TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ARTO MÄKINEN

AKTIOJÄRJESTELMÄ MONIYDINSUORITTIMELLE

Diplomityö

Tarkastaja: Prof. Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekuntaneuvoston
kokouksessa 09.12.2015

TIIVISTELMÄ

ARTO MÄKINEN: Aktiojärjestelmä moniydinsuorittimelle
Tampereen teknillinen yliopisto
Diplomityö, 44 sivua, 4 liitesivua
lokakuu 2016
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastajat: Prof. Hannu-Matti Järvinen
Avainsanat: aktioparadigma, aktio, objekti, rinnakkaisuus, käyttöjärjestelmä

Aktioparadigma kuvaa ohjelmien suorituksen aktioiden ja objektien avulla. Objektit kuvaavat järjestelmän dataa ja aktiot sisältävät suoritettavan koodin. Jokainen aktio käsittelee objektien osajoukkoa. Jos aktiot eivät käsittele samoja objekteja, ne voidaan suorittaa vapaassa järjestyksessä. Tämän ansiosta aktiot voidaan suorittaa myös samanaikaisesti. Paradigman mukainen järjestelmä koostuu vuorontajasta, sovellussuorittimista sekä aktio- ja objektisäiliöistä. Vuorontaja valitsee suoritettavat aktiot ja huolehtii, että samoja objekteja käsittelevät aktiot eivät päädy suoritukseen samanaikaisesti. Sovellussuorittimet suorittavat aktiot.

Aktiojärjestelmässä rinnakkaisohjelmointi on siten helpompaa, koska poissulkemisesta ei tarvitse huolehtia. Myöskään erillistä viestinvälitystä ei tarvita objektien toimiessa jaettuina datasäiliöinä. Aktioilla on vahti, jonka avulla voidaan lisäksi toteuttaa aktioiden välinen synkronointi helposti. Vahti määrittää, missä järjestelmän tilassa aktio voi tulla suoritukseen. Vahdin tila perustuu järjestelmän objektien tilaan, joista erityisesti tarkkaillaan vahdin omistavan aktion käsittelemiä objekteja.

Lopputuloksena toteutettu järjestelmä toimii pohjana tulevalle kehitykselle. Se myös mahdollistaa aktioparadigman esittelyn ja levittämisen, koska kuka tahansa voi ottaa sen kokeiltavakseen. Tällä tavoin voidaan tehdä uudenlaista, hajautettua ja yhteisöllistä tutkimusta.

ABSTRACT

ARTO MÄKINEN: Action system for a multicore processor
Tampere University of Technology
Master of Science thesis, 44 pages, 4 Appendix pages
October 2016
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Prof. Hannu-Matti Järvinen
Keywords: action paradigm, action, object, concurrency, operating system

Action paradigm describes computer program execution using actions and objects. Objects hold data and actions contain the executed code. Actions handle a subset of objects. Actions that handle distinct objects can be executed in any order. This allows concurrent execution of actions. A system following this paradigm consists of action scheduler, application processors, and action and object containers. Action scheduler chooses actions for execution and makes sure that only actions with distinct objects are executed simultaneously. Application processors do the actual execution of actions.

Since the system takes care of mutual exclusion, concurrent programming becomes easier. Actions also have a guard, that can be used for synchronation conditions. The action guard tells when the action can be executed. The guards state is based on the objects in the system, mostly the ones its action uses.

The system implemented can be used as a basis for future developement and further testing. It allows demonstrating the action paradigm. Because anyone can experiment with it, due to the hardware and software used, new forms of community based distributed research can be conducted.

ALKUSANAT

Tämä on Tampereen teknilliselle yliopistolle tehty diplomityö. Työn aihe on professori Hannu-Matti Järviseltä, joka toimi myös työn tarkastajana. Työn oikolukemisessa on auttanut Birgit Mäkinen.

Tampere, 18.09.2016

Arto Mäkinen

SISÄLLYS

LYHENTEET JA MERKINNÄT

Linux	Avoimen lähdekoodin käyttöjärjestelmä.
PID	Process Identifier. Linuxin prosesseille antama yksikäsitteinen tunniste.
FIFO	First in first out. Jonon toimintaperiaate, jossa alkiot käsitellään samassa järjestyksessä kuin ne on lisätty.
nimi	Tiedostopolut ja lähdekoodista suoraan tulevat nimet merkitään kursiivilla.

1. JOHDANTO

Rinnakkaisuuden lisääntyessä tietotekniikassa täytyy etsiä ratkaisuja sen ongelmien ratkaisemiseksi. Lukkiutuminen, poissulkeminen ja nälkiintyminen ovat hankalia ongelmia. Niiden ratkaisemiseksi voidaan soveltaa aktioparadigmaa. Aktioparadigma soveltuu hyvin myös reaktiivisiin järjestelmiin, joissa järjestelmän tarkoitus on reagoida ulkoisiin tapahtumiin ennalta määritellyllä tavalla.

Aktioparadigman avulla voidaan korvata prosessipohjainen suoritusmalli, joka nykyisin on käytössä. Säikeet ja prosessit korvataan aktioilla ja data tallennetaan objekteihin. Aktiot ovat atomisia suorituskokonaisuuksia. Ne voidaan suorittaa toisistaan riippumatta, jos ne eivät käsittele samoja objekteja.

Tässä diplomityössä esitellään aktioparadigma ja sen mukaisesti toteutettu järjestelmä eli aktiosuoritin. Työn tavoitteena on kehittää testijärjestelmä tutkimusta ja aktioparadigman levitystä varten. Samalla pyritään luomaan järjestelmän osien väliset rajapinnat, joiden avulla aktiosuorittimesta voidaan kehittää oma laitteistopohjainen toteutus.

Työssä kuvattu aktiosuoritin toimii jo olemassa olevalla laitteistolla. Moniydinsuorittimen avulla järjestelmästä saadaan oikeasti rinnakkainen, jolloin aktioparadigman hyödyt tulevat paremmin esille. Laitteistoksi valittiin hyvin saatavilla oleva ja edullinen RaspberryPi 2 malli B. Sen käyttö mahdollistaa järjestelmän levittämisen helposti, koska sitä käytetään erityisesti kaikenlaisiin kokeiluihin.

Helpolla levittämisellä pyritään saamaan paradigmasta laajempi kokemuspohja. Muilta kehittäjiltä ja asiasta kiinnostuneilta voidaan saada arvokkaita ideoita sekä uusia näkökulmia aktioparadigmaan. Näiden avulla paradigman haasteita voidaan yrittää ratkoa helpommin.

Aktioparadigma kuvataan tarkemmin luvussa 2. Sen jälkeen esitellään toteutetun järjestelmän toiminnot. Lopuksi käydään tarkemmin läpi järjestelmän tekninen to-

teutus. Toteutuksen yhteydessä myös kuvataan kehityksessä käytetyt kirjastot, teknologiat ja muut valmiit ratkaisut.

2. AKTIOPARADIGMA

Tässä luvussa kuvataan työn pohjana oleva aktioparadigma ja sen päälle rakentuva aktiojärjestelmä teoriatasolla. Teoreettinen kuvaus toimii pohjana luvulle 3, jossa kuvataan toteutettava järjestelmä. Tässä luvussa käydään läpi myös teoreettiset rajoitteet ja vaatimukset järjestelmälle.

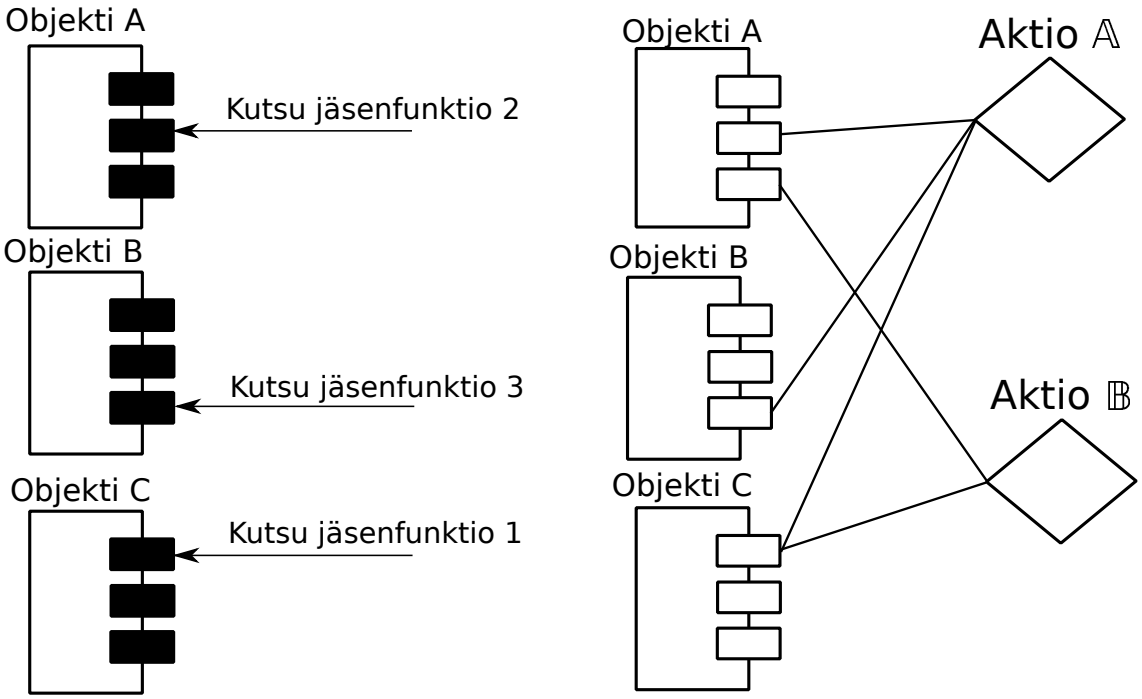
Aktioparadigma on vaihtoehtoinen sovellusten mallinnustapa. Se eroaa nykyisin eniten käytetystä prosesseihin ja säikeisiin perustuvasta mallista sovellusten jaottelun ja suoritustavan suhteen. Aktioparadigmassa sovellus koostuu aktioista ja objekteista. Aktiot sisältävät varsinaisen suoritettavan koodin. Objektit puolestaan toimivat datan säilytyspaikkoina. Aktiot käsittelevät objektien tietoja.

2.1 Aktiot ja objektit

Aktiolla on joukko objekteja, joihin se viittaa suorituksensa aikana. Näitä objekteja kutsutaan aktion osallistujiksi. Sama objekti voi olla osallistujana useassa aktiossa, jolloin aktiot voivat kommunikoida sen välityksellä. Jokainen objekti on kuitenkin vain kerran saman aktioin osallistujana. Aktio ei välttämättä muuta kaikkien osallistujiensa tilaa. [3]

Objektien lisäksi aktioilla on vahti, joka rajoittaa sen suoritusta. Aktion vahti voidaan jakaa paikalliseen, yhteiseen ja globaaliin osaan. Paikallinen viittaa yhteen osallistujaan kerrallaan, yhteinen mihin tahansa aktion osallistujaan ja globaali mihin tahansa järjestelmän objektiin. Globaalia osaa vahdista ei toteuteta, sillä sen toteutus on monimutkaista ja sen tuomat hyödyt vähäisiä. [3]. Yksi globaalin vahdin ongelmista on, että objektit pitäisi kaikki lukita sen suorituksen ajaksi.

