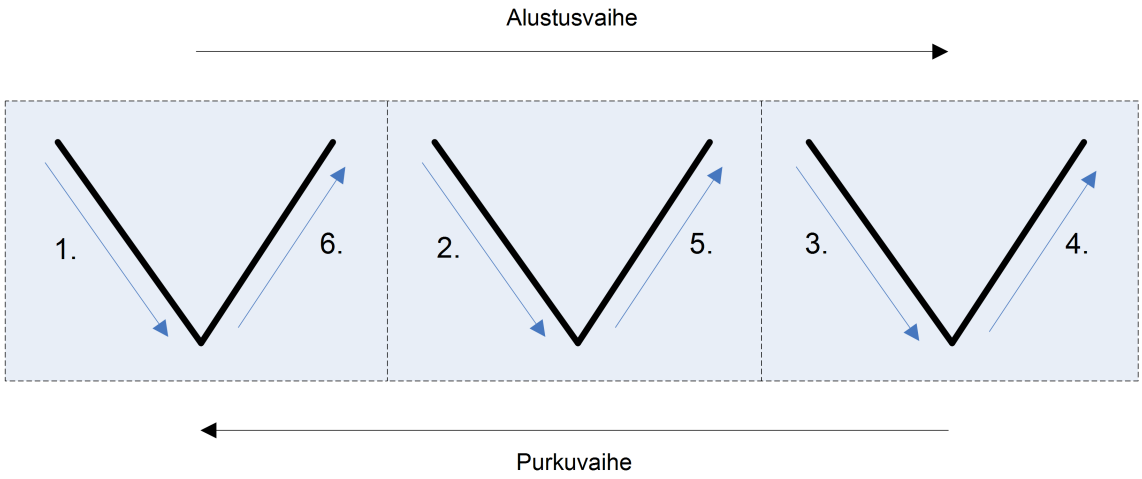
5.2.1. Tehtävien suorittaminen

Askeltaja saa ilmoituksen suoritettavasta sekvenssistä viestinä, joka sisältää sekvenssille annetun nimen sekä listan sen sisältämistä tehtävistä. Tämä viesti voi tulla miltä tahansa komponentilta. Sekvenssien sisältämiä tehtäviä suoritetaan sarjallisesti, mutta askeltaja voi ylläpitää myös monen eri sekvenssin suoritusta samanaikaisesti. Askeltaja käynnistää jokaiselle sekvenssille oman säikeensä. Yksittäinen säie hallitsee siis aina yhden sekvenssin suorittamista, mutta sillä ei kuitenkaan ole varsinaisesti omaa liityntää viestiväylään. Kaikki viestimekanismiin liittyvät operaatiot tapahtuvat sekvenssejä hallitsevan olion jäsenmetodien avulla. Yksittäiselle sekvenssille tulevat viestit vastaanotetaan hallintaoliossa, joka ilmoittaa saapuneesta viestistä eteenpäin.

Sekvenssi sisältää listan suoritettavista tehtävistä, johon voidaan dynaamisesti lisätä uusia tehtäviä. Listan tehtävät suoritetaan siten, että jokaisesta tehtävästä suoritetaan ensin alustustoiminnot, ja sen jälkeen purkutoiminnot ajetaan käänteisessä järjestyksessä lopusta alkuun päin (kuva 5.15).

Listan loppuun lisättäviä uusia tehtäviä saattaa tulla toisten tehtävien suorituksesta. Esimerkiksi ensimmäisenä suoritettu toiminto saattaa käskeä askeltajaa lisäämään sekvenssin loppuun joihinkin uusiin komponentteihin liittyviä toimintoja. Tehtävien lisäys on loogisesti järkevää vain alustustoimintojen aikana, jolloin uuden tehtävän lisäys ei eroa tilanteesta, jossa tehtävä olisi ollut listassa jo alusta alkaen.

Askeltaja osaa automaattisesti järjestää tehtävien sisältämät toiminnot oikeaan järjestykseen niiden sisältämien parametrien perusteella, ja tämän jälkeen se siirtyy suorittamaan listan ensimmäisen tehtävän ensimmäistä toimintoa. Tehtäviin ja yksittäisiin toimintoihin liittyvät tarkemmat tiedot askeltaja noutaa tietokannasta. Näitä tietoja ovat muun muassa tehtävään sisältyvät toiminnot parametreineen.



Kuva 5.15: Sekvenssin tehtävälistan suoritusjärjestys

Toiminnon sijaintitieto on oleellista paikan selvittämisessä. Käytännössä sijaintitieto kuvataan tason ja ryhmän avulla, joista jälkimmäinen kertoo onko toiminto alustus- vai purkutoimenpide. Suorittamista voidaan ohjata toimintoihin liittyvillä parametreilla. Esimerkiksi tehtävään liittyvä ohjaustieto voi määrätä kyseisen tehtävän virhepolitiikan, ja toimintoon liittyvä tieto taas kertoa toistojen määrän.

Toiminnon suorittaminen aloitetaan lähettämällä siitä vastuussa olevalle komponentille viesti käyttäen järjestelmän viestinvälitysmekanismia. Koska viestinvälitysmekanismi on asynkroninen, voi askeltaja hallita samanaikaisesti useiden tehtävien suoritusta toisistaan riippumatta. Viestin lähettämisen jälkeen askeltajan säie jää odottamaan ilmoitusta suorituksen valmistumisesta.

Kun komponentti on saanut toiminnon suoritettua, lähettää se sekvenssin viestinkäsittelijälle takaisin kuittausviestin joko onnistuneesta tai epäonnistuneesta suorituksesta. Suorituksen epäonnistuttua askeltaja välittää valitun virhepolitiikan perusteella tiedon myös ylemmälle tehtävälle tai lopulta tehtävälistalle. Onnistuneen suorituksen tapauksessa askeltaja siirtyy suoraan seuraavaan toimintoon.

Sekvenssiä voidaan suorittaa kahdella eri tavalla – manuaalisesti tai automaattisesti. Manuaalisesta suorittamisessa askeltaja pysähtyy odottamaa jatkamiskäskyä jokaisen toiminnon jälkeen. Automaattisessa suorittamisessa taas askeltaja aloittaa suoraan seuraavan toiminnon suorittamisen edellisen valmistuttua. Manuaalisen suorittamisen tapauksessa odottaminen koskee vain haluttua sekvenssiä – muut voivat jatkaa suoritustaan normaalisti eteenpäin.

Kun tehtävä on suoritettu alustustoimintojen loppuun asti ja kyseisessä tehtävässä ei ole enää mitään suoritettavaa ennen purkutoimintoja, siirtyy askeltaja listan seuraavaan tehtävään. Uudet tehtävät suoritetaan samalla periaatteella kuin aiemmatkin, mutta viimeisen tehtävän alustustoimintojen valmistuttua siirrytään purkamaan suoritusta. Purkaminen tapahtuu yksittäisen tehtävän tasolla samassa

järjestyksessä kuin alustustoimenpiteetkin, mutta tehtävälistassa edetään lopusta alkuun päin. Kun kaikki tehtävät on suoritettu loppuun asti, on myös koko sekvenssin suoritus päättynyt. Tämän jälkeen askeltaja ilmoittaa sekvenssin valmistumisesta ja vapauttaa kyseisen sekvenssin varaamat resurssit.

5.2.2. Virhetilanteiden käsittely

Virhetilanteet tapahtuvat toimintoja suoritettaessa, ja niistä ilmoitetaan askeltajalle viestillä. Jos virhetilanne tapahtuu alustustoimenpiteissä, reagoi askeltaja ilmoitukseen siirtämällä suorituksen vastaavalle tasolle purkutoimenpiteissä. Purkutoimintojen tapauksessa taas ainut järkevä toipumiskeino on jatkaa niiden suorittamista edelleen. Siirtymät alustustoiminnoista purkutoimintoihin tapahtuvat aina yhden tehtävän sisällä, mutta tehtävälle voidaan määrittää parametrilla, miten se raportoi virheistä ylemmälle tasolle. Tätä virhepolitiikkaa vaihtamalla voidaan virheet jättää myös kokonaan huomiotta tai käsitellä ne täysin tehtävän sisällä kertomatta virheestä eteenpäin.

Ylimmän tason tehtävistä tehtävälistalle asti vuotavat virheet aiheuttavat purkuoperaatioiden aloittamisen koko sekvenssin laajuudessa. Jos virhetilanne päätyy tehtävälistalle asti on kyseisen virheen aiheuttanut tehtävä jo suorittanut oman virhekäsittelynsä, jolloin on tarpeellista purkaa vain jo kokonaan alustetut tehtävät — ja nekin käänteisessä järjestyksessä alustukseen nähden.

5.2.3. Tietokantaliityntä

Askeltaja viittaa tehtäviin niiden nimien perusteella, jolloin niiden tarkemmat tiedot täytyy noutaa jostain ulkopuolisesta lähteestä ja säilöä paikallisesti ohjelman muistiin. Ulkopuolisella lähteellä viitataan tässä tapauksessa tietokantaan, mutta periaatteessa olisi mahdollista noutaa tiedot myös jostain muusta tietovarannosta. Tietokannan puuttuessa esimerkiksi XML-muotoinen tallennus voisi tulla kyseeseen.

Tietokannasta voidaan noutaa tehtävien kuvauksia sekä toimintojen parametreja. Myös kokonaisten tehtävälistojen noutaminen on mahdollista. Tällöin askeltajan kutsujan ei aina tarvitse välittää tietoa kaikista tehtävälistaan kuuluvista tehtävistä, vaan pelkkä sekvenssin nimen kertominen riittää.

Tiedot noudetaan nimien perusteella ja noudettu data tallennetaan tietokantakuorman vähentämiseksi paikalliseen välimuistiin. Jos välimuistia ei käytettäisi ja jokaisen viittauksen yhteydessä tiedot noudettaisiin aina uudestaan, saattaisi monen yhtäaikaisen sekvenssin suoritus yhdistettynä toimintojen määrän optimointiin (kohta 6.7) aiheuttaa useita satoja tietokantakyselyjä sekuntia kohti.