Vahti on totuusarvoinen lauseke, jonka on oltava tosi ennen kuin siihen liittyvä aktio voidaan suorittaa. Sen tehtävä on estää aktion suoritus, jos aktion osallistujat eivät ole aktiolle sopivassa tilassa. Vahdit voivat toimia aktioiden välisinä synkro-



Kuva 2.1 *Vasemmalla oliopohjaisen ja oikealla aktiopohjaisen sovelluksen rakenne.[3]*

nointiehtoina. [3] Tällöin aktioiden vahdit määritellään niin, että ne suoritetaan aina halutussa järjestyksessä. Synkronoitavilla aktioilla on oltava yhteisiä osallistujia, jotta synkronointi toimisi.

Aktiot voidaan ajatella mahdollisina tilasiirtyminä. Näin ajateltuna objektit kuvaavat järjestelmän tilan. Vahti taas määrittää missä tiloissa aktion määrittämä tilasiirymä on mahdollinen. Koska aktiot vastaavat tilasiirtymiä, niillä ei ole sisäistä muistia. Kaikki tieto on tallennettu objekteihin. [3]

Kuvassa 2.1 on esitetty sovelluksen rakenne verrattuna oliopohjaiseen sovellukseen. Oliopohjaisen sovelluksen mustatut suorakaiteet viittaavat olioiden jäsenfunktioiden toteutuksiin ja toteutuksen piilottamiseen. Valkoiset suorakaiteet kuvaavat varsinaisen toteutuksen sijaitsevan aktiossa. Objektit voivat olla olioita, jolloin aktio toimii jäsenfunktioiden kutsujana. Aktiosovelluksen kutsut on mallinnettu viivoina korostamaan ulkoisen kutsujan puutetta. Aktio on itse suoritettava yksikkö. [3]

2.2 Aktioiden temporaalilogiikka

Teoriatasolla aktioparadigma voidaan määritellä käyttäen aktioiden temporaalilogiikkaa. Aktioiden temporaalilogiikka perustuu predikaattilogiikkaan lisäten siihen operaattorin aina, sekä aktiot ja niiden käsittelyn. Sen on alun perin esitellyt Leslie Lamport [6]. Tämä aliluku perustuu Lamportin [6] ja Kurki-Suonion teksteihin [5].

Temporaalilogiikka on predikaattilogiikan laajennos. Siihen on lisätty operaattori $\Box$ eli aina. Sen avulla voidaan ilmaista, että lause E pätee kaikille käyttäytymisen σ tiloille eli $\sigma[[\Box E]]$. Lisäksi operaattori $\Diamond$ eli lopulta voidaan määritellä $\Diamond \triangleq \neg\Box\neg E$.

Suorituksen teoreettista kuvausta kutsutaan käyttäytymiseksi väärinkäsitysten välttämiseksi. Käyttäytyminen σ on järjestetty, numeroituvasti ääretön joukko tiloja $\sigma = \langle s_0, s_1, s_2, ...\rangle$. Kaavassa $s_i \in \Sigma$, jossa Σ on kaikkien tilojen joukko.

Äärelliset käytännön suoritukset kuvataan käyttäytymisinä, joissa lopputila toistuu loputtomasti. Käyttäytymisen ensimmäinen tila s_0 on sen alkutila. Käyttäytymisiin voidaan soveltaa temporaalilogiikan kaavoja. Merkintä $\sigma[[P]]$ tarkoittaa, että käyttäytyminen σ toteuttaa lauseen P.

Aktiot kuvataan käyttäen joukkoja V ja V' kuvaamaan muuttujia ennen sijoitusta ja sen jälkeen. Siis muuttuja x' kuvaa muuttujaa x aktiossa tapahtuneen sijoituksen jälkeen. Muuttujat eivät ole varsinaisesti tyypitettyjä, mutta voivat saada arvoja kokonaislukujen, merkkijonojen, totuusarvojen sekä merkkien joukosta.

Aktio on siis totuusarvoinen lauseke. Sen avulla mille tahansa tilaparille (s, t) voidaan saada totuusarvo $s[[A]]t$. Jos se on tosi, tilat s ja t tyydyttävät aktion A. Lausekkeelle saadaan arvo, kun aktion lausekkeeseen sijoitetaan pilkullisiin muuttujiin arvot tilasta t ja pilkuttomiiin muuttujiin arvot tilasta s.

Näiden avulla voidaan määritellä järjestelmälle elävyysominaisuudet heikko ja vahva reiluus. Epämuodollisesti määriteltynä elävyysominaisuus kertoo, että jotain toivottua tapahtuu lopulta. Niiden lisäksi järjestelmällä on turvaominaisuuksia, jotka estävät epätoivotut tapahtumat. Kaava 2.1 on heikon reiluuden kaava ja kaava 2.2 vahvan reiluuden.

$$WF(A) \triangleq \Diamond\Box Enabled A^{+} \Rightarrow \Box\Diamond\langle A^{+}\rangle \tag{2.1}$$

$$SF(A) \triangleq \Box\Diamond EnabledA^+ \Rightarrow \Box\Diamond\langle A^+\rangle \vee \Box\Diamond A^0 \tag{2.2}$$

Kaavoissa $\langle A\rangle$ tarkoittaa aktion A suoritusta. Merkintä $EnabledA$ tarkoittaa aktion A olevan vireessä. Merkintä A^+ on aktion osa, joka muuttaa tarkasteltavia muuttujia. A^0 taas on aktion osa, joka ei muuta tarkasteltavia muuttujia. Osat ovat toisensa poissulkevia, joten toisen ollessa vireessä toinen ei ole. Kumman tahansa osan toteutuminen lasketaan aktion A suoritukseksi.

Molemmat ehdot vaativat aktion tulevan suoritetuksi, jos se on vireessä. Erona on, että vahva reiluus vaatii aktion suorituksen aktion aktivoituessa äärettömän usein. Heikkolle reiluudelle riittää aktion suoritus sen jäädessä vireeseen.

Käyttäytymiset ovat sarjallisia, mutta aktioiden temporaalilogiikassa ei ole operaattoreita seuraava ja edellinen. Tästä syystä käyttäytymisiä, joissa aktioiden järjestystä on vaihdettu, ei voi erottaa toisistaan. Aktioiden järjestyksellä on merkitystä vain niiden käsitellessä samoja objekteja. [3]

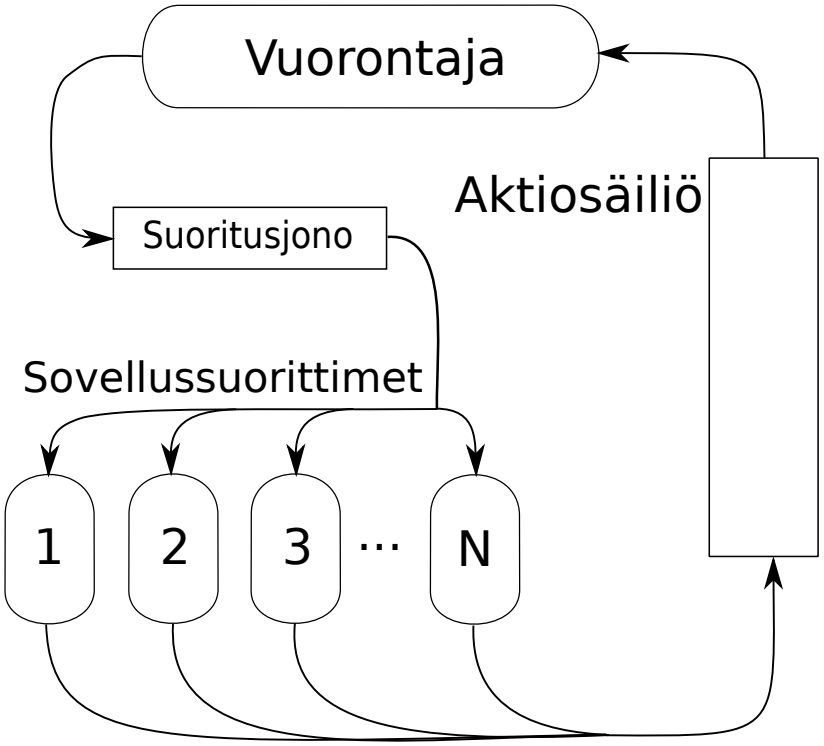
Aktiot, jotka kuuluvat erillisten aktioiden joukkoon **S**, voidaan suorittaa samanaikaisesti. Joukko **S** toteuttaa kaavan 2.3. Kaavassa V_{A_a} on joukko muuttujia, joihin aktio A sijoittaa ja V_{A_r} on joukko muuttujia, joihin aktio A vain viittaa. Kaavaa voidaan soveltaa myös objekteille. [3]

$$\forall A, B \in \mathbf{S}, A \neq B : V_{A_a} \cap V_{B_a} = \emptyset \wedge V_{A_a} \cap V_{B_r} = \emptyset \wedge V_{B_a} \cap V_{A_r} = \emptyset \tag{2.3}$$

2.3 Aktiojärjestelmä

Aktiojärjestelmä eli aktiosuoritin on aktioparadigman mukaisesti toteutettu järjestelmä. Järjestelmän tulee toteuttaa kaava 2.2 kaikkien aktioiden joukolle. Jokaiselle yksittäiselle aktiolle reiluutta ei pystytä täysin takaamaan. Järjestelmässä rinnakkaisuus tapahtuu siten, että kaava 2.3 toteutuu. Järjestelmä koostuu aktioista, objekteista, vuorontajasta ja sovellussuorittimista.

Aktio on suorituksen perusyksikkö. Aktion käyttämät objektit eli osallistujat lukitaan aktion käyttöön, kun se valitaan suoritukseen. Siten jokaista objektia käsittelee vain yksi aktio kerrallaan. Aktiot suoritetaan siis käytännössä atomisina operaatioina, mikä erottaa ne prosessipohjaisen mallin säikeestä tai prosessista. Tästä syystä



Kuva 2.2 *Aktion kulku järjestelmässä. Perustuu lähteeseen [3]*

aktioiden ei tarvitse huolehtia poissulkemisista.

Suoritukseen valinnan ja objektien lukitsemisen tekee vuorontaja. Se on osa järjestelmän ydintä. Aktio on vireessä, kun sen vahti on tosi. Vuorontaja valitsee suoritettavan aktion vireessä olevien aktioiden joukosta huomioiden niiden osallistujat. Koska muut aktiot voivat muuttaa objektien tilaa ja siten aktioiden vahtien tiloja, ei voida taata aktion pääsevän suoritukseen aina vahdin ollessa tosi.

Varsinaisesta suorituksesta vastaavat sovellussuorittimet. Ne sekä suorittavat aktiot että laskevat vahdit. Yhtä aikaa suorituksessa olevien aktioiden määrä riippuu käytössä olevien sovellussuorittimien määrästä [3]. Sovellussuoritin suorittaa yhtä aktiota kerrallaan. Käytännön aktiosuorittimissa sovellussuorittimien määrällä on jokin laitteiston asettama yläraja, mutta teoriassa rajoitusta ei ole.

Aktion kulku järjestelmässä on kuvattu kuvassa 2.2. Siinä on kuvattujen osien lisäksi suoritusjono, jossa suoritukseen valitut aktiot odottavat sovellussuorittimen vapautumista. Jos vuorontajalla ja sovellussuorittimilla ei ole yhteistä muistia, täytyy aktioita ja objekteja kopioida muistien välillä. [3]

2.4 Paradigman hyödyt ja haitat

Aktioparadigma on houkutteleva vaihtoehto prosessipohjaiselle suoritusmallille varsinkin rinnakkaisuuden lisääntyessä. R. Kurki-Suonio on esittänyt aktioparadigman käyttöä reaktiivisten järjestelmien määrittelyssä [5]. Aktioparadigmassakin on omat ongelmansa ja haasteensa, jotka on huomioitava.

Kuvatulla tavalla toimivassa järjestelmässä ohjelmoijan ei tarvitse huolehtia poissulkemisista ja kriittisten alueiden käsitettä ei tarvita. Myös aktiotason lukkiutumiset ovat mahdottomia. Viestinvälityskin tulee tarpeettomaksi.[3]

Aktioiden vahdit toimivat synkronointiehtoina. Tällöin kaikki rinnakkaisen suorituksen ongelmat ovat järjestelmän harteilla, eikä ohjelmoijan tarvitse huolehtia niistä. Aktioiden koodia ei tarvitse muuttaa, vaikka sovellussuorittimien määrä järjestelmässä muuttuisi. [3]

Koska sovellussuorittimien määrä ei vaikuta aktioiden toimintaan, niitä voidaan poistaa käytöstä tai ottaa käyttöön suorituksen aikana. Näin voidaan säästää virtaa, kun kaikkien suorittimien ei tarvitse olla päällä järjestelmän kuorman ollessa matala. [3]

Suurimpana ongelmana aktioparadigmassa on, että ihmiset eivät ole kovin hyviä ajattelemaan sen mukaisesti. Ihmisillä on taipumus etsiä syy-seuraussuhteita jopa paikoista, joissa niitä ei oikeasti ole [4]. Toisena ongelmana ovat tähänastiset tutkimukset. Ne ovat keskittyneet sovellusten määrittelyyn ja pohjautuvat edelleen prosessipohjaiseen suoritusympäristöön. [3]

Vaikka aktiosuoritin ratkaisee useita rinnakkaisuusongelmia, siinäkin esiintyy nälkiintymistä ja lukkiutumia. Nälkiintymiseen voidaan vaikuttaa hyvällä vuoronnusalgoritmilla. Lukkiutumat ovat tarkoituksellisia tai korkeamman tason virheitä. Tarkoituksellisella lukkiutumalla saadaan esimerkiksi suoritus loppumaan.[3]

3. JÄRJESTELMÄN KUVAUS

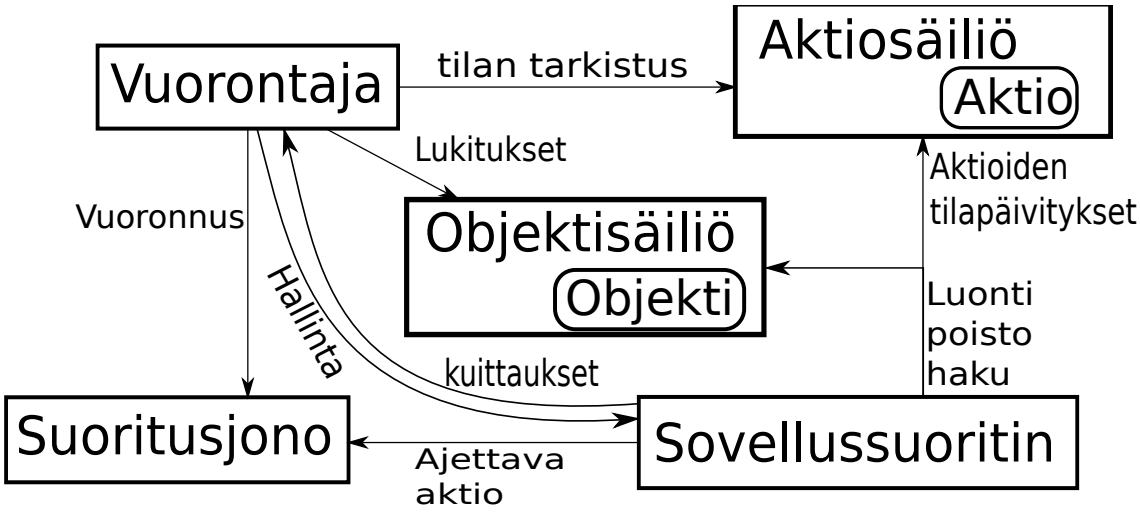
Tässä luvussa esitellään järjestelmän eri osat ja niiden toiminnot. Ensin käydään läpi aktiot ja objektit, sitten sovellussuorittimet ja lopuksi vuorontaja. Sovellussuorittimista ja vuorontajasta kuvataan erityisesti niiden rajapinnat toimintotasolla. Varsinainen toteutus käydään läpi luvussa 4.

Toteutuksen helpottamiseksi ja nopeuttamiseksi käytetään järjestelmän alustana avoimen lähdekoodin Linux-käyttöjärjestelmää. Avoin lähdekoodi soveltuu hyvin tutkimuskäyttöön ja mahdollistaa alustan muokkaamisen tarpeen mukaan. Käytetty ympäristö on kuvattu tarkemmin luvussa 4.1.

Järjestelmä koostuu vuorontajasta ja sovellussuorittimista. Niiden välillä, osana rajapintaa, toimivat aktiosäiliö ja suoritusjono. Aktiosäiliön kautta vuorontaja voi tarkistaa aktiiviset aktiot ja selvittää lukittavat objektit. Suoritusjonosta sovellussuorittimet saavat suoritettavat aktiot. Toteutettu järjestelmä mukailee rakenteeltaan ja toiminnoiltaan professori Hannu-Matti Järvisen kuvaamaa[3] järjestelmää.

Toimintojen jako sovellussuorittimien ja vuorontajan kesken perustuu osittain siihen, onko toiminto toteutettu Linuxin ytimen osana vai sen ulkopuolella. Kaikki Linuxin ulkopuolella toteutettu toiminnallisuus kuuluu sovellussuorittimille. Linuxin osana toteutettu toiminnallisuus on kuvattu osana vuorontajaa, mutta kaikki toiminnot eivät suoranaisesti liity vuorontajan tehtäviin. Luvussa 5 on käyty läpi vuorontajalle kuulumattomat toiminnot.

Kuvassa 3.1 esitellään järjestelmän rakenne. Siitä voidaan myös nähdä osien tärkeimmät vuorovaikutukset. Aktiot saavat osallistujansa sovellussuorittimen kautta, joten ne eivät käsittele objektisäiliötä suoraan. Sovellussuorittimet kuittaavat aktioiden ja vahtien suoritukset sekä omat tilanmuutoksensa vuorontajalle. Näiden kuittausten avulla vuorontaja voi helposti pitää kirjaa järjestelmän kokonaistilasta. Vuorovaikutusten kuvaukset ovat seuraavissa luvuissa ja tekninen toteutus luvussa 4. Kuvan nuolet osoittavat toiminnon aloittajasta kohteeseen. Sovellussuorittimet



Kuva 3.1 *Järjestelmän rakenne*

eivät tee objekteille tilapäivityksiä, koska se on vuorontajan vastuulla.

3.1 Aktiot ja objektit

Aktiot ja objektit ovat järjestelmän perusosia. Sovellusten kaikki suoritettava koodi on aktioissa ja data objekteissa. Aktioiden ja objektien luonti, poistaminen ja kopionti tehdään sovellussuorittimien kautta. Tässä järjestelmässä objekti on aina liitettävä aktioon luonnin yhteydessä. Aktiolla ei välttämättä tarvitse olla osallistujia, mutta silloin se ei myöskään voi kommunikoida muiden aktioiden kanssa. Sellaisella aktiolla ei myöskään voi olla paikallisia vahteja ja yhteinen vahti voi olla vain vakioarvoinen. Tälläinen aktio on käytännössä aina vireessä ja voidaan ottaa suoritukseen koska tahansa.

Aktiolla on tässä järjestelmässä tunniste, vahtifunktiot, osallistujat, parametrit sekä tila. Aktiot voidaan laittaa suoritusjonoon paikallisten vahtien ollessa tosia. Yhteinen vahti suoritetaan vasta, kun aktio on otettu suoritukseen. Näin varmistutaan vahdin tilan oikeellisuudesta, sillä aktion osallistujat ovat vain sen itsensä käytössä. Aktion runko suoritetaan kuitenkin vasta koko vahdin ollessa tosi. Aktion parametrit pysyvät vakioina aktion olemassaolon ajan. Niiden avulla voidaan säätää aktion käyttäytymistä.

Aktioiden vahdit on jaettu paikallisiin ja yhteisiin suorituskyvyn parantamiseksi. Jokaiseen aktion osallistujaan voi liittyä paikallinen vahti. Kaikkiin osallistujiin ei

kuitenkaan tarvitse liittyä paikallista vahtia. Yhteinenkään vahti ei ole pakollinen. Jos aktiolla ei ole lainkaan vahteja, se on aina vireessä. Tällöin ainoastaan jaetut objektit rajoittavat sen suoritusmahdollisuuksia.

Aktioiden vahteihin voi liittyä ajastuksia. Ajastukset viivästävät aktion suoritusta vahdin tullessa todeksi. Niiden avulla voidaan rajoittaa aktion suoritustiheyttä. Jos johonkin vahtiin liittyy ajastus, aktio otetaan suoritukseen aikaisintaan ajastuksen päätyttyä. Koko vahdin on edelleen oltava tosi ja aktion osallistujien vapaita, jotta aktio suoritetaan.

Objektit ovat järjestelmässä yksinkertaisia. Niillä on tunniste, luokka sekä varsinainen data. Tunnisteen avulla objektit erotetaan toisistaan ja varmistetaan oikeat lukitukset. Luokka pitää sisällään objektin tyypin, joka kuvaa kaikille saman luokan objekteille yhteiset asiat. Näihin kuuluvat objektin datan koko, luokan nimi sekä alustus-, purku- ja kopiointifunktiot. Funktioiden avulla järjestelmä voi alustaa objektin datan haluttuun tilaan ja varmistaa, että kaikki objektiin liittyvä data vapautetaan. Objektin data sisältää varsinaisen sovellusta kiinnostavan tiedon.

Objektit voivat sisältää mitä tahansa dataa, paitsi toisia objekteja. Ilman tätä rajoitusta järjestelmän olisi vaikea varmistua objektien lukituksista. Tämä ei juurikaan rajoita käytettävissä olevan datan määrää, sillä aktiolla voi olla useita suuriakin objekteja käytössään. Näin tarvittavat tiedot voidaan ryhmitellä käyttötarpeiden mukaan.