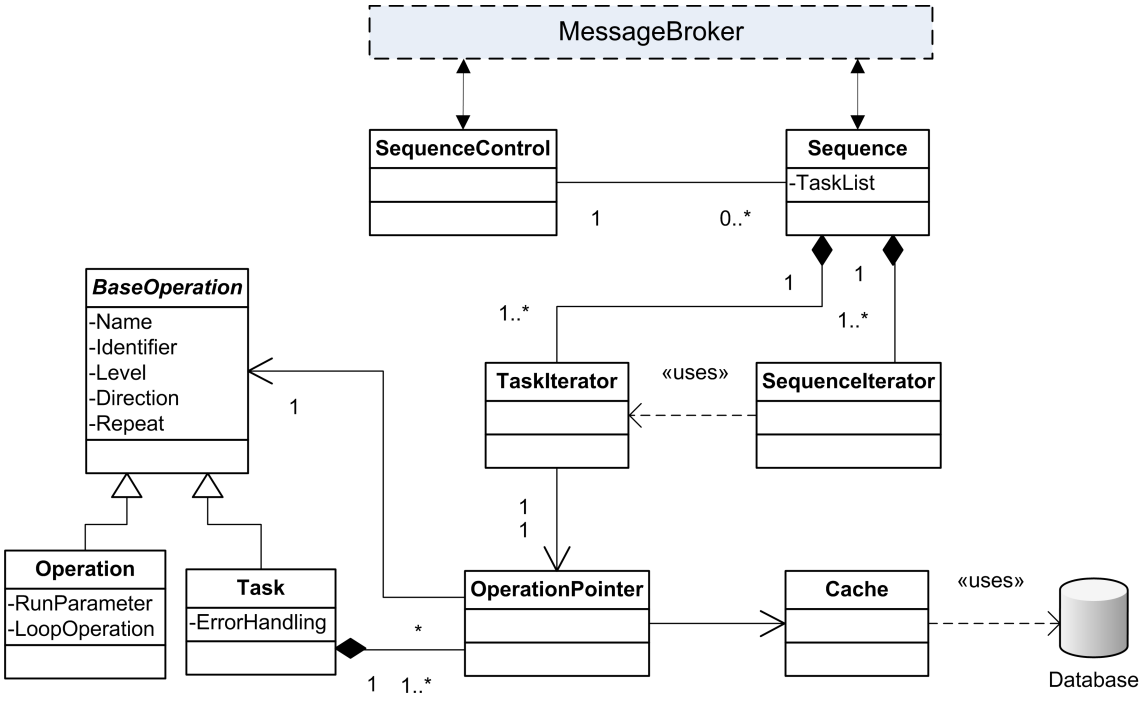
6. TOTEUTUS

Tässä luvussa kuvataan askeltajan toteuttamiseksi vaadittavat oliot, tietorakenteet ja algoritmit. Luokat esitellään niiden välisten suhteiden ja toimintatapojen tasolla — tarkkoja rajapintoja ei määritetä tämän diplomityön puitteissa.

6.1. Askeltajan rakenne

6.1.1. Luokkarakenne

Askeltaja koostuu useista luokista (kuva 6.16), jotka on jaettu loogisesti omiksi osakomponenteikseen vastuualueidensa perusteella.



Kuva 6.16: Askeltajakomponentin keskeisimmät luokat ja niiden väliset suhteet

SequenceControl on luokkahierarkiassa kaikista korkeimmalla tasolla ja se toimii rajapintana muille komponenteille. `SequenceControl`in tehtävänä on hallita sekvenssiolioita, eikä se sisällä mitään muuta ohjaukseen liittyvää toimintalogiikkaa. Sen tehtävänä on ainoastaan käynnistää uusia sekvenssejä ja poistaa valmistuneita. `SequenceControl`-oliolta voidaan myös kysyä sekvenssien suorituksen tilaa.

Sequence on vastuussa yhden sekvenssin suorituksesta. Se hallitsee suoritusta täysin itsenäisesti, mutta käyttää `SequenceControl`-luokan palveluita viestien lähettämisessä. `Sequence`-olio ylläpitää listaa tehtävistä, joita se suorittaa koh-

dassa 5.2.1 kerrotulla tavalla. Olio on elossa vain siihen liittyvän sekvenssin suorituksen ajan.

TaskIterator toteuttaa iteraattorisuunnittelumallin ja sitä käytetään tehtävien ja toimintojen muodostaman puurakenteen läpikäyntiin. Luokka sisältää yksittäisen tehtävän läpikäymiseen tarvittavan toimintalogiikan ja sen virheenkäsittelyn.

SequenceIterator hallitsee `TaskIterator`-olioiden avulla kokonaisen sekvenssin suoritusta. Luokan rajapinta on hyvin samankaltainen `TaskIterator`in kanssa, mutta se sisältää tehtävälistan läpikäymiseen vaadittavan toimintalogiikan. Luokka toteuttaa myös tilojen optimoinnissa tarvittavan toiminnallisuuden, koska optimointialgoritmi edellyttää pääsyä sekvenssin globaaliin tilan.

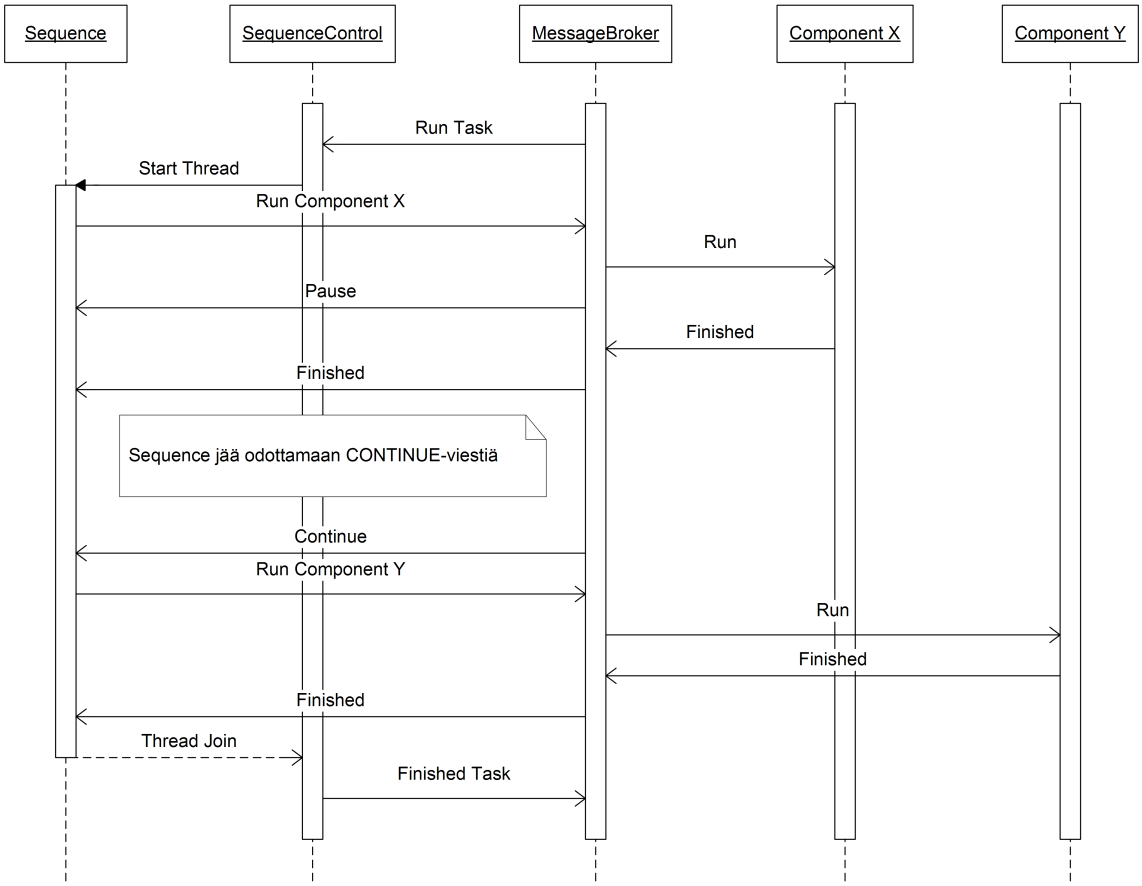
OperationPointer on osoitinmuuttujien laiskaan alustukseen (*lazy initialization*) käytettävä kääreluokka (*wrapper*). Luokan avulla voidaan linkit tehtävien välillä muodostaa käyttäen niiden nimiä, ja nämä linkit evaluoidaan osoittimiksi vasta niitä tarvittaessa. `OperationPointer`-olioita käytetään tehtävissä, kun halutaan viitata sen sisältämiin toimintoihin. Luokan tarkoituksena on lisätä järjestelmän joustavuutta ja mahdollistaa puurakenteiden kuvaus käyttämättä muistiosoitteita.

Cache toimii paikallisena tallennuspaikkana, josta `OperationPointer` noutaa nimen perusteella tarvittavia olioita. `Cache` toimii samalla myös rajapintana tietokannan suuntaan ja ylläpitää välimuistissaan tietokannasta noudettuja tehtävien kuvauksia. `Cache`a laajentamalla voidaan tuoda tuki myös muille tietovarastoille tietokannan lisäksi. Noudetut tehtävät ja toiminnot säilötään lokaaliin välimuistiin, joka on käyttäjälle täysin läpinäkyvä — `Cache`n käyttäjä ei siis tiedä noudettiinko haluttu olio oikeasti tietokannasta, vai palautettiinko vain lokaali kopio. Välimuistiin haetut kopiot poistetaan tietyin väliajoin, jotta mahdollisesti muuttuneet toimintokuvaukset päivittyvät myös sekvenssien suoritukseen. On kuitenkin pidettävä huolta, ettei toimintokuvaus päivity kesken sitä käyttävän sekvenssin suorituksen — esimerkiksi toimintojen poistaminen tehtävästä kesken sen suorituksen saattaa aiheuttaa suoritukseen määrittelemättömän toiminnan.

Luokkien välisiä suhteita selvitetään kuvassa 6.17, josta käy ilmi yksinkertaisen sekvenssin suoritus.

6.1.2. Tehtävien ja toimintojen mallinnus

Perustana toiminnoille ja tehtäville on `BaseOperation`-luokka, joka sisältää molemmille yhteiset ominaisuudet eli nimen, tason, suunnan sekä toistojen määrän. Se on



Kuva 6.17: Yksinkertaisen sekvenssin suoritus

määritetty abstraktiksi kantaluokaksi, jolloin sitä ei voi instantioida. `BaseOperation`-luokasta on kuitenkin periytetty `Operation`-luokka, joka kuvaa yksittäistä toimintoa ja `Task`-luokka, joka kuvaa tehtävää.

Tehtävät ja toiminnot mallinnetaan askeltajakomponentin sisällä puurakenteena. Rekursiokoostesuunnittelumallin [15, s. 109–116] mukaisesti tehtävät sisältävät viitteitä `BaseOperation`-olioihin, jolloin tehtävät voivat olla myös sisäkkäisiä. Olioiden väliset riippuvuudet on toteutettu käyttäen `OperationPointer`-olioita, jolloin osoitinmuuttujien sijaan voidaan käyttää toimintojen nimiä viitteinä ja osoittimen arvo evaluoidaan vasta tarvittaessa. Tämä on tarpeen joustavuuden lisäämiseksi, koska muutoin tehtävien rekursiiviset suhteet olisivat erittäin haastavia toteuttaa.

6.2. Iteraattorit

6.2.1. TaskIterator

Sekvenssin tehtävät ja operaatiot ovat tallennettuna puutietorakenteeseen. Tällaisen rekursiokoosteen läpikäyminen ei ole aivan triviaalia, mutta toisaalta se muistuttaa paljon esimerkiksi hakemistopuun selaamista. Perusmekanismi tietorakenteiden se-

laamiseen on iteraattorisuunnittelumallin mukainen, jota myös C++:n standardikirjasto noudattaa. Standardikirjasto on osa C++-ohjelmointikieltä ja se tarjoaa sovelluskehittäjän käyttöön yleisimpiä tietorakenteita ja niihin liittyviä palveluita [10]. Koska C++:n iteraattorimekanismi on hyväksi havaittu, pyritään myös askeltajakomponentin suunnittelussa samankaltaiseen rajapintaan ja toiminnallisuuteen.

Yksinkertaisten tietorakenteiden — esimerkiksi taulukoiden — iteraattorit on usein toteutettu yksinkertaisesti osoitinmuuttujina. Iteraattori voi kuitenkin olla myös luokkatyyppinen, jolloin seuraajaoperaatioon (`operator++`) on mahdollista sisällyttää myös monimutkaista toiminnallisuutta. Iteraattoriluokka ei myöskään ole toiminnaltaan yksinkertainen, koska sen täytyy pitää kirjaa esimerkiksi sisäkkäisistä tehtävistä. Tähän kirjanpitoon liittyy esimerkiksi niiden tilan muistaminen myös uusien sisäkkäisten tehtävien suorituksen ajan. Tämän ratkaisemiseksi sovelletaan iteraattorin toteutuksessa Memento-suunnittelumallia.

Memento-suunnittelumallin mukaisessa toteutuksessa voidaan olion tila tallentaa johonkin ulkopuoliseen olioon. Iteraattorissa tätä hyödynnetään sijaintitieton ylläpitämiseksi yksittäisen tehtävän sisällä. Tämä yksinään ei kuitenkaan vielä riitä iteraattorin toteuttamiseksi. Memento-olion ja iteraattorirajapinnan väliin tarvitaan kolmaskin luokka, joka muodostaa pinon sisäkkäisistä tehtävistä. Iteraattori käyttää siis sisäisesti kahta apuluokkaa, `TaskState`a ja `TaskStack`ia, joista ensimmäinen muistaa tilan yhden tehtävän sisällä, ja toinen taas pinoaa edellämainittuja iteraattoreita kohdatessaan sisäkkäisiä tehtäviä kesken tietorakenteen selauksen.

6.2.2. SequenceIterator

Tehtävälistan läpikäymiseksi toteutetaan myös toinen iteraattoriluokka, jonka tehtävänä on hallita tehtävälistan suoritusta `TaskIterator`eiden avulla. Korkeamman tason lähestymistapa tarvitaan, koska listan suorituslogiikka poikkeaa selvästi yksittäisen tehtävän läpikäymisestä (kuva 5.15). Tehtävät toimivat omina suorituskokonaisuuksinaan, joilla on oma sisäinen logiikkansa. `SequenceIterator` ei ole tietoinen `TaskIterator`in toimintalogiikasta, mutta se tietää silti, missä tehtävässä `TaskIterator` kullakin hetkellä on. `SequenceIterator` tarvitsee toiminnassaan tietoa siitä, onko kyseessä purku- vai alustustoiminto, ja kuinka syvällä tehtäväpuussa suoritus on.

Tehtävälistan suoritus voidaan jakaa kahteen vaiheeseen — alustamiseen ja purkamiseen. Alustusvaiheessa `SequenceIterator` siirtyy aina listan seuraavaan tehtävään kohdatessaan ensimmäisen purkutoiminnon. Kun kaikki alustustoiminnot on suoritettu siirrytään purkutilaan, jossa suoritetaan jokainen yksittäinen tehtävä loppuun asti käyden listaa läpi päinvastaisessa järjestyksessä.

6.3. Suoritusmoodit

Sekvenssi voidaan suorittaa usealla eri tavalla: automaattisesti, manuaalisesti tai pysäyttäen alustustoimintojen jälkeen. Automaattisessa suorituksessa askeltaja käynnistää välittömästi seuraavan toiminnon suorituksen edellisen valmistuttua, kun taas manuaalisessa moodissa se jää odottamaan jatkamisen oikeuttavaa CONTINUE-viestiä. Alustustoimintojen jälkeen pysäyttävässä moodissa toiminta on lähes sama kuin automaattisessa suorituksessa, mutta suoritus pysäytetään automaattisesti alustustoimintojen valmistuttua. Suoritusmoodia voidaan vaihtaa kesken sekvenssin suorituksen lähettämällä sille ExecutionMode-viesti.

6.4. Toimintojen ja tehtävien ominaisuudet

Jokaiseen toimintoon ja tehtävään liittyy kiinteästi muutamia tietoja, joita ei ole mahdollista muuttaa dynaamisesti. Näillä tiedoilla esimerkiksi määritetään kohteen sijainti sekvenssissä sekä annetaan sille nimi. Näiden kenttien arvoihin ei voida vaikuttaa sekvenssin suoritusta aloitettaessa.

Name Nimen avulla erotetaan järjestelmän eri tehtävät ja toiminnot toisistaan. Nimellä on merkkijonotyyppinen arvo, joka voi koostua kirjaimista, numeroista ja pisteistä. Nimi toimii samalla myös komponentin viestijonon tunnisteena, joten toiminnot käynnistetään lähettämällä viesti nimen osoittamaan kohteeseen. On kuitenkin huomioitava, ettei tätä tule sekoittaa sekvenssin nimeen, joka myös toimii viestijonon tunnisteena, mutta eri yhteydessä.

Identifier Kaikille toiminnoille ja tehtäville voidaan määrittää erityinen tunniste, jonka avulla optimoinnissa (kohta 6.7) poistetaan toisteiset toiminnot. Tunnisteella on merkkijonotyyppinen arvo, joka voi koostua kirjaimista, numeroista ja pisteistä. Tavanomaisessa tilanteessa sama tunniste määritetään yhdelle alustustoiminnolle ja yhdelle purkutoiminnolle, jotka sijaitsevat saman tehtävän sisällä, jolloin ne muodostavat yhdessä loogisen parin. Tätä ei kuitenkaan ole ohjelmallisesti rajoitettu, joten ”parit” voivat muodostua myös useammasta kuin kahdesta toiminnosta. Teoriassa on mahdollista muodostaa pareja myös eri tehtävien välille, mutta tämä vaikeuttaa tunnisteiden ylläpitoa huomattavasti, koska tunnisteiden muodostamat parit eivät tällöin ole suoraan nähtävissä. Tämän ominaisuuden käyttötarkoitus on kuvattu kattavammin kohdassa 6.7.

Level Jotta toiminnon paikka yksittäisen tehtävän sisällä voidaan määrittää, annetaan sille tasoa kuvaava numeerinen lukuarvo. Nollataso on sekvenssin yläosassa, ja suurimmat tason arvot saadaan sekvenssin pohjalla. Tason avulla

tehdään myös virhetilanteessa päätös siitä, mihin purkutilaan siirrytään. Taso kuvautuu siis suoraan tehtävän v-mallisesta kuvauksesta.

Direction Toiminnoille annetaan myös suunta tehtävän sisällä, jolloin se yhdessä tasotiedon kanssa määrittää sijainnin tehtävässä. Arvoiksi tämä ominaisuus voi saada `IN` (alustustoiminto) tai `OUT` (purkutoiminto).

6.5. Parametrit

Toiminnoille ja tehtäville voidaan antaa parametreja, jotka toimivat lisämääreinä niiden suoritukselle. Parametreilla voidaan esimerkiksi hallita suorituksen etenemistä estämällä sen keskeyttäminen tiettyjen toimintojen kohdalla tai sallia purkutoimenpiteiden aloitus vain tietyissä kohdissa, ennalta määrättyjen toimintojen yhteydessä. Parametrit ovat siis ohjeita askeltajalle, mutta myös suoritettava komponentti voi käyttää osaa niistä.

Jos tehtävän sisältämä toiminto on linkki toiseen tehtävään, on sille mahdollista antaa suoritusta ohjaavia parametreja. Näillä parametreilla voidaan määrätä esimerkiksi sekvenssin suoritustapa. Sekvenssi voidaan suorittaa kokonaisuudessaan ilman pysähdyksiä tai siten askeltaen, että jokaisen uuden toiminnon kohdalla odotetaan käskyä jatkaa eteenpäin. Parametreilla on mahdollista ohjata alempien sekvenssien suoritusta ylemmiltä tasoilta.