Aktion ja objektin välinen osallistujasuhde voi olla joko vain luku -tyyppinen tai luku ja kirjoitus -tyyppinen. Vain luku -tyyppisessä suhteessa aktio ei muuta objektin arvoa, mutta voi lukea sen. Tämä mahdollistaa toiminnan tehostamisen, sillä vain luku objekteihin liittyviä paikallisia vahteja ei tarvitse tarkastaa uudelleen aktion suorituksen jälkeen. Aktion tulee ilmoittaa järjestelmälle, mitä objekteja se on muuttanut. Kaikki muuttuneisiin objekteihin liittyvät vahdit tarkistetaan, vaikka vahdit eivät liittyisi suoritettuun aktioon. Näin aktioiden tilat päivittyvät, eikä niiden vahteja tarvitse erikseen ottaa laskettaviksi. Yhteinen vahti tarkistetaan vasta juuri ennen rungon suoritusta, jotta mikään toinen aktio ei pääse muuttamaan viitattuja objekteja vahdin suorituksen aikana. Paikalliset vahdit voidaan tarkistaa, koska ne viittaavat vain yhteen objektiin ja se on suorituksessa olevan aktion käytössä eli lukittuna.

Tästä toimintatavasta johtuen aktio merkitään vireessä olevaksi, kun sen paikalliset

vahdit ovat tosia. Siten sen yhteinen vahti saadaan tarkistettavaksi ja runko suoritettavaksi, jos yhteinenkin vahti on tosi. Tällaistä aktiota kutsutaan aktiiviseksi. Aktiivisten aktioiden yhteisen vahdin tilaa ei aina tiedetä, jolloin ne on otettava jossain vaiheessa suoritukseen koko vahdin tilan selvittämiseksi.

3.2 Sovellussuoritin

Sovellussuoritin toimii abstraktiona aktioiden ja muun järjestelmän välillä. Se myös suorittaa aktiot ja niiden vahdit. Kaikki aktioiden kommunikointi järjestelmän kanssa kulkee sovellussuorittimien aktioille tarjoaman rajapinnan kautta. Keskittämällä kaikki yhteen rajapintaan järjestelmän käytöstä tulee helpompaa.

Muulle järjestelmälle tarjottu rajapinta on vuorontajalle ja käyttäjille. Osa toiminnoista kuuluu osittain sovellussuorittimille ja osittain vuorontajalle. Näiden ominaisuuksien kuvaukset on jaettu tämän luvun ja luvun 3.3 kesken. Vastaava jako toistuu toteutuksen kuvauksessa luvussa 4.

Jokaisella järjestelmän sovellussuorittimella on käytössään yksi laitteiston suoritinydin. Näiltä ytimiltä estetään muun ohjelmakoodin ajaminen kokonaan, jotta niiden suorituskyky on täysin sovellussuorittimien käytössä. Yksi suoritinydin jätetään vuorontajalle ja käytetyn Linux-järjestelmän prosesseille. Muita prosesseja tarvitaan järjestelmän käyttämiseen, koska järjestelmälle ei ole toteutettu käyttöliittymää tai monia muita sovelluksia.

3.2.1 Aktioille tarjottu rajapinta

Aktioille tarjotaan rajapinta, joka mahdollistaa järjestelmän erikoisominaisuuksien käyttämisen. Perustoimintoina tarjotaan aktioiden ja objektien luonti, poisto sekä kopiointi. Vahteihin liittyvät ajastukset kuuluvat myös rajapintaan. Laitteistotuen testausta varten tarjotaan syötteen pyytäminen käyttäjältä. Rajapinta on pyritty pitämään yksinkertaisena ja toimintojen päällekkäisyyksiä on pyritty välttämään.

Aktioiden ja objektien lisäykset ja poistot välitetään aktio- ja objektisäiliöille. Ne hoitavat varsinaisen poistamisen. Aktioita lisätessä on varmistettava, että aktio ei päädy suoritukseen ennen kuin sen osallistujat ovat kaikki olemassa. Vastaavasti, kun aktio poistaa osallistujansa, aktio itse on poistettava. Näin on toimittava, koska aktiota ei voida suorittaa, jos jokin sen osallistujista ei ole olemassa.

Objekteissa käytetään viitelaskentaa, joka estää objektin lopullisen poistamisen aktioiden viitatessa siihen. Kun viimeinen objektiin viittaava aktio poistetaan tai se poistaa objektin, objekti poistetaan lopullisesti. Samalla objektin datalle kutsutaan objektin luokan purkufunktiota. Objektien luokkia ei poisteta automaattisesti.

Vahteihin liittyvät ajastukset rekisteröidään sekä sovellussuorittimille että myöhemmin vuorontajalle. Sovellussuoritin huolehtii ajastuksiin liittyvästä laskennasta ja kertoo vuorontajalle, milloin aktio voidaan ottaa seuraavan kerran suoritukseen. Vuorontaja varmistaa, että aktio ei tule suoritukseen liian aikaisin.

Lisäksi aktioille tarjotaan järjestelmän tilan luku. Tähän sisältyvät seuraavaksi suoritukseen tuleva aktio sekä aktioiden ja objektien määrä järjestelmässä. Vaikka järjestelmä pystyy ilmoittamaan, mikä aktio ajetaan seuraavaksi, suoritusajankohdasta ei voida sanoa mitään. Se saattaa jopa tulla suoritukseen ennen kuin kysyvän aktion suoritus loppuu. Tieto haetaan vuorontajan suoritusjonosta.

Kehityksen helpottamiseksi tarjotaan yhteiseen lokiin kirjoittaminen. Lokin avulla voidaan selvittää ongelmia ja arvioida järjestelmän käyttäytymistä. Voidaan esimerkiksi arvioida, kuinka tasaisesti suorituskuorma jakautuu eri sovellussuorittimille. Lokiin kirjoittaminen toteutetaan siten, että mahdollisimman paljon tietoa päätyisi lokiin myös virhetilanteissa. Kuitenkin tarkoituksena on tarjota mahdollisuus informatiivisten lokiviestien kirjaamiseen. Lokiin kirjataan lisäksi järjestelmän omia tilaviestejä.

Aktiot voivat pyytää käyttäjältä syötettä siihen tarkoitetun objektin avulla. Objekti rekisteröidään aktiolle ja järjestelmän syötejonoon. Syötejonosta objekteihin tuodaan syötettä samassa järjestyksessä kuin ne on rekisteröity. Jotta käyttäjä tietää, mitä häneltä halutaan, aktio voi antaa käyttäjälle näytettävän viestin. Syöteobjektiin ja sen käsittelevään aktioon liitetään järjestelmän tarjoama vahtifunktio, joka varmistaa syötteen saapumisen ennen kuin aktio aktivoidaan. Syöteobjektien määrää yhdellä aktiolla ei ole erityisesti rajoitettu.

Syötteen käsittelevä aktio ei eroa muista aktioista muuten kuin syöteobjektin osalta. Syöteobjekti on aktion kannalta samanlainen kuin muutkin sen osallistujat. Sovellus ei kuitenkaan voi liittää syöteobjektiin paikallista vahtia, koska järjestelmä käyttää omaa vahtiansa. Yhteinen vahti voi käsitellä syöteobjektia, mutta sen tilan ei pitäisi riippua syötteestä. Jos yhteisen vahdin tila riippuu syötteestä, aktio ei välttämättä pääse pyytämään uutta syötettä. Syöteobjektin vahtiin voi liittyä ajastus.

3.2.2 Muulle järjestelmälle tarjottu rajapinta

Muulle järjestelmälle tarjottava rajapinta koostuu kahdesta osasta. Ensimmäinen on vuorontajalle tarjottu rajapinta, joka sisältää sovellussuorittimien hallintaan liittyvät toiminnot. Toinen osa on käyttäjille ja kehittäjille tarjottu rajapinta, joka mahdollistaa järjestelmän tilan seuraamisen. Se toimii yhdessä vuorontajan luvussa 3.3 kuvatun käyttäjille tarjotun rajapinnan kanssa.

Vuorontajalle tarjotaan sovellussuorittimen käynnistäminen uudelleen. Näin vuorontaja voi alustaa uudelleen järjestelmän ja keskeyttää huonosti toimivan aktion suorituksen. Jokainen sovellussuoritin on tässä suhteessa itsenäinen, joten järjestelmän uudeleen alustamisessa ne on käynnistettävä uudelleen erikseen.

Käyttäjille tarjotaan aktiosovellusten käynnistäminen, kaikille sovellussuorittimille yhteinen loki sekä tiedon syöttäminen syötettä odottaville aktioille. Lisäksi annetaan mahdollisuus seurata järjestelmän käyttäytymistä suorituksen aikana. Näiden avulla sovelluskehitys on helpompaa.

Sovellussuorittimien loki sisältää aktioiden lokiviestit ja sovellussuorittimien omat viestit. Jokaisessa viestissä on liitettynä viestin kirjoittaneen sovellussuorittimen tunniste, jotta viestit voidaan kohdistaa järkevästi. Sovellussuorittimien omat viestit merkitään niin, että ne eivät sekoitu aktioiden viesteihin. Merkinnän avulla lokista voidaan suodattaa järjestelmän viestit, kun ollaan kiinnostuneita aktioiden toiminnasta ja päinvastoin.

Käyttäjille tarjotut tiedot ovat aktioiden ja objektien lukumäärät sekä sovellussuorittimien tila. Sovellussuorittimilla on neljä tilaa, jotka ovat odottaa aktiota, yhteinen vahti, aktion runko ja paikallinen vahti. Tilat toistuvat suorituksen aikana luetellussa järjestyksessä, paitsi aktion vahdin ollessa epätosi. Silloin aktion runkoa ei voida suorittaa ja siten vastaava tila hypätään yli.

Paikalliset vahdit tarkistetaan rungon suorituksen jälkeen ennen aktion palauttamista vuorontajalle. Näin aktion tekemät muutokset voidaan huomioida ja samalla tarkistaa muiden aktioiden samoihin objekteihin liittyvät vahdit. Muiden aktioiden vahdit voidaan tarkistaa, koska objektit on lukittu suorituksessa olevalle aktiolle.

Jotta vuorontaja pystyy seuraamaan järjestelmän toimintaa, sovellussuorittimet tekevät kuittauksia suorittamistaan tilamuutoksista. Kuittausten avulla vuorontaja voi reagoida muutoksiin nopeasti, eikä sen tarvitse tarkkailla muutoksia itse. Vuo-

rontajan tarjoama kuittausrajapinta on kuvattu aliluvussa 3.3.1.

3.3 Vuorontaja

Tässä luvussa kuvataan vuorontajan toiminta ja sen tarjoamat rajapinnat. Vuorontaja tarjoaa kaksi rajapintaa, joista tärkein on sovellussuorittimille tarjottu rajapinta. Ongelmien selvityksen tueksi vuorontaja mahdollistaa lisäksi tilan lukemisen järjestelmän ulkopuolelta. Vuorontajan rajapinta on pyritty pitämään mahdollisimman yksinkertaisena ja kaikki tarpeettomiksi katsotut toiminnot on jätetty pois. Yksinkertaisuuteen pyrittiin, jotta vuorontajan ei tulevaisuudessakaan tarvitsisi toteuttaa suurta määrää toimintoja.

Vuorontajan tehtävänä on valita suoritukseen tulevat aktiot ja hallinnoida sovellussuorittimia. Aktioiden valinnassa on huomioitava aktioiden vahdit ja ajoitukset sekä varmistettava, että aktion osallistujat ovat vain sen itsensä käytössä. Vuorontaja myös osallistuu laitteistotuen toteutukseen. Kuvattujen rajoitusten ja tavoitteiden täyttämiseksi vuorontaja pitää kirjaa suorituksessa olevista aktioista, niiden osallistujista sekä aktioiden ajoituksista.