Parametrit voidaan luokitella ryhmiin niiden kohteen perusteella. Osa parametreista on liitettävissä vain tehtäviin ja osa täytyy antaa kokonaiselle sekvenssille.

6.5.1. Sekvenssiparametrit

Sekvenssiparametrit voidaan antaa vain kokonaiselle sekvenssille.

ExecutionMode määrää sekvenssin suoritusmoodin. Sen avulla voidaan vaikuttaa yleisellä tasolla siihen, kuinka sekvenssi suoritetaan.

Taulukko 1: ExecutionMode-parametrin saamat arvot

Arvo	**Selite**
MANUAL	Sekvenssin suoritus pysäytetään jokaisen toiminnon jälkeen. Suoritusta jatketaan lähettämällä CONTINUE-viesti askeltajalle.
AUTOMATIC	Suoritetaan sekvenssi kokonaisuudessaan.
INIT	Suoritetaan sekvenssistä alustustoiminnot ja pysäytetään sen jälkeen suoritus. Purkutoiminnot aloitetaan CONTINUE-viestillä ja suoritetaan automaattisesti loppuun.

6.5.2. Yhteiset parametrit

Yhteiset parametrit ovat sellaisia, jotka voidaan antaa sekä toiminnoille että tehtäville.

Repeat määrittää suoritettavien toistojen määrän, ja oletuksena kohdetoiminto suoritetaan vain kerran. On myös mahdollista suorittaa toimintoa loputtomiin asettamalla tämän parametrin arvoksi 0. Ikuisesta silmukasta voidaan poistua lähettämällä sekvenssille viesti `EXIT`, joka aiheuttaa purkutoimintojen käynnistymisen. Tehtävälistan tehtävien tapauksessa tämä parametri jätetään huomioitta, koska listan tehtävät suoritetaan aina vain kerran. Oletusarvo on 1.

ExecutionControl mahdollistaa suorituksen ohjaamisen yksittäisen suoritusalkion tarkkuudella. Tällä parametrilla voidaan esimerkikri estää sekvenssin pysäytys toiminnon kohdalla, tai määrittää kokonainen tehtävä atomiseksi, jolloin käyttäjä ei voi keskeyttää sen suoritusta. On myös mahdollista pakottaa suoritus pysähtymään parametrin kohteen jälkeen.

Taulukko 2: ExecutionControl-parametrin saamat arvot

Arvo	**Selite**
ATOMIC	Estää suorituksen pysäyttämisen kesken tehtävän. Voidaan asettaa vain tehtävälle.
NONSTOP	Estää suorituksen pysäyttämisen kohteen jälkeen. Jos ohjelman suoritus kuitenkin yritetään pysäyttää, jatketaan suoritusta kunnes päädytään sellaiseen toimintoon, jonka kohdalla pysäyttäminen on taas sallittua.
STOP	Pysäyttää suorituksen kohteen jälkeen ja jää odottamaan CONTINUE -käskyä. Manuaalisessa suoritusmoodissa tällä parametrilla ei ole vaikutusta, koska suoritus pysähtyy joka tapauksessa.

IfVariable mahdollistaa suorittamisen ehdollisesti, jolloin suoritusta ennen evaluoidaan muuttujan arvo ja verrataan sitä annettuun arvoon. Tälle parametrille annetaan tekstimuotoisena arvona sen muuttujan nimi, jonka arvoa verrataan. Sen lisäksi tulee määritellä IfValue -parametri, joka kertoo arvon, johon muuttujaa verrataan. Oletusarvoisesti operaattorina käytetään yhtäsuuruusoperaattoria, mutta tarvittaessa operaattori voidaan vaihtaa liittämällä viestiin mukaan IfOperator-parametri. Jos arvo-muuttuja-pari on halutulla operaattorilla evaluoitaessa tosi suoritetaan toiminto; muussa tapauksessa se ohitetaan. Muuttujien toteutus on osa tuoterunkoa, eikä niiden toimintaa kuvata tarkemmin tässä diplomityössä.

IfOperator mahdollistaa vertailuoperaattorin vaihtamisen IfVariable -parametrin yhteydessä. Tämän parametrin avulla voidaan määrittää, mitä vertailuoperaattoria muuttujien vertailussa käytetään.

Taulukko 3: IfOperator-parametrin saamat arvot. IfValue on operaattorin oikealla puolella.

Arvo	**Selite**
EQUAL	Yhtäsuuruus (oletus)
NOTEQUAL	Epäyhtäsuuruus
LESS	Pienempi kuin
LESSOREQUAL	Pienempi tai yhtäsuuri kuin
GREATER	Suurempi kuin
GREATEROREQUAL	Suurempi tai yhtäsuuri kuin

IfValue muodostaa IfVariable -parametrin kanssa kiinteän parin, eivätkä ne voi esiintyä erikseen. IfValue pitää sisällään arvon, johon IfVariablen määrittämän muuttujan arvoa verrataan. Arvo määritetään tekstimuotoisena.

6.5.3. Toimintoparametrit

Toimintoparametreilla voidaan vaikuttaa yksittäisiin toimintoihin.

RunParameter sallii datan välittämisen suoritettavalle toiminnolle. Parametrin arvo välitetään osana RUN-komentoa ja se mahdollistaa myös dynaamisesti laskettavien arvojen lähettämisen.

LoopOperation mahdollistaa suorituksen rajoittamisen vain tiettyihin toimintoihin toiminnan nopeuttamiseksi. Manuaalisessa suorituksessa tämän parametrin arvo jätetään huomioitta.

6.5.4. Tehtäväparametrit

Tehtäväparametrit voidaan antaa vain kokonaisille tehtäville.

ErrorHandling määrittää toiminnan virhetilanteessa, eli virhepolitiikan. Parametrin arvolla voidaan vaikuttaa tehtävien tapaan reagoida virheisiin sekä tapaan välittää tieto virheestä eteenpäin tehtävien kutsupinossa. Tämä parametri vaikuttaa myös siihen, miten tehtävälista reagoi virhetilanteeseen.

Taulukko 4: ErrorHandling-parametrin saamat arvot

Arvo	**Selite**
CONTINUE	Suoritetaan purkutoimenpiteet kyseisestä tehtävästä, mutta jatketaan kuitenkin ylemmällä tasolla normaalisti eteenpäin.
EXIT	Suoritetaan purkutoimenpiteet kyseisestä tehtävästä ja välitetään tieto virheestä eteenpäin kutsupinossa. Tämä on myös virheenkäsittelyn oletustoiminta.
IGNORE	Jatketaan tehtävän suoritusta ja jätetään virhe huomioitta.

6.6. Viestiväylä ja viestit

Tuoterunko hyödyntää viestiväylää kaikessa kommunikoinnissaan. Viestiväylällä liikkuvat viestit on kääritty tuoterungolle spesifisen binääriformaatin sisään, mutta tuoterunko tarjoaa rajapinnan binääristen viestien käsittelyyn. Tuoterunko tarjoaa viestejä kuvaavat luokat sekä muutaman apuluokan yleisten tehtävien helpottamiseksi. Luokista löytyvät myös rajapintafunktiot viestien luomisen yksinkertaistamiseksi. Sovellusohjelmoijalle viestit näkyvät vain tuoterungon tarjoaman rajapinnan läpi, mutta tuoterungon komponentit mahdollistavat myös muunlaisia esitystapoja viesteille.

Järjestelmään voidaan ladata erityisiä streamer-komponentteja, jotka mahdollistavat viestien määrittämisen myös tekstimuotoisena — esimerkiksi käyttäen XML:ä. Viestien hierarkkisen rakenteen kuvaamiseen juuri XML onkin omiaan. Streamerit muuttavat tekstinä määritetyt pakettikuvaukset binäärisiksi viesteiksi, jolloin niitä on kätevä hyödyntää ohjelmiston kehityksessä nopeaan testaukseen. Uusia streamerkomponentteja luomalla voidaan tuoterunkoon tuoda tuki uusille kuvausformaateille.

Kaikille järjestelmän komponenteille määritetään yksikäsitteinen nimi, joka toimii sen viestijonon tunnisteena. Nimiä voidaan jaotella erilaisiin ryhmiin liittämällä niiden nimien eteen ryhmän tunnuksena toimiva etuliite. Myös jälkiliitteitä voidaan käyttää esimerkiksi saman komponentin eri instanssien tunnistamisessa. Liitteiden käyttö mahdollistaa kätevästi viestien osoittamisen suurelle ryhmälle eri komponentteja, koska viestinvälitys tukee jokerimerkkien (*wild card*) käyttöä vastaanottajan tunnistamisessa.

6.6.1. Viestien yleinen rakenne

Viestiväylällä liikkuva tieto kuvataan paketteina, jotka ovat SCS-standardin mukaisia. Alimmalla tasolla tarkasteltuna nämä paketit ovat puhdasta binääridataa, mutta niitä voidaan kuvata käyttäen useita eri notaatioita. Tässä diplomityössä viestien rakenne kuvataan käyttäen geneeristä XML:ää, jossa aaltosuljenotaatiota käytetään

muuttuvissa kentissä. Seuraavassa kohdassa kuvatut viestit eivät myöskään ole täydellisesti minimoituja, vaan niitä on laajennettu esimerkin omaisesti.

Viestit koostuvat avain-arvo-pareista, joissa avaimet ovat merkkijonoja, mutta arvot voivat olla hyvin erilaisia tietotyyppejä: lukuarvoja, merkkijonoja, taulukoita tai jopa toisia paketteja. Pakollisia kenttiä kaikissa viesteissä ovat vastaanottajan ja komennon nimet. `SequenceControl`-oliolle osoitettuihin viesteihin on myös asetettava kohdesekvenssin nimi, jotta viesti päätyy oikealle vastaanottajalle. Tämän lisäksi viestiin voidaan liittää komentoon liittyviä kenttiä, kuten esimerkiksi tehtävien nimiä tai parametreja.

Myös kokonainen paketti voidaan liittää osaksi viestiä. Tätä viestinvälitysjärjestelmä hyödyntää niin sanottujen Success- ja Error-pakettien avulla, jotka voidaan määrittää mihin tahansa viestiin. Niistä toinen lähetetään automaattisesti takaisin viestin lähettäjälle riippuen komponentin suorituksen onnistumisesta. Koska askeltaja luo viestit ja liittää niihin haluamansa Success- ja Error-paketit, voi se tällä tavoin hallita myös saamiensa kuittausviestien sisältöä.

6.6.2. Viestien kuvaukset

Askeltajan pääkomponentti (`SequenceControl`) on ensimmäinen rajapinta, johon ulkopuolinen komponentti ottaa yhteyden, kun se haluaa suorittaa jonkin sekvenssin. `SequenceControl` ottaa vastaan listan tehtävistä, jotka muodostavat halutun sekvenssin. Viestiin voi liittyä myös parametreja, joilla ohjataan sekvenssin suoritusta. Sekvenssikohtaisten parametrien lisäksi niitä voidaan kohdistaa myös sekvenssin sisältämille toiminnoille ja tehtäville. Saamansa aloitusviestin (listaus 1) se delegoi edelleen uudelle säikeelle (`Sequence`), joka hallinnoi kyseisen työn suoritusta. Säikeiden yhdistyttyä `SequenceControl` lähettää alkuperäiselle kutsujalle kuittauksen tehtävän onnistumisesta tai epäonnistumisesta. On huomioitava, että `SequenceControl` toimii välikätenä `Sequence`lle kohdistetuille viesteille, eli se ei käsittele kaikkia alla mainittuja viestejä itse.

Toimintojen käynnistämiseen käytetään `SequenceControl`in lähettämää Run-pakettia (listaus 2). Paketin sisällä välitetään myös Success- ja Error-paketit, joista toisen komponentti lähettää takaisin askeltajalle kun suoritus on saatu valmiiksi. `SequenceControl` saa paketin rungon valmiiksi luotuna, jonka jälkeen siihen liitetään tarvittavat Success- ja Error-paketit. Pakettiin ei tarvitse tehdä muita muutoksia, vaan se on valmiina lähetettäväksi. Pakettien sisällöt on kuvattu yleisellä tasolla, mutta ne voivat sisältää myös muita kenttiä, joista askeltajakomponentti ei ole tietoinen. Run-paketin runko saadaan tehtäväkuvauksien noutamisen yhteydessä valmiina.

Sekvenssin suoritus voidaan pysäyttää seuraavaan sallittuun pysähtymispisteeseen käyttäen Stop-viestiä (listaus 3). Viestin saavuttua sekvenssiä siis suoritetaan

Listaus 1: SequenceControlin vastaanottama Run-viesti

```
<Packet>
   <Receiver>SequenceControl</Receiver>
   <Command>Run</Command>
   <SequenceName>{SequenceName}</SequenceName>
   <ExecutionMode>Automatic</ExecutionMode>
   <Tasks>
      <Task>
         <Name>{TaskName}</Name>
         <Repeat>1</Repeat>
         <ExecutionControl>Atomic</ExecutionControl>
      </Task>
   </Tasks>
</Packet>
```

Listaus 2: Toimintojen vastaanottama ja SequenceControlin lähettämä Run-viesti, yleinen rakenne

```
<Packet>
   <Receiver>{Operation}</Receiver>
   <Command>Run</Command>
   <Packet>
      <Receiver>SequenceControl</Receiver>
      <Command>Success</Command>
      <SequenceName>{SequenceName}</SequenceName>
   </Packet>
   <Packet>
      <Receiver>SequenceControl</Receiver>
      <Command>Error</Command>
      <SequenceName>{SequenceName}</SequenceName>
   </Packet>
</Packet>
```

Listaus 3: Stop-viesti

```
<Packet>
   <Receiver>SequenceControl</Receiver>
   <Command>Stop</Command>
   <SequenceName>{SequenceName}</SequenceName>
</Packet>
```

Listaus 4: Continue-viesti

```
<Packet>
   <Receiver>SequenceControl</Receiver>
   <Command>Continue</Command>
   <SequenceName>{SequenceName}</SequenceName>
</Packet>
```

niin kauan, kunnes saavutaan tilaan, jossa pysäyttäminen on sallittua.

Jos sekvenssi on pysäytettynä, voidaan pysähtyneen sekvenssin suoritusta jatkaa Continue-viestillä (listaus 4). Manuaalisessa suoritusmoodissa suoritus pysähtyy jokaisen toiminnon jälkeen, jolloin suoritusta on aina jatkettava käsin käyttäen Continue-viestiä.

Sekvenssin suoritus voidaan keskeyttää väkisin käyttäen Exit-viestiä (listaus 5). Tällainen tarve voi esiintyä muun muassa, jos suoritus on ikuisessa silmukassa tai sekvenssin suoritus halutaan lopettaa nopeasti mutta hallitusti. Tämän viestin vastaanottamisesta aiheutuu samanlainen suoritustilan purkaminen kuin mikä virhetilanteesta aiheutuisi.

Sekvenssin suoritus voidaan keskeyttää väkisin myös käyttäen Abort-viestiä (listaus 6). Tällöin sekvenssin suoritus keskeytetään niin, ettei edes purkutoimintoja suoriteta. Tämän viestin vastaanotettuaan askeltaja ei kuitenkaan ilmoita juuri sillä hetkellä suorituksessa olevalle toiminnolle keskeyttämisestä, vaan sekvenssiolio

Listaus 5: Exit-viesti

```
<Packet>
   <Receiver>SequenceControl</Receiver>
   <Command>Exit</Command>
   <SequenceName>{SequenceName}</SequenceName>
</Packet>
```

Listaus 6: Abort-viesti

```
<Packet>
   <Receiver>SequenceControl</Receiver>
   <Command>Abort</Command>
   <SequenceName>{SequenceName}</SequenceName>
</Packet>
```

Listaus 7: AddTasksToSequence-viesti

```
<Packet>
   <Receiver>SequenceControl</Receiver>
   <Command>AddTasksToSequence</Command>
   <SequenceName>{SequenceName}</SequenceName>
   <Tasks>
      <Task>
         <Name>{TaskName}</Name>
         <!-- Mahdolliset parametrit -->
      </Task>
   </Tasks>
</Packet>
```

yksinkertaisesti poistetaan askeltajan tietorakenteista. Kyseisellä hetkellä suoritettavana oleva toiminto siis suorittaa laskentansa loppuun asti, vaikkei sitä kutsunutta sekvenssiä olekaan enää olemassa.

Suorituslista-ajattelusta johtuen on sekvenssiin mahdollista lisätä uusia tehtäviä kesken sen suorituksen. Tehtävien lisäämisessä käytetään AddTasksToSequence-viestiä (listaus 7), jolla voidaan lisätä myös useita tehtäviä kerralla sekvenssin tehtävälistan loppuun.

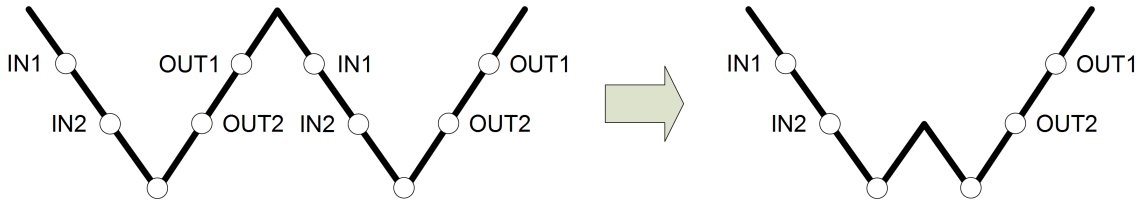
`SequenceControl` voi vastaanottaa tehtävien ja toimintojen kuvauksia myös viestiväylän kautta, jolloin kuvaukset toimitetaan osana OperationDetails-viestiä (listaus 8). Usean eri toiminnon ja tehtävän tiedot voidaan välittää yhdessä viestissä sillä rajoitteella, että toiminnot tulee esitellä ennen tehtäviä. Operaatioiden kuvausten yhteydessä toimitetaan myös viite valmiiseen Run-pakettiin, jota käytetään toiminnon aloittamisessa. Valmiin Run-paketin avulla voidaan toiminnoille välittää mahdollisia lisäparametreja sekä muuta sellaista tietoa, josta askeltajan ei tarvitse olla tietoinen.

Listaus 8: OperationDetails-viesti

```
<Packet>
   <Receiver>SequenceControl</Receiver>
   <Command>OperationDetails</Command>
   <Items>
      <Packet>
         <Name>{Name}</Name>
         <Direction>{Direction}</Direction>
         <Level>{Level}</Level>
         <Identifier>{Identifier}</Identifier>
         <Type>Operation</Type>
         <RunPacket />
      </Packet>
      <Packet>
         <Name>{Name}</Name>
         <Direction>{Direction}</Direction>
         <Level>{Level}</Level>
         <Identifier>{Identifier}</Identifier>
         <Type>Task</Type>
         <Operations>
            <Operation>
               <Name>{A}</Name>
               <!-- Mahdolliset parametrit -->
            </Operation>
            <Operation>
               <Name>{B}</Name>
               <!-- Mahdolliset parametrit -->
            </Operation>
         </Operations>
         <ExecutionControl>{Value}</ExecutionControl>
      </Packet>
   </Items>
</Packet>
```

6.7. Toimintojen määrän optimointi

Jotta suorituksessa voitaisiin välttyä turhalta redundanssilta, tulee olla mahdollista karsia sellaisia toimintoja, joiden suorittaminen ei ole lopputuloksen kannalta välttämätöntä (kuva 6.18). Toimintojen karsinta edellyttää kuitenkin, että ohitettavien toimintojen välillä on selkeä suhde. Tällainen suhde voidaan toteuttaa esimerkiksi antamalla toiminnoille tunnisteet, joiden perusteella niiden välinen yhteys voidaan tunnistaa. Tunnisteiden pitää liittää toisiinsa sellaiset alustus- ja purkutoiminnot, jotka liittyvät selkeästi samaan asiaan. Yleensä tällaiset parit sijaitsevat yhden tehtävän sisällä. Vain tunnisteelliset toiminnot yritetään ohittaa optimoimalla.