Vuorontajan suorituskyky on erityisen tärkeää, sillä siitä muodostuu helposti järjestelmän pullonkaula. Myös aktiosäiliön suorituskyky vaikuttaa tähän, sillä vuorontaja käsittelee aktiota sen kautta. Kuitenkin muistinkulutus täytyy samalla pitää alhaisena, jotta objektien ja aktioiden tallennukseen olisi riittävästi tilaa.

Vuoronnusalgoritmin täytyy pystyä toteuttamaan lukitukset sekä saada aktio suoritukseen mahdollisimman usein sen vahdin ollessa tosi. Toteutettavassa järjestelmässä kuitenkin pysytään yksinkertaisessa algoritmissa, jotta algoritmin toteutukseen ei kuluisi liikaa aikaa. Koska aktioparadigma ei vaadi tiettyä järjestystä toisistaan riippumattomille aktioille, voidaan vuoronnusalgoritmi valita hyvin vapaasti. Toteutetun järjestelmän käyttämä algoritmi on kuvattu aliluvussa 4.4.

Vuorontaja tarjoaa osansa järjestelmän tilan seurantarajapinnasta. Tähän rajapintaan kuuluvat vuorontajan loki, suoritusjonon ja sovellussuorittimien tilan seuranta sekä objektien lukituksen seuranta. Vuorontaja kirjoittaa eri lokiin kuin sovellussuorittimet ja aktiot, koska tämä on toteutuksen kannalta yksinkertaisinta.

3.3.1 Sovellussuorittimille tarjottu rajapinta

Sovellussuorittimille tarjotut toiminnot ovat pääosin kirjanpidollisia. Niiden avulla varmistetaan järjestelmän tilan säilyminen yhtenäisenä. Kirjanpidon päivityksen lisäksi ne voivat aiheuttaa vuoronnuksen suorituksen ja objektien lukitsemisen tai lukituksen avaamisen.

Sovellussuorittimet saavat suoritettavat aktiot vuorontajan suoritusjonosta. Aktion haku on käytännössä ainoa toiminto, joka ei ole tarkoitukseltaan ainoastaan kirjanpidollinen. Sitä käytetään myös sovellussuorittimien hallintaan. Haun yhteydessä sovellussuoritin voidaan keskeyttää, jos järjestelmässä ei ole suoritettavia aktioita. Suoritusjonon ollessa tyhjä suoritetaan vuoronnus, jotta voidaan varmistua suoritettavien aktioiden loppumisesta. Jos suoritettava aktio kuitenkin löytyy, se välitetään sovellussuorittimelle. Suoritusjono toimii FIFO(engl. First in first out)-periaatteella, jolloin aktiot tulevat suoritukseen samassa järjestyksessä kuin ne on jonoon laitettu.

Aktioiden loppukuittaus on kirjanpidon kannalta tärkeä toiminto. Sen avulla sovellussuoritin kertoo olevansa vapaa suorittamaan seuraavaa aktiota ja vuorontaja voi tarkistaa järjestelmän tilan. Loppukuittaus sisältää tiedon aktion uudesta tilasta. Samalla aktion osallistujien lukitus avataan, jolloin muut niitä käyttävät aktiot voivat päästä suoritukseen.

Loppukuittauksen yhteydessä vuorontajalle täytyy kertoa, jos aktio poistettiin. Poistosta ilmoittaminen varmistaa, että aktio on poistunut järjestelmästä. Lisäksi se kertoo vuorontajalle, että aktiota ei enää voi käsitellä. Loppukuittaus aiheuttaa vuoronnuksen, jos järjestelmässä on aktiivisia aktioita ja suoritusjonossa on tilaa. Objektien poistot aktioilta välitetään vuorontajalle, jotta niiden lukitukset voidaan avata.

Suoritusjonon seuraavan aktion selvitys tarjotaan, jotta sovellussuorittimet voivat kertoa sen aktioille. Tämä saattaa aiheuttaa vuoronnuksen, jos suoritusjono on tyhjä. Jos järjestelmässä ei ole suoritukseen kelpaavia aktioita, ilmoitetaan siitä kyselyn tehneelle sovellussuorittimelle. Näin voi käydä kahdessa tilanteessa. Ensimmäisessä vaihtoehdossa järjestelmässä ei ole aktiivisia aktioita, jotka eivät ole jo suorituksessa. Toisessa vaihtoehdossa aktiivisia aktioita on, mutta jokaiselta vähintään yksi osallistuja on lukittuna. Näissä tilanteissa ei voida tietää, mikä aktio seuraavaksi tulee suoritukseen.

Syötteen vastaanottamista varten vuorontajassa on mahdollista rekisteröidä aktio

odottamaan syötettä. Vuorontaja pitää kirjaa vastausta odottavista aktioista ja ilmoittaa sovellussuorittimille, kun vastaus on saatu. Vastauksen saavuttua voidaan merkitä käytetyn objektin paikallinen vahti todeksi ja tarkistaa aktion muut paikalliset vahdit. Siitä eteenpäin aktiota käsitellään samoin kuin muitakin aktioita.

4. JÄRJESTELMÄN TOTEUTUS

Tässä luvussa kuvataan järjestelmän tekninen toteutus. Ensimmäisenä käydään läpi kehityksessä käytetty laitteisto, tärkeimmät kirjastot ja kolmansien osapuolien sovellukset. Sen jälkeen esitellään luvussa 3 kuvattujen toimintojen toteutus.

Järjestelmän toteutus on jaettu vuorontajaan ja sovellussuorittimiin. Vuorontajan ohjelmakoodi on Linuxin osana, joten kaikki vuorontajan toiminnot suoritetaan ytimen tilassa. Suurin syy vuorontajan liittämiselle Linuxin osaksi on laitteiston käytön mahdollistaminen helposti. Sovellussuorittimet on toteutettu omana ohjelmanaan. Ne suoritetaan käyttäjätilassa. Suoritusjono on toteutettu myös Linuxin sisällä osana vuorontajaa. Aktio- ja objektisäiliöiden toteutus on osittain sovellussuorittimien osana ja osittain vuorontajan osana. Tämä jako johtuu käyttäjätilan ja ytimen tilan erottelusta Linux-järjestelmissä. Käyttäjän tilasta ei suoraan pääse käsiksi ytimen muistiin ja ytimen tilasta käyttäjän muistin käsittely on hidasta ja työlästä [7].

Järjestelmän tilan seurantaa varten on toteutettu neljä virtuaalitiedostoa. Ne toteutettiin osaksi Linuxin *proc*-virtuaalitiedostojärjestelmää. Virtuaalitiedosto on tiedosto, jonka sisältö luodaan muistinvaraisesti tietorakenteiden pohjalta. Virtuaalitiedostoon voidaan kirjoittaa, jos siihen on liitetty kirjoituksen vastaanottava funktio. Kirjoitettua dataa ei yleensä tallenneta, vaan sen perusteella suoritetaan jokin toiminto. Virtuaalitiedostojärjestelmä on muistinvarainen tiedostojärjestelmä, joka sisältää vain virtuaalitiedostoja. [7] Tiedostot ja niiden toteutus on kuvattu aliluvussa 4.4.

4.1 Käytetty ympäristö

Järjestelmä on kirjoitettu C-kielellä Linux-käyttöjärjestelmän päälle. Kääntäjänä on käytetty GNU Compiler Collectionin C-kääntäjää. Sovellussuorittimet eristetään omille suoritinytimilleen käyttäen Linuxin *cpusets*-ominaisuutta. Sen avulla on mahdollista määritellä suoritinydinjoukkoja, joille voidaan määritellä ajettavat prosessit.

Testilaitteistona toimii RaspberryPi 2 malli B. Se valittiin, koska siinä on neliydinsuoritin, se on edullinen ja hyvin saatavilla. Lisäksi sille on saatavilla hyvä dokumentaatio sekä valmiiksi räätälöity Linux-järjestelmä. Näin se mahdollistaa järjestelmän testauksen helposti. Samalla sen käytön toivotaan houkuttelevan muita tutustumaan toteutettuun järjestelmään ja aktioparadigmaan.

Aktiosäiliön toteutuksessa on käytetty valmista puna-mustan puun toteutusta. Myös C:n standardikirjastoa on käytetty kehityksessä. Vuorontajan toteutuksessa on käytetty Linuxin tarjoamia tietorakenteita ja ominaisuuksia.

Vaikka sovellussuorittimien kääntäminen ei vaadi vuorontajan tai edes Linuxin koodia, ne toimivat ainoastaan vuorontajan toteutuksen sisältävän Linux-ytimen päällä. Sovellussuorittimet tarvitsevat aliluvussa 4.4 kuvatut uudet järjestelmäkutsut toimiakseen.

4.2 Aktiot ja objektit

Tässä osiossa kuvataan aktioiden ja objektien rakenne järjestelmässä. Järjestelmän toteutuksen kannalta ne tomivat tietosäiliöinä. Aktion kautta järjestelmä pääsee käsiksi sen runkoon, osallistujiin sekä vahteihin. Aktion runko ja vahdit sisältävät varsinaisen suoritettavan ohjelman.

Sekä aktio että objekti ovat järjestelmässä tietueita. Niille on annettu positiiviset kokonaislukutunnisteet. Tunnisteiden avulla ne voidaan yksilöidä helposti. Aktioita ja objekteja luodaan, poistetaan ja kopioidaan aliluvussa 4.3 kuvatuilla funktioilla.

Objektin luokkaan on tallennettu alustus-, poisto- ja kopiointifunktiot sekä objektin koko. Luokan avulla järjestelmä voi automaattisesti varata muistia sekä suorittaa alustukset sopivassa kohdassa. Luokka myös helpottaa samanlaisten objektien luomista. Objektin tietuetyyppi on nimeltään *Object* ja luokan *ObjectClass*. Luokan funktioiden esittelyt löytyvät taulukosta 4.1. Mitään luokan funktioista ei ole pakko määritellä.

Erillistä objektisäiliötä ei toteutettu, vaan aktioilla on taulukko omista osallistujistaan. Taulukon koko on 32 alkiota, joka on osallistujien maksimimäärä. Määrä mahtuu yhteen lippumuuttujaan, jonka avulla voidaan tehokkaasti merkitä muutetut objektit. Lippumuuttujana käytetään kokonaislukua, jolle on annettu oma tyyppinimi *Mask*. Oma tyyppi selkeyttää muuttujan käyttötarkoitusta. Aktion vastuulla

Taulukko 4.1 *Objektin funktioesittelyt*

Funktio	**Kuvaus**
void Constructor(void* value)	rakentajafunktio
void CopyConstructor(void* old_value, void* new_value)	kopiorakentajafunktio
void Destructor(void* value)	purkajafunktio

on päivittää lippuja oikein. Osallistujien vahdit tarkistetaan vain lipun kertoessa niiden muuttuneen.