Kuva 6.18: Suoritettavien toimintojen optimointi

Toimintojen minimointi perustuu siihen, että askeltaja pitää kirjaa asioista, jotka on alustettu. Askeltaja siis sisältää tunnistelistan, johon lisätään uusia tunnisteita aina sellaisen kohdattaessa. Jos listalta löytyy jo jokin tunniste, ei sellaisella varustettua alustustoimintoa ole tarpeen suorittaa enää uudestaan. Listalta tunniste poistetaan vasta, kun sitä vastaava purkutoiminto suoritetaan. Alustustoimintojen minimointi onnistuu näin varsin helposti, mutta purkutoimintojen optimointi vaatii hiukan monimutkaisempaa toiminnallisuutta.

Purkutoiminnoissa pelkkien tallennettujen tunnisteiden tarkastelu ei riitä, vaan on tarpeen selata sekvenssiä eteenpäin ja tutkia minkälaisia alustus- tai purkutoimintoja on tulossa. Tätä varten sekvenssi tallentaa suoritustilansa, ja palauttaa sen ennalleen, kun tulevat toiminnot on käyty läpi. Tiloja selataan sekvenssin loppuun asti, joka saattaa pitkien sekvenssien tapauksessa aiheuttaa kuormitusta järjestelmälle. Eteenpäin selattaessa askeltaja seuraa, mitä tunnisteita saavutetaan. Jos sekvenstä löydetään vielä jokin samalla tunnisteella varustettu toiminto, voidaan tarkastelun kohteena oleva toiminto määritellä toisteiseksi (*redundant*) ja siten ohittaa.

Tunnisteita määriteltäessä tulee kiinnittää huomiota siihen, että purku- ja alustustoiminnot on loogista esittää aina pareittain. Tunnisteparilla tarkoitetaan yhtä alustus- ja yhtä purkutoimintoa, joille on määritetty sama tunniste. Parillisella tunnisteella on siis olemassa vastineensa. Yksittäisessä sekvenssissä voi olla useita samalla tunnisteella varustettuja pareja ja tällaisen sekvenssin suoritusta optimointialgoritmi pyrkii tehostamaan.

Optimointialgoritmi ei kuitenkaan toimi väärin tilanteessa, jossa sekvenssi sisäl-

tää parittomia tunnisteita. Jos optimoija törmää tilanteeseen jossa alustus–purkutoimintoparia seuraa pariton alustustoiminto, suoritetaan näistä kolmesta toiminnosta vain ensimmäinen. Tällöin lopputulos on sama kuin parillisten tunnisteiden tapauksessa. Parittoman purkutilan tapauksessa taas optimointi jopa korjaa sekvenssin toimintaa — tällöin lopputulos on sama kuin parillisten tunnisteiden tapauksessa.

Tehtävien lisäyskään ei vaikuta optimointialgoritmin toimintaan. Ainoastaan sekvenssin viimeisessä tehtävässä suoritettu uuden tehtävän lisäys saattaa jättää osan ideaalisista optimoinneista tekemättä. Näitä optimointeja ei kuitenkaan ole mahdollista suorittaa käytettävissä olevien tietojen perusteella, koska optimoidessa ei voida ennakoida minkälaisia tehtäviä listan perään mahdollisesti lisätään. Näiden optimointitoimenpiteiden ohittaminen ei aiheuta loogisesti virheellistä toimintaa.

Optimointi toteutetaan osaksi `SequenceIterator`-luokkaa. Se olisi teknisesti mahdollista sisällyttää myös `Sequence`-luokkaan, mutta on luokkien vastuualueiden kannalta loogisempaa laittaa se osaksi iteraattorin toteutusta. `SequenceIterator` vastaa suorituslogiikasta kokonaisen sekvenssin laajuudessa, jolloin on järkevää liittää myös tilojen minimointi siihen.

7. ARVIOINTI

Tässä luvussa pohditaan askeltajan vaikutusta järjestelmään, esitetään muutamia jatkokehitysideoita sekä arvioidaan työtä kokonaisuutena.

7.1. Askeltajan yleiset vaikutukset

Askeltaja vaikuttaa järjestelmään laajemmin kuin pelkästään paikallisessa mittakaavassa. Kaikki vaikutukset eivät kuitenkaan aiheudu suoraan askeltajasta, vaan ne ovat useamman eri ominaisuuden yhteistoiminnan ansiota — joskin askeltaja on kyseisen ominaisuuden mahdollistaja.

Ohjelmiston suorituslogiikkaan askeltajalla on suuri vaikutus. Ohjelmistorungosta eriytetty sovellus, joka ei sisällä askeltajaa, pystyy toimimaan itsenäisesti vain hyvin rajoittuneesti. Tällaisessa tilanteessa tuoterungolla on jatkuvasti kontrolli ohjelman suoritukseen, mutta sille ei ole mahdollista määrittää mitä toimintoja sen tulisi suorittaa. Askeltaja lisää tuoterunkoon mahdollisuuden automatisoida asioita sekä luoda erilaisia ketjuja yksittäisistä toiminnoista. Tällöin askeltaja on tuoterungon ytimen sijaan se komponentti, joka ohjaa järjestelmän suoritusta.

7.1.1. Rinnakkaisuuden hallinta

Rinnakkaisuuden hallintaa askeltaja helpottaa huomattavasti. Jokainen sekvenssi voidaan ajatella omaksi säikeekseen, jolloin yksittäisen säikeen hallinta onnistuu toimintokokonaisuuden tarkkuudella. Näiden suoritussäikeiden välillä ei myöskään tarvita niin vahvaa synkronointia ja poissulkemista kuin perinteisen monisäikeisen sovelluksen tapauksessa. Askeltajakomponentin ei tarvitse kiinnittää erityistä huomiota monisäikeistyksen haasteisiin, koska ne on siirretty suoritettavien toimintojen vastuulle. Jos vastaava järjestelmä toteutettaisiin perinteiseen tyyliin ilman hajautusta ja viestinvälitystä, muodostuisi monisäikeistyksen hallinnasta huomattava työtaakka. Myös järjestelmän ylläpito on rinnakkaisuuden abstrahoinnin myötä huomattavasti helpompaa kuin yksiprosessisessa järjestelmässä.

7.1.2. Joustavuuden lisäys

Askeltaja lisää koko järjestelmän joustavuutta monella eri tapaa. Se ei välttämättä edes tiedä, mitä toimintoja sen tulee suorittaa, koska kaikkien suoritettavien tehtävien nimet kerrotaan sille erikseen. Edes tehtävälistan saatuaan ei askeltaja voi olla varma, tuleeko listan loppuun vielä lisää uusia tehtäviä. Suorituspolut ovat siis erittäin dynaamisia, ja ne voivat olla saman syötteen peräkkäisillä ajokerroilla erilaisia. Tämä tuo käyttäjän kannalta epätoivottavaa epädeterministisyyttä sekvenssin

suoritukselle, mutta se voidaan minimoida hyvällä dokumentoinnilla. Suorituspolut voidaan myös määritellä hyvin korkealla tasolla käyttäen useita eri esitystapoja. Tehtäviä voidaan teoriassa noutaa niin tietokannasta, levyltä kuin viestiväylänkin kautta toisilta komponenteilta.

Sekvenssien nimeämiskäytännöllä voidaan parantaa huomattavasti askeltajan hallittavuutta. Jokerimerkkien avulla voidaan käskyttää useita eri sekvenssejä yhdellä viestillä, ja näin esimerkiksi tietyn sekvenssiryhmän askellus on helppoa. Askeltajan viestiväyläliityntä tuo myös useita muitakin etuja. Vaikka askeltajalla ei ole mitään omaa hallintakäyttöliittymää, voidaan sellainen helposti toteuttaa ulkopuolisena komponenttina, joka keskustelee askeltajan kanssa viestiväylän avulla. Tästä on enää pieni askel siirtyä komponenttiin, joka osaa myös osittain automatisoidusti antaa askeltajalle komentoja. Ajastetut toiminnot ovat yksi esimerkki toiminnallisuudesta, joka on helppo toteuttaa osaksi ulkopuolista komponenttia. Askeltajan laajentaminen on siis yksinkertaista, kunhan se vain tarjoaa riittävän rajapinnan muiden komponenttien käyttöön.

Kun tarkastellaan SequenceControlin ja Sequencen käyttämiä viestejä, havaitaan, että kokonaisista sekvensseistä voidaan tehdä rekursiivisia. Askeltaja voi siis osana jonkin sekvenssin suoritusta käynnistää kokonaan uusia sekvenssejä, jotka näkyvät alkuperäiselle sekvenssille yksittäisen tehtävän tavoin. Tämä mahdollistaa askeltajan hallitsemisen itsensä avulla.

7.1.3. Vikasietoisuuden kasvaminen

Askeltajan käyttö yhdessä tuoterungon kanssa mahdollistaa mielenkiintoisia mekanismeja vikasietoisuuden kasvattamiseksi. Koska askeltaja ei välitä siitä, mikä komponentti hajautetussa järjestelmässä toiminnon lopulta suorittaa, on tätä mahdollista hyödyntää redundanssin lisäämiseksi. Komponenttien keskinäisellä sopimuksella voi jokin toinen verkon solmu ottaa vastuun toiminnon suorittamisesta kontolleen, jos alkuperäinen komponentti on jostain syystä estynyt suorittamaan sitä. Tällainen muistuttaa jaettuja kirjastoja, jossa toteutusta voidaan vaihtaa täysin pääohjelmasta riippumatta yksinkertaisesti vaihtamalla kirjastotiedosto toiseen, esimerkiksi tietylle prosessoriarkkitehtuurille optimoituun versioon. Käyttäjät voivat muokata sekvenssin toimintaa myös kesken sen suorituksen vaihtamalla toimintoja suorittavia komponentteja toisiin.

Toimintojen suorituksessa voidaan hyödyntää myös kuorman tasaisesta jakamisesta (*load balancing*) tuttua mekanismia, jossa yksi välityspalvelimena (*proxy*) toimiva komponentti jakaa askeltajalta saamiaan työpyyntöjä ryhmälle sen alaisuudessa olevia toisia komponentteja. Sen lisäksi, että suorituskyky paranee, myös vikasietoisuus kasvaa.

Tuoterunko tarjoaa lisäksi laiteriippumattomuuden, jolloin yksi sekvenssi voi olla

suorituksessa monessa eri fyysisessä laitteessa. Myös askeltajan hallintaan voidaan osallistua fyysisesti eri laitteilta, jolloin ei helposti muodostu tilannetta, jossa askeltajan toimintaa ei mikään komponentti pääsisi ohjaamaan. Komponenttien hajautus voi olla myös haitta. Erityisesti huonojen tietoliikenneyhteyksien vaikutus korostuu viestinvälityksessä, mikä voi vaikuttaa sekvenssien suoritukseen negatiivisesti.

7.1.4. Tehokkuus

Ajonaikaisella optimoinnilla on iso vaikutus järjestelmän suorituskykyyn niin paikallisessa kuin globaalissakin mittakaavassa. Paikallisesti optimointi aiheuttaa kuorman nousua, mutta globaalisti se pienentää koko järjestelmän kuormaa — varsinkin viestiväylän osalta. Viestiväylän alla piilevä fyysinen siirtotie on järjestelmän tulevissa käyttökohteissa todennäköinen pullonkaula, jolloin tällainen vaihtokauppa on perusteltua.

Ajonaikainen optimointi itsessään ei ole kovinkaan yleinen mekanismi perinteisiä ohjelmistoja käsiteltäessä. Käännettävät ohjelmointikielet eivät tätä toteuta, mutta tulkattavissa kielissä on tällainen optimointi hyvinkin tavanomaista. Esimerkiksi Java suorittaa optimointia ajonaikaiseen profilointitietoon perustuen. Ajonaikainen optimointi siis vaatii riittävän korkean tason ohjelmointikielen tai välikerroksen, joka mahdollistaa suorituksen hallinnan. Tämän tuoterungon tapauksessa välikerros mahdollistaa suorituksen hallinnan, jota taas hyödynnetään optimoinnin toteutuksessa.

7.2. Vaikutukset arkkitehtuuriin

Arkkitehtuurin tasolla askeltaja toimii tuoterungon mallia vastaan. Tuoterunko noudattaa kerrosarkkitehtuuria, mutta viestiväylän ja askeltajan yhteistoiminnan ansiosta kaikki järjestelmään liittyneet komponentit ovat vertaisiaan. Tällöin myös tietovuot eri komponenttien välillä kulkevat kerrosarkkitehtuurin vastaisesti, vaikka kerrosarkkitehtuuria käytettäessä olisi tarkoituksenmukaista yrittää rajoittaa tiedon liikesuunta vain eri kerrosten välille. Tämä saattaa vaikeuttaa järjestelmän toiminnan hahmottamista, mutta se mahdollistaa erittäin joustavan toimintaympäristön.

Tässä työssä esitetty järjestelmä on viestinvälityksen ansiosta tapahtumapohjainen. Kukin komponentti reagoi saamiinsa viesteihin, eivätkä ne siis suorita toimintojaan täysin itsenäisesti. Askeltaja tuo kuitenkin järjestelmään mahdollisuuden suorittaa komponenttien toimintoja proseduraalisesti, jolloin myös tapahtumapohjaisessa ohjelmistossa voidaan luoda toiminnoista loogisia suoritusketjuja.

7.3. Jatkokehitys

Askeltajaa voidaan kehittää edelleen usealla eri osa-alueella. Varsinkin toiminnan tehostamisella voi olla tulevaisuudessa suuri merkitys, jos askeltajan kuormitus kasvaa. Ensimmäinen näistä on toimintojen minimointialgoritmi, jonka suorittaminen tulee suurten sekvenssien tapauksessa viemään suurimman osan askeltajan prosessointiajasta. Tätä voitaisiin tehostaa ajamalla optimointialgoritmi vain, kun sekvenssiin tulee lisää uusia tehtäviä. Optimointidata voitaisiin tallentaa jonkinlaiseen tietorakenteeseen (esimerkiksi listaan), jota käytäisiin läpi samalla, kun varsinaista sekvenssiä suoritetaan.

Algoritmia voidaan tehostaa myös muuttamalla sen toiminta koskemaan vain ylimmän tason tehtäviä tehtävälistassa. Tämän lisäksi on mahdollista myös asettaa rajoite, jossa optimoitavien tunnisteiden täytyy olla samalla tasolla. Tällöin optimointialgoritmi olisi erittäin nopea — joustavuuden kustannuksella. Laskentakuorma on kuitenkin mahdollista saada useita kertaluokkia pienemmäksi alkuperäiseen toteutukseen verrattuna.

Toinen toimintojen optimointiin liittyvä kehitysmahdollisuus on niiden optimointi globaalissa laajuudessa eli myös eri sekvenssien välillä. Jos tulevaisuudessa huomataan tarve suorittaa paljon samoilla tunnisteilla varustettuja toimintoja myös rinnakkaisissa sekvensseissä, saattaa tällaisen toteuttaminen tulla kyseeseen. Se edellyttää kuitenkin huomattavia muutoksia rajapintoihin sekä olioiden vastuualueiden määrityksiin.

Askeltajan käyttöä voidaan helposti laajentaa lisäämällä sen käyttöön uusia tietokantoja. Tietokantaliityntä ei välttämättä ole käytettävissä kaikilla alustoilla tai sen käyttö on epävarmaa, jolloin esimerkiksi XML-muotoinen varasto paikallisella levyllä voisi toimia korvaajana.

8. YHTEENVETO

Diplomityössä pohdittiin erilaisia toteutusmekanismeja askeltavalle suorittamiselle ja todettiin hajautuksen helpottavan tällaisen järjestelmän kehittämistä huomattavasti. Monitorointi- ja hallintakomponentin yhdistäminen osoittautui luonnolliseksi ratkaisuksi, josta ei ole negatiivisia sivuvaikutuksia järjestelmälle. Huomattiin myös että viestiväylän olemassaolo ja järjestelmän asynkronisuus voidaan helposti valjastaa askeltajan käyttöön.

Askeltajakomponentti tuo useita etuja järjestelmälle. Se lisää huomattavasti koko järjestelmän joustavuutta, hallittavuutta sekä jäljitettävyyttä. Myös deterministisyyden lisääntyminen asynkronisessa viestinvälitysjärjestelmässä on toivottua. Debuggereita voidaan käyttää helposti vertailukohtana, koska niiden käyttötarkoitus on jokseenkin samanlainen. Askeltaja mahdollistaa niihin verrattuna huomattavasti monipuolisemmat hallinnointityökalut eri säikeiden käsittelyyn, lisäksi sen avulla voidaan myös jäljittää kuhunkin tilaan johtaneet tapahtumaketjut.

Monipuolisuudella on kuitenkin hintansa. Askeltajan suorituskykyyn vaikuttavat hajautuksen vuoksi myös ohjelmistosta riippumattomat seikat. Esimerkiksi verkkoviive ja solmujen katoaminen verkosta aiheuttavat sen, ettei askeltaja pärjää suorituskyvyssä perinteisille hajauttamattomille sovelluksille. Lisäksi askeltaja toimii arkkitehtuuritasolla järjestelmän yleistä arkkitehtuuria vastaan, koska askeltaja ohittaa kaikki riippuvuussuhteet suoritettavien komponenttien välillä.

Askeltaja on kuitenkin sellainen komponentti, jonka hyödyt ovat melko kiistattomat. Sen haittavaikutukset ovat pienet hyötyihin verrattuna. Dynaamisuus on askeltajan valttikortti, joka on suurelta osin viestiväylän ja hajautuksen ansiota.

Työtä kokonaisuutena arvioitaessa onnistui se mielestäni varsin hyvin, vaikkakin ohjelmistoalalle tyypillisesti aikataulu hieman venyi suunnitellusta. Aika ei tietenkään riittänyt kaiken täydelliseen suunnitteluun, vaan joitain osuuksia jouduttiin jättämään keskeneräisiksi tai jättämään pois tämän diplomityön sisällöstä. Olen kuitenkin pääosin tyytyväinen askeltajan arkkitehtuuriin, mutta tiettyihin komponentteihin olisi voinut käyttää suuremman osan käytettävissä olevasta ajasta.

Prototyypin ohjelmoiminen kirjoitustyön ohessa pienensi huomattavasti huolta suunnitelmien toimivuudesta, koska uudet ideat oli helppo testata käytännössä ennen asian viemistä eteenpäin. Prototyyppi myös paljasti muutamia ongelmakohtia askeltajassa, joista iteraattoritoteutukset ovat esimerkkinä — teknisesti elegantti suunnitelma muuttuu helposti vaikealukuiseksi koodiksi.

VIITTEET

[1] The ADAPTIVE Communication Environment, *Overview of ACE*, `http://www.cs.wustl.edu/~schmidt/ACE-overview.html`, Viitattu 1.7.2009.

[2] Boost Library Documentation *Class barrier*, `http://www.boost.org/doc/libs/1_33_1/doc/html/barrier.html`, Viitattu 15.6.2009.

[3] National Transportation Safety Board, *Cockpit Voice Recorders (CVR) and Flight Data Recorders (FDR)*, `http://www.ntsb.gov/aviation/CVR_FDR.htm`, Viitattu 4.6.2009.

[4] Java 2 Reference, *Cyclic Barrier*, `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/CyclicBarrier.html`, Viitattu 12.6.2009.

[5] Lasse Ylinen, *Datapalvelin*, Diplomityö, Tampereen Teknillinen Yliopisto, 2009.