Vahdeille on omat tietuetyyppinsä, joissa on osoitin vahdin omistavaan aktioon, vahdin toteuttava totuusarvoinen funktio sekä vahdin tila. Lisäksi paikallisen vahdin tietueessa on tallennettuna sen käyttämä objekti. Vahdit voivat olla kolmessa tilassa, jotka ovat *TRUE*, *FALSE* ja *EVALUATING*. Vahdin ollessa tilassa *EVALUATING* sen arvoa ei tiedetä, jolloin vahti täytyy suorittaa. Tilat *TRUE* ja *FALSE* merkitsevät, että vahdin tiedetään olevan tosi tai epätosi. Näissä tiloissa olevaa vahtia ei tarvitse suorittaa. Aktion suorituksen jälkeen kaikki sen muuttamiin osallistujiin liittyvien vahtien tilat on päivitettävä. Aktion asettamien muutoslippujen avulla muuttumattomiin osallistujiin liittyviä vahteja ei tarvitse laskea uudelleen. Jos objektin arvo ei muutu, siihen liittyvien vahtienkaan arvo ei muutu.

Sekä objekteilla että aktioilla on yksisuuntaisesti linkitetty lista niihin liittyville paikallisille vahdeille. Linkitetty lista valittiin, koska se on yksinkertainen toteuttaa, se ei rajoita alkioiden määrää ja paikalliset vahdit täytyy useimmiten käydä kuitenkin kaikki läpi. Ainoastaan aktiota poistettaessa ja kopioitaessa etsitään tiettyä vahtia. Nämä listat viittaavat samoihin tietuemuuttujiin, jolloin niiden tilat ovat aina samat. Samalla säästyy muistia, kun samansisältöistä tietuetta ei tallenneta kahdesti.

Objektien vahtilistan avulla voidaan tarkistaa muidenkin kuin suorituksessa olevan aktion paikalliset vahdit objektin muuttuessa. Aktion vahtilistasta voidaan tarkistaa, pitääkö aktio aktivoida vai ei. Tässä hyödynnetään paikallisten vahtien tilan tallennusta. Aktion vahtilistaa läpikäydessä vain tarkistetaan vahdin tila tietueesta. Läpikäynnin tulos tallennetaan aktiotietueeseen. Vahtien varsinainen laskenta tehdään objektin vahtilistan läpikäynnin yhteydessä. Tällä tavoin paikallisten vahtien tarkistus on tehokasta ja kaikkien aktioiden vahdit saadaan tarkistettua.

Objektin dataan aktio pääsee käsiksi funktion *ap_object_data(Object* obj)* avulla. Se palauttaa osoittimen annetun objektin sisältämään dataan. Lokiviestejä varten

funktio *ap_ object_ id(Object* obj)* palauttaa annetun objektin tunnisteen. Funktioiden avulla saadaan selkeästi rajattua objektitietueiden käsittelyä. Ne myös mahdollistavat objektitietueen rakenteen piilottamisen aktioilta. Näin aktiot eivät helposti pääse sotkemaan objektien tietueita.

Aktion rungon toteuttava funktio saa parametreinaan aktion parametritaulukon, osallistujataulukon sekä osoittimen lippumuuttujaan. Aktion parametrit ovat etumerkillisiä kokonaislukuja, jotka säilyvät vakioina kutsujen välillä. Lippumuuttujaan aktion pitää kirjata, mitä osallistujia on muokattu suorituksen aikana. Taulukossa 4.2 ovat aktioiden ja vahtien funktioiden esittelyt. Toteutuksissa on käytettävä yhteensopivia funktioita.

Taulukko 4.2 *Aktioiden ja vahtien funktioesittelyt*

Funktio	**Kuvaus**
void ActionBody(int params[], Object* objs[], Mask* unchanged)	aktion runkofunktio
bool LocalGuardFun(Object* obj)	paikallisen vahdin funktio
bool CommonGuardFun(int params[], Object* objs[])	yhteisen vahdin funktio

Aktion rungolle annetaan suoraan aktiotietueeseen tallennettu taulukko sekä parametreista että objekteista. Muutoslippu on sovellussuorittimen paikallinen muuttuja, jonka kaikki bitit alustetaan arvoon 1 ennen aktion kutsumista. Muutoslippu on toteutettu käänteisenä, eli lippu on 1, kun kyseinen osallistuja ei ole muuttunut. Aktion alussa kaikki liput ovat ykkösiä.

Perustietojen lisäksi aktiolla on lista sen luomista aktioista. Listan avulla uudet aktiot voidaan aktivoida ne luoneen aktion loputtua. Näin aktio voi alustaa aktiot ja objektit ongelmitta. Tämä ratkaisu on toteutuksellisesti yksinkertainen ja sillä saadaan turvattua objektien käsittely.

Aktiot ryhmitellään järjestelmässä toisiinsa liittyviksi kokonaisuuksiksi, jota kutsutaan aktiosovellukseksi. Siihen kuuluu alustusaktio, joka luo kyseisen sovelluksen aloittavat aktiot ja objektit. Mikä tahansa aktio voi luoda uusia aktioita ja objekteja. Aktiosovellus käännetään dynaamisesti linkitettynä kirjastona, joka ladataan dynaamisesti sovellussuorittimien muistiin. Sovelluksien lataamiseksi on toteutettu *action_ loader*-komentorivisovellus, joka ottaa komentoriviparametrikseen ladattavan kirjaston nimen. Kirjaston vaatimuksena on, että se sisältää *init*-nimisen funktion.

Aktiosovelluksen aloittaa alustusaktio, jonka runkona käytetään sovelluksen toteut-

tavassa kirjastossa olevaa *init*-funktiota. Alustusaktiolla ei ole osallistujia tai parametreja ja se suoritetaan vain kerran. Alustusaktion tarkoituksena on luoda ja alustaa sovelluksen aloittamiseen tarvittavat aktiot ja objektit. Tarvittaessa aktioita ja objekteja voidaan luoda lisää myöhemmin aktivoitavissa aktioissa. Erillistä lopetusaktiota ei ole, vaan suoritus voidaan pysäyttää joko poistamalla kaikki aktiot tai tarkoituksellisella lukkiumalla.

Järjestelmän kannalta kaikkien aktioiden poistaminen on parempi, mutta joissain tapauksissa monimutkaistaa aktioiden toteutusta. Poistaminen on parempi, koska poistettaessa järjestelmään ei jää turhia aktioita. Järjestelmä ei pysty poistamaan tarkoituksellisesti lukittuja aktioita, koska tahallisen lukkiuman tunnistus on hankalaa. Aktiot voivat lukkiutua myös virheellisen toiminnan seurauksena, jolloin olisi hyvä saada mahdollisimman paljon tietoa tilanteesta.

4.3 Sovellussuoritin

Sovellussuorittimet toimivat järjestelmässä käyttäjätilan prosesseina. Niiden ohjelmakoodi sisältää aktioiden, objektien ja osan aktio- ja objektisäiliöiden ohjelmakoodista. Loput säiliöiden ohjelmakoodista on osana vuorontajaa. Vuorontaja ja sovellussuorittimet pitävät kirjaa osittain päällekkäisistä asioista. Näin on pyritty helpottamaan ja nopeuttamaan käyttäjätilan sovellussuorittimien ja ytimen tilan vuorontajan kommunikointia keskenään.

Sovellussuorittimien hallintaan käytetään signaaleja. Ne ovat prosessille lähetettäviä useimmiten numeromuotoisia viestejä [2]. Kun sovellussuoritin saa signaalin *SIGUSR1*, se käynnistyy uudelleen. Samalla se lopettaa suorituksessa olevan aktion suorituksen. Sovellussuoritin sammuttaa itsensä saadessaan järjestelmältä virhesignaalin, joita lähetetään esimerkiksi virheellisistä muistiviittauksista. Virheen tapahtuessa vuorontajalle annetaan tieto virheestä, joka kirjataan vuorontajan lokiin. Sen jälkeen käynnistetään uusi sovellussuoritin, mikä käytännössä vastaa sovellussuorittimen käynnistämistä uudelleen.

4.3.1 Aktioiden ja objektien käsittely

Seuraavaksi kuvatut aktioiden ja objektien käsittelyfunktiot voivat ottaa aktion tunnisteeksi luvun nolla, joka viittaa aina kutsun suorittaneeseen aktioon. Muita tunnisteita käytetään vain uusia aktioiden luonnin yhteydessä. Objektien tapauksessa

käytetään osoittimia objektitietueisiin. Objekteilla tunniste nolla on virheellinen.

Aktion luominen tapahtuu funktiolla *ap_add_action*, joka ottaa parametreikseen aktion rungon toteuttavan funktion, yhteisen vahdin, osallistujien lukumäärän, parametrien määrän ja parametrit. Parametrit välitetään taulukkona, josta ne kopioidaan aktion tietueeseen. Parametrien maksimimäärä on 32, kuten osallistujienkin. Funktio palauttaa luodun aktion tunnisteen, jotta sille voidaan lisätä osallistujia.

Sekä aktioiden että objektien tunnisteet tulevat globaaleista etumerkittömistä kokonailukumuuttujista. Uutta aktioita tai objektia luotaessa muuttujan arvo otetaan uudeksi tunnisteeksi ja muuttujan arvoa kasvatetaan yhdellä. Tässä on käytetty lukitusta, jotta tunnisteet olisivat yksikäsitteisiä. Lukittu alue on pyritty pitämään lyhyenä, jotta se ei heikentäisi järjestelmän suorituskykyä turhaan.

Osallistujat lisätään *ap_add_new_object* tai *ap_add_existing_object* funktiolla. Ensimmäinen luo uuden objektin ja lisää sen aktion osallistujiin. Funktio ottaa parametrikseen objektin luokan, aktion tunnisteen ja paikallisen vahdin. Jälkimmäinen taas lisää olemassaolevan objektin aktiolle ja ottaa parametrikseen objektin, aktion tunnisteen ja paikallisen vahdin. Lisäämällä sama objekti useammalle aktiolle saadaan aktiot kommunikoimaan keskenään. Jos aktiolla on jo täysi määrä osallistujia, uusia ei enää lisätä. Objektit välitetään aktiolle lisäysjärjestyksessä.

Objektin luokka luodaan funktiolla *ap_create_object_class*, jolle annetaan luokan nimi, koko, rakentajafunktio, purkajafunktio sekä kopiorakentajafunktio. Rakentajafunktioita kutsutaan aina luotaessa luokan objektia, myös kopioinnin yhteydessä ennen kopiorakentajaa. Kopiorakentajafunktiota kutsutaan objektia kopioitaessa ja purkajafunktiota objektia poistettaessa.

Jos jotain funktioista ei tarvita, funktiolle voidaan antaa vastaavaksi parametriksi *NULL*. Tällöin kyseistä funktiota ei kutsuta. Objektia luotaessa ensin luodaan objektitietue, johon tallennetaan objektin tunniste, luokka ja osoitin datalle varattavaan muistiin. Muistia varataan luokan koon verran ja se nollataan, jos rakentajafunktiota ei ole määritelty. Kopioinnin yhteydessä ensin luodaan uusi objekti vanhan luokalla ja näiden dataosoittimet välitetään kopiorakentajalle. Poistettaessa ensin kutsutaan luokan purkajafunktiota, jonka jälkeen datalle varattu muisti ja objektitietue vapautetaan.

Ohjelmassa 4.1 on esimerkki aktioiden ja objektien käsittelystä. Objektien luokka

luodaan ensimmäisenä, jotta se on saatavilla kaikille luotaville objekteille. Ensimmäisenä luotavalle aktiolle lisätään osallistujaksi uusi objekti, jolle asetetaan paikallinen vahti funktiolla *local_guard*. Seuraavalle aktiolle lisätään sama objekti osallistujaksi kuin ensimmäiselle. Esimerkistä on jätetty pois aktioiden runkofunktioiden ja paikallisen vahdin toteuttavan funktion esittelyt ja rungot.

Ohjelma 4.1 *Aktioiden ja objektien luonti*

```
1 ObjectClass* cls = ap_create_object_class("Luokka", sizeof(int),
2     NULL, NULL, NULL)
3
4 int ac_id = ap_add_action(body, NULL, 1, 0, NULL);
5 Object* obj = ap_add_new_object(cls, ac_id, local_guard);
6
7 ac_id = ap_add_action(body2, NULL, 1, 0, NULL)
8 ap_add_existing_object(obj, ac_id, NULL);
```

Osallistujan poistaminen aktiolta tapahtuu funktiolla *ap_remove_participant*, joka ottaa parametrikseen poistettavan osallistujan ja aktion tunnisteen. Kun osallistuja poistetaan aktiolta, kyseinen aktio merkitään poistettavaksi. Objektia ei poisteta ennen kuin kaikki siihen viittavat aktiot on poistettu. Kun viimeinen objektiin viittaava aktio poistetaan, objekti poistetaan ja sille suoritetaan luokan purkufunktio. Aktioiden poistaminen tehdään, kun se otetaan suoritukseen poistokäskyn jälkeen. Aktio ei voi poistaa muiden aktioiden objekteja tai muita aktioita. Tämä rajoitus on tehty varmistamaan, että jokin järjestelmän aktio ei pysty sotkemaan järjestelmää. Poikkeuksena on objektin lopullinen poistaminen, jolloin kaikki siihen viittaavat aktiot merkitään poistettaviksi.

Objektin lopullinen poistaminen tehdään funktiolla *ap_delete_object*. Se poistaa parametrina saamansa objektin ja kaikki siihen liittyvät aktiot järjestelmästä lopullisesti. Jos poiston seurauksena jonkin muun objektin viitelaskuri on nolla, sekin poistetaan. Kaikille poistettaville objekteille kutsutaan purkufunktiota ennen objektille varattujen muistialueiden vapauttamista.

4.3.2 Vahdit

Aktioiden paikalliset vahdit lasketaan aktion suorituksen jälkeen muuttuneille objekteille. Yhteinen vahti lasketaan ennen rungon suoritusta. Aktion runkoa ei kuitenkaan suoriteta ennen kuin koko vahti on suoritettu ja se on tosi. Yhteisessä vahdissa

käytetään tilan tallennusta, jolloin sitä ei tarvitse laskea uudelleen joka kerta.

Osallistujien muutoslippujen käsittelyyn tarjotaan funktio *ap_participant_changed*, joka ottaa parametrikseen objektin indeksin osallistujataulukossa ja osoittimen lippumuuttujaan. Funktio asettaa kyseistä osallistujaa vastaavan lipun nollaksi, jolloin siihen liittyvät paikalliset vahdit tarkistetaan. Funktion tarkoituksena on yhtenäistää lippujen käsittely ja yksinkertaistaa ohjelmointia. Jos aktio muuttaa kaikkia osallistujiaan, se voi asettaa lippumuuttujan suoraan nollaksi. Lippuja ei tarvitse erikseen asettaa, jos osallistujia ei muuteta.

Ohjelmassa 4.2 esitellään yksinkertaisen aktion runko, vahdit ja alustus. Kyseinen ohjelma voidaan ajaa toteutetulla järjestelmällä. Riveillä 1 ja 2 otetaan mukaan tarpeelliset kirjastot. Otsikkotiedosto *action_interface.h* on aktioille tarjotun rajapinnan esittelytiedosto. Se on kokonaisuudessaan liittessä A. Funktio *init* toimii aktiosovelluksen alustusaktion runkona. Siitä luodaan aktio automaattisesti aktiosovellusta ladattaessa. Kuvattu ohjelma summaa kaksi positiivista lukua yhteen ja kirjoittaa tuloksen lokiin. Ohjelma 4.2 on kokonainen aktiosovellus ja se voidaan suorittaa toteutetulla järjestelmällä.

Ohjelma 4.2 *Yksinkertainen aktiosovellus*

```
1  #include "action_intterface.h"
2
3  bool local(Object* obj)
4  {
5      int *value = ap_object_data(obj);
6      return *value > 0;
7  }
8
9  bool common(int parameters[], Object* participants[])
10 {
11     int *value1 = ap_object_data(participants[0]);
12     int *value2 = ap_object_data(participants[1]);
13     return *value1 < *value2;
14 }
15
16 void body(int parameters[], Object* participants[], Mask* unchanged)
17 {
18     int *value1 = ap_object_data(participants[0]);
19     int *value2 = ap_object_data(participants[1]);
20     int result = *value1 + *value2;
21     ap_log_message("Tulos: %i", result);
22     ap_participant_changed(3, unchanged);
23 }
24
25 void init(int parameters[], Object* participants[], Mask* unchanged)
26 {
27     ObjectClass* cls = ap_create_class("Luku", sizeof(int),
28         NULL, NULL, NULL);
29     unsigned ac_id = ap_add_action(body, common, 3, 0, NULL);
30     Object* obj = ap_add_new_object(cls, ac_id, local);
31     int *val = *ap_object_data(obj);
32     *val = 1;
33     obj = ap_add_new_object(cls, ac_id, local);
34     val = *ap_object_data(obj);
35     *val = 2;
36 }
```

Ojelmassa 4.2 nähdään toteutetun rajapinnan yksi heikkous. Aktiosovelluksen alustukseen tarvitaan melko paljon ohjelmakoodia, vaikka kyseessä on toiminnaltaan yksinkertainen sovellus. Alustuskoodia voidaan tiivistää esimerkiksi silmukoilla, jos järjestelmässä on paljon samanlaisia aktioita ja objekteja.

4.3.3 Aktiosäiliö

Sovellussuorittimien käyttäjän tilan muistissa aktioita säilytetään puna-mustassa puussa, jossa avaimena toimii aktion tunniste. Tähän otettiin valmis kirjastototeutus, jonka lisenssi ei rajoittanut sitä käyttävän ohjelmiston levitystä. Se on kuitenkin abstrahoitu aktiosäiliön rajapinnan taakse, jolloin se on tarvittaessa helppo vaihtaa. Puna-musta puu valittiin hyvän suorituskyvyn, pienen muistinkäytön ja yksinkertaisuuden vuoksi [8]. Käytännössä mikä tahansa kokonaislukutunnisteen perusteella toimiva, nopea hakurakenne sopisi lähes suoraan sen tilalle.

Sovellussuorittimessa aktiolla on kahdeksan eri tilaa. Näiden avulla sovellussuorittimet pitävät kirjaa aktion tilasta vuorontajan kutsuja varten. Tilojen avulla merkitään uudet aktiot, suorituksessa olevat aktiot, aktioiden aktiivisuus, poistettavat aktiot sekä alustusaktiot. Poistettavien ja alustusaktioiden käsittely sovellussuorittimessa poikkeaa muista. Alustusaktio poistetaan heti sen suorituksen loputtua. Poistettavin aktioihin liittyviä vahteja ei lasketa, eikä aktion runkoa suoriteta.

Kaikki aktiot luodaan alun perin tilassa *AC_NEW*. Heti luonnin jälkeen ne deaktivoidaan, jotta aktio lisätään vuorontajan tietorakenteisiin. Luovan aktion päätyttyä kaikki luodut aktiot aktivoidaan riippumatta niiden vahtien tilasta, jota ei tässä vaiheessa tiedetä. Aktion ensimmäisen suorituskerran tarkoituksena on suorittaa sen osallistujien paikalliset vahdit ja siten aktivoida tai deaktivoida niihin liittyvät aktiot. Luotaessa vahdit ovat tilassa *EVALUATING*.

Aktio poistetaan järjestelmästä vasta, kun se tulee suoritukseen seuraavan kerran poistofunktion kutsumisen jälkeen. Poistettuun aktioon liittyviä vahteja ei tarkisteta, vaan aktio on aina aktiivinen. Sen runkoa ei enää suoriteta poistamisfunktion kutsun jälkeen. Näin toimimalla aktion tekemät muutokset välittyvät sen osallistujien muiden aktioiden vahdeille.

Kun osallistuja poistetaan, myös aktio, jolta se poistetaan, merkitään poistettavaksi. Näin varmistetaan, että aktio ei yritä viitata vapautettuun muistiin. Objektien luokkia ei vapauteta automaattisesti. Osallistuja poistetaan aktiolta vasta vahtien suorituksen jälkeen, jotta siihen liittyvät muut aktiot saadaan käsiteltyä oikein.

4.3.4 Aktioiden ajastukset

Aktiolle voidaan lisätä vahteihin liittyvät ajastukset kahdella tavalla. Ensimmäinen tapa ovat staattiset ajoitukset, jolloin kutsutaan funktiota *ap_ add_ static_ timings*. Se ottaa parametrikseen aktion tunnisteen ja ajastustietueen, joka kopioidaan aktiolle. Staattisia ajastuksia ei voi muuttaa asettamisen jälkeen. Toinen tapa ovat dynaamiset ajastukset, jotka voidaan lisätä funktiolla *ap_ add_ dynamic_ timings*. Ajoitustietueen sijaan sille annetaan ajoitusfunktio, joka ottaa parametrikseen aktion parametrit ja palauttaa ajoitustietueen. Ajoitusfunktioita kutsutaan aina ennen ajastuksien laskemista.

Jotta ajastukset voidaan huomioida vuoronnuksessa, ne rekisteröidään vuorontajalle. Vuorontajalle rekisteröidään aktion viiveiden perusteella laskettua aikaisinta mahdollista suoritusajankohtaa. Siten vuorontajan osuus säilyy yksinkertaisena. Ajoitus rekisteröidään järjestelmäkutsulla *add_ wait_ time*. Sen toiminta ja ajoitusten käsittely vuorontajassa on kuvattu aliluvussa 4.4.

Ohjelmassa 4.3 aktiolle lisätään staattinen ajastus. Ensin on luotava ajastustietue halutuilla arvoilla, minkä jälkeen se voidaan lisätä aktiolle. Dynaamisten ajastusten lisäyksessä ohjelmassa 4.4 on esitelty ajastusfunktio ja sen lisäys aktiolle. Molemmissa esimerkeissa *ac_ id* on jonkin olemassaolevan aktion tunniste. Annetut ajastukset ovat sekunneissa.

Ohjelma 4.3 *Staattisen ajastuksen lisääminen aktiolle*

```
1 ActionTimings timings = {0};
2 timings.rate = 10
3 timings.local_timeouts[0] = 5;
4 timings.local_timeouts[1] = 2;
5 ap_add_static_timings(ac_id, timings);
```

Ajastustietueen *local_ timeouts*-kenttään voi asettaa arvon useammalle osallistujalle kuin aktiolla oikeasti on. Niistä vain aktion osallistujien mukaiset arvot otetaan huomioon. Jos esimerkiksi ohjelmassa 4.3 käytetyllä aktiolla olisi vain yksi osallistuja, ohjelman rivillä neljä asetettu ajastus jätettäisiin huomiotta.