[6] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[7] Python Library Reference, *Event Objects*, `http://www.python.org/doc/2.5.2/lib/event-objects.html`, Viitattu 10.6.2009.

[8] Jack G. Ganssle, *The Great Watchdogs*, versio 1.2, 2004, Saatavilla `http://www.ganssle.com/watchdogs.pdf`, Haettu 5.6.2009.

[9] Jonathan B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*, 1996, John Wiley & Sons, ISBN 0-471-14966-7.

[10] ISO/IEC 14882:2003, *International Standard: C++*, Second edition, 2003-10-15.

[11] Ilkka Haikala, Hannu-Matti Järvinen, *Käyttöjärjestelmät*, Talentum, 2004.

[12] Antti Viidanoja, *Mesh-verkon reitittimen toteutus*, Diplomityö, Tampereen Teknillinen Yliopisto, 2009.

[13] Daniel Dvorak, Benjamin Kuipers, *Model-Based Monitoring of Dynamic Systems*, 1989, Saatavilla `ftp://ftp.cs.utexas.edu/pub/qsim/papers/Dvorak+Kuipers-ijcai-89.ps`, Haettu 4.6.2009.

[14] Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore, Jim Brandy, *Non-stop Multi-Threaded Debugging in GDB*, Saatavilla `http://www.codesourcery.com/publications/non_stop_multi_threaded_debugging_in_gdb.pdf`, Haettu 9.6.2009.

[15] Kai Koskimies, Tommi Mikkonen, *Ohjelmistoarkkitehtuurit*, Talentum, 2005.

[16] Patria Oyj, *Yrityksen sisäinen muistio*.

[17] Blaise Barney, *POSIX Threads Programming*, `https://computing.llnl.gov/tutorials/pthreads/`, Viitattu 10.6.2009.

[18] Baris Simsek, *Signals*, 2005, Saatavilla `http://www.enderunix.org/simsek/articles/signals.pdf`, Haettu 9.6.2009.

[19] Martin McCarthy, *Thread-Specific Data and Signal Handling in Multi-Threaded Applications*, `http://www.linuxjournal.com/article/2121`, Viitattu 9.6.2009.

[20] Timo Kuosmanen, *Tuoterunko hajautetussa ympäristössä*, Tietotekniikan pro gradu -tutkielma, Jyväskylän yliopisto, 2007.

LIITE 1 TAPAHTUMAOBJEKTI

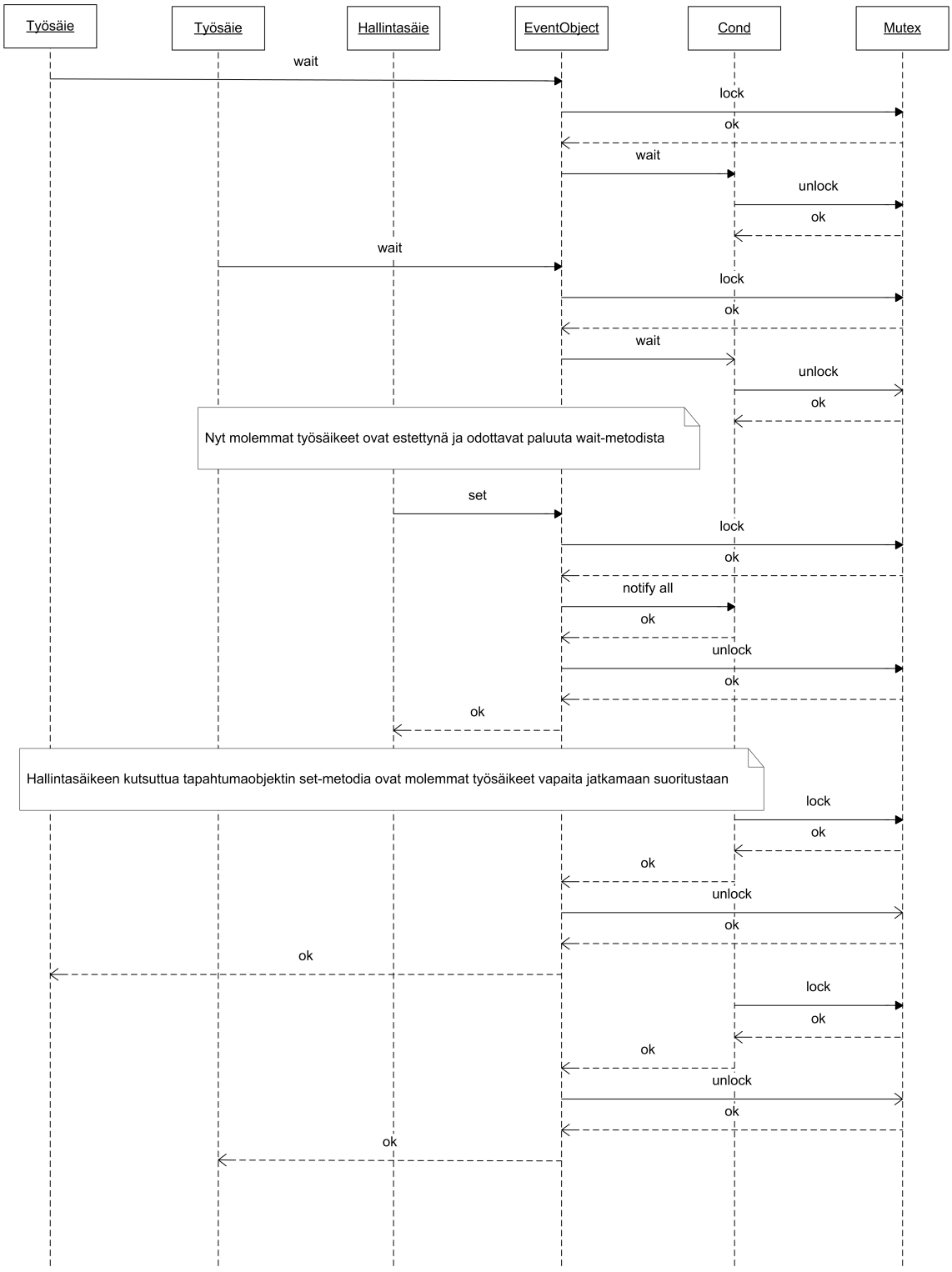
```
class EventObject {
    wait() {
        m.lock()
        while (!value) {
            // wait vapauttaa ja lukitsee mutex-objektin automaattisesti
            c.wait(m)
        }
        m.unlock()
    }

    set() {
        m.lock()
        value = true
        c.notifyAll()
        m.unlock()
    }

    clear() {
        m.lock()
        value = false
        m.unlock()
    }

    m = mutex()
    c = cond()
    value = false
}
```

LIITE 2 TAPAHTUMASEKVENSSIKAAVIO SÄIKEIDEN SYNKRONOINNISTA



LIITE 3 ASKELTAJA

```
#ifndef CONTROLLER_HH
#define CONTROLLER_HH

#include <pthread.h>

class Controller {
    public:
        Controller() : mutex(), cond(), value(false) {
            pthread_mutex_init(&mutex, 0);
            pthread_cond_init(&cond, 0);
        }

        ~Controller() {
            pthread_mutex_destroy(&mutex);
            pthread_cond_destroy(&cond);
        }

        void wait() {
            pthread_mutex_lock(&mutex);

            if (!value) {
                pthread_cond_wait(&cond, &mutex);
            }

            pthread_mutex_unlock(&mutex);
        }

        void go() {
            pthread_mutex_lock(&mutex);
            value = true;
            pthread_cond_broadcast(&cond);
            pthread_mutex_unlock(&mutex);
        }

        void step() {
            pthread_mutex_lock(&mutex);
            value = false;
```

```
            pthread_cond_broadcast(&cond);
            pthread_mutex_unlock(&mutex);
        }

        void stop() {
            pthread_mutex_lock(&mutex);
            value = false;
            pthread_mutex_unlock(&mutex);
        }

    private:
        pthread_mutex_t mutex;
        pthread_cond_t cond;
        bool value;
};

#endif
```

LIITE 4 SYKLINEN ESTE

```
#ifndef BARRIER_HH
#define BARRIER_HH

#include <pthread.h>

class Barrier {
    public:
        Barrier(unsigned int count_) : mutex(), cond(), threshold(count_),
                                       count(count_), generation(0) {
            pthread_mutex_init(&mutex, 0);
            pthread_cond_init(&cond, 0);
        }

        ~Barrier() {
            pthread_mutex_destroy(&mutex);
            pthread_cond_destroy(&cond);
        }

        void wait() {
            pthread_mutex_lock(&mutex);
            unsigned int gen = generation;

            if (--count == 0) {
                generation++;
                count = threshold;
                pthread_cond_broadcast(&cond);
                pthread_mutex_unlock(&mutex);
                return;
            }

            while (gen == generation) {
                pthread_cond_wait(&cond, &mutex);
                pthread_mutex_unlock(&mutex);
            }
        }

    private:
```

```
        pthread_mutex_t mutex;
        pthread_cond_t cond;
        unsigned int const threshold;
        unsigned int count;
        unsigned int generation;
};

#endif
```

LIITE 5 PROTOTYYPPI

```
#include <iostream>
#include <string>
#include "controller.hh"
#include "barrier.hh"

unsigned int const NUM_THREADS = 4;

Controller controller;
Barrier barrier(NUM_THREADS);

void *ThreadFunc(void*) {
    int count = 0;

    while (true) {
        barrier.wait();
        controller.wait();
        std::cout << ++count << std::endl;
    }
}

int main() {
    pthread_t threads[NUM_THREADS];

    for (unsigned int i = 0; i < NUM_THREADS; ++i)
        pthread_create(&threads[i], 0, ThreadFunc, 0);

    std::string tmp;

    while (std::cin >> tmp) {
        if (tmp == "go")
            controller.go();
        else if (tmp == "step")
            controller.step();
        else if (tmp == "stop")
            controller.stop();
        else
            break;
```

```
        }
    }
```