Ohjelma 4.4 *Dynaamisen ajastuksen lisääminen aktiolle*

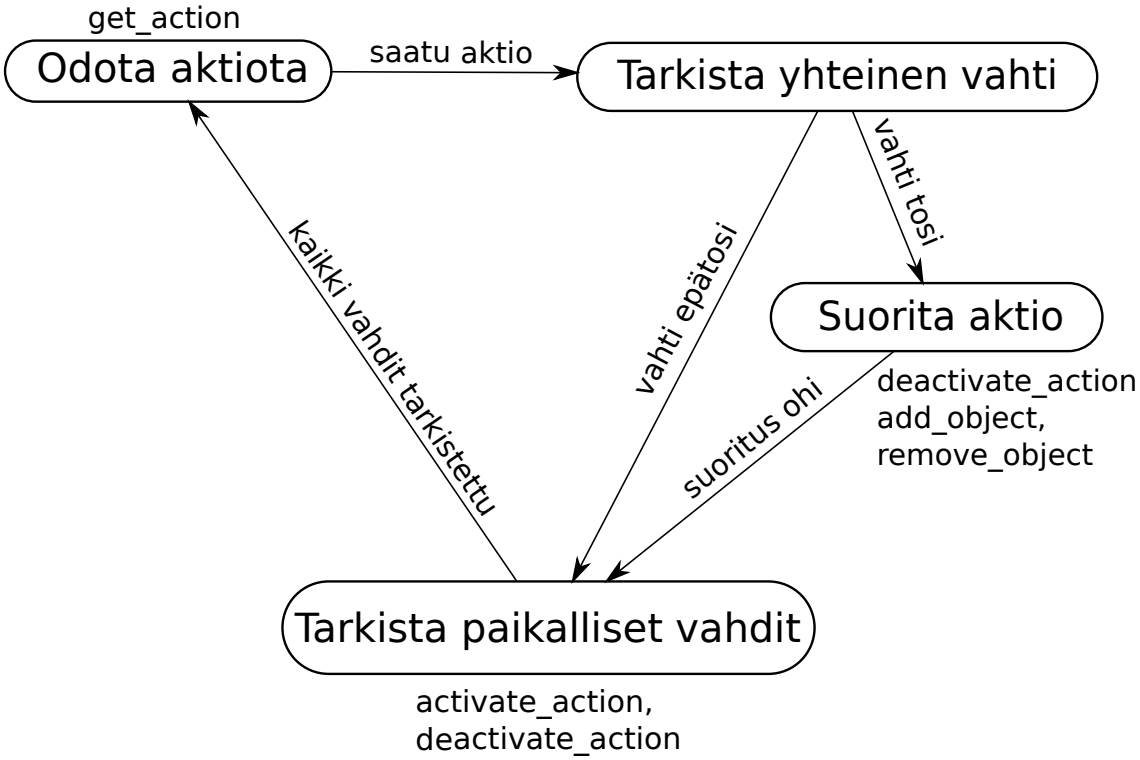
```
1 ActionTimings timing_func(int paramters[])
2 {
3     ActionTimings tim = {0}
4     /* funktio random(0) palauttaa tässä kokonaisluvun väliltä 0- x */
5     tim.local_timeouts[0] = random(parameters[0]);
6 }
7
8 ap_add_dynamic_timings(ac_id, timing_func);
```

4.3.5 Aktioiden suoritus

Kun sovellussuoritin saa aktion tunnisteen suoritusjonosta, se hakee aktion aktiosäiliöstä. Jos suoritukseen tuleva aktio on merkitty poistettavaksi, se poistetaan. Jos aktiota ei poistettu, varmistetaan vahdin tila. Yhteinen vahti suoritetaan, jos sen tilaa ei tiedetä. Tällöin jokin osallistuja on muuttunut edellisen suorituksen jälkeen. Koko vahdin ollessa tosi suoritetaan aktion runko. Paikallisten vahtien tila tarkistetaan aktiotietueen kentästä, koska vahdit joko on jo suoritettu tai tämä on aktion ensimmäinen suorituskerta. Ensimmäisellä suorituskerralla kaikkien vahtien tila on *EVALUATING*.

Aktion rungon suorituksen jälkeen ensin aktivoidaan aktion luomat aktiot. Seuraavaksi aktio poistetaan, jos se oli alustusaktio. Muussa tapauksessa lasketaan kaikki muuttuneisiin osallistujiin liittyvät paikalliset vahdit. Myös muut kuin suorituksessa olleen aktion paikalliset vahdit lasketaan. Näiden perusteella päivitetään aktioiden tilat ja valitaan suorituksen päättävä järjestelmäkutsu. Jos aktio poisti itsensä, sen omia vahteja ei tarkisteta. Sen osallistujien muihin aktioihin liittyvät paikalliset vahdit suoritetaan normaalisti. Alustusaktiot voidaan poistaa heti, koska niillä ei ole osallistujia. Niitä ei myöskään tarvitse suorittaa uudelleen, koska ne ovat jo tehneet tehtävänsä.

Paikallisia vahteja tarkistettaessa merkitään myös muuttuneisiin objekteihin liittyvien aktioiden yhteiset vahdit tilaan *EVALUATING*. Näin yhteiset vahdit tunnistetaan laskettaviksi. Lopuksi sovellussuoritin palaa hakemaan uutta aktiota suoritusjonosta. Kuvassa 4.1 on sovellussuorittimen tilakaavio. Tiloihin on myös merkitty niihin liittyvät järjestelmäkutsut. Paikallisten vahtien tarkistus käytännössä ohitetaan, jos aktion osallistujat eivät muutu. Sovellussuoritin kuitenkin käy vahtitarkistustilassa varmistamassa osallistujien tilan.



Kuva 4.1 *Sovellussuorittimien tilakaavio*

Paikalliset vahdit on laskettava, koska ne saattavat olla tilassa *EVALUATING*. Tällöin kyseessä on aktion ensimmäinen suorituskerta. Ensimmäisen suorituskerran jälkeen paikallisia vahteja ei lasketa uudelleen, jos niihin liittyvä objekti ei ole muuttunut. Käytännössä siis yhteisen vahdin ollessa epätosi paikallisia vahteja ei lasketa.

Aktion suorituksen yhteydessä kutsutaan *deactivate_action* järjestelmäkutsua, jos aktio luo uusia aktioita. Järjestelmäkutsut *add_object* ja *remove_object* lisäävät ja poistavat objekteja aktiolta. Järjestelmäkutsujen tarkemmat kuvaukset löytyvät luvusta 4.4. Kaikkien uusien järjestelmäkutsujen esittelyt löytyvät liitteestä B. Kutsut on kuitenkin tarkoitettu käytettäväksi järjestelmän sisäisesti, mutta niiden avulla voidaan esimerkiksi toteuttaa vaihtoehtoinen sovellussuoritin.

4.3.6 Laitteiston käyttö

Yhteiseen lokiin kirjoitus tehdään funktiolla *ap_log_message*. Funktio kirjoittaa järjestelmän lokiin *syslog* käyttäjätason viestejä. Lokin lukemistapa riippuu käytetystä pohjajärjestelmästä ja sen asetuksista. Jokaisessa viestissä on annetun viestin lisäk-

si kutsun suorittaneen sovellussuorittimen PID (engl. Process identifier) ja viestin kirjoitusaika. PID on Linuxin prosessille antama yksikäsitteinen tunniste [7]. Parametreikseen funktio ottaa merkkijonon, joka kuvaa viestin muodon, ja tarvittavan määrän parametreja viestin täyttämiseen.

Aktiopohjaisesta laitteistotuesta toteutetaan testausta varten syötteen haku, joka on kuvattu aliluvusa 4.3.7. Sen avulla voidaan sekä monipuolistaa toteutettavia aktioita että testata konseptin käytettävyyttä. Muita laitteita voi toteutetussa järjestelmässä käyttää Linuxin rajapintojen avulla, jotka eivät noudata aktioparadigmaa. Ne aiheuttavat ylimääräistä odottelua järjestelmässä, koska laitteen vastausta odotettaessa ei voida suorittaa muita aktioita. Jatkossa tarkoituksena on muuttaa laitteistotuki kokonaan aktioparadigmaan sopivammaksi.

4.3.7 Käyttäjän syötteen käsittely

Syötteen antamiseen käytetään Linuxin *proc*-virtuaalitiedostojärjestelmää. Syötteen vastaanottajana toimii virtuaalitiedosto */proc/actionsys/input*. Tiedostoon kirjoitettu syöte siirretään syötettä odottavista aktioista ensimmäiselle, jos sen varaamassa puskurissa on riittävästi tilaa. Jos tilaa ei ole, ylijäävä osa menee hukkaan. Tiedosto on kuvattu tarkemmin aliluvussa 4.4.3.

Funktiolla *ap_add_user_input* voidaan aktioon liittää syöteobjekti, jonka avulla aktiolle välitetään käyttäjän syöte. Funktio ottaa parametrikseen syötteen käsittelevän aktion tunnisteen, varattavan puskurin koon ja viestin, joka näytetään käyttäjälle. Viestin tarkoitus on kertoa, mitä aktio haluaa syötteeksi. Käyttäjä voi lukea viestin virtuaalitiedostosta */proc/actionsys/input*.

Syötteen käsittelevä aktio on tavallinen aktio, jonka yksi osallistuja sisältää käyttäjän syötteen. Tähän objektiin liittyvä paikallinen vahti varmistaa, että aktio ei tule suoritukseen ennen kuin syöte on saatavilla. Jos aktio haluaa odottaa syötettä uudelleen, se voi kutsua funktiota *ap_reset_input*. Funktio ottaa parametrikseen käsittelevän aktion tunnisteen ja käytettävän syöteobjektin. Sen jälkeen aktio laitetaan uudelleen syötejonoon. Funktio nollaa syötteen tallennukseen käytetyn puskurin.

Syöteobjektin data on tietuetyyppiä *UserInput*. Sen tärkein kenttä on *buffer*, joka sisältää saadun syötteen. Puskuri on tyyppiä *char**. Järjestelmä ei muokkaa annettua syötettä mitenkään, joten käsittelevän aktion on tehtävä kaikki itse. Aktion on myös varauduttava virheelliseen syötteeseen.

Ohjelmassa 4.5 on esitetty käyttäjän syötteen käsittely. Funktio *handle_body* on syötteen käsittelevän aktion runko ja funktio *req_body* on syötettä pyytävän aktion runko. Syötettä pyydettäessä on ensin luotava aktio, joka käsittelee syötteen. Sen jälkeen sille voidaan lisätä syöteobjekti. Esimerkissä syötettä käsittelevällä aktiolla ei ole muita osallistujia kuin syöteobjekti. Syötteen käsittelevä aktio ottaa ensin syötetietueen sen sisältävästä objektista. Sen jälkeen se kirjaa saamansa syötteen järjestelmän lokiin ja pyytää syötettä uudelleen. Lopuksi se ilmoittaa järjestelmälle, että se on muuttanut kaikkia osallistujiaan asettamalla *unchanged* muuttujan nollaksi.

Ohjelma 4.5 *Syötettä pyytävä ja käsittelevä aktio*

```
1  void handle_body(int params[], Object *participants[], Mask *unchanged)
2  {
3      UserInput* input = (UserInput*)ap_object_data(participants[0]);
4      ap_log_message("Syöte: %s", input->buffer);
5      // 0 tarkoittaa pyynnön suorittavaa aktiota
6      ap_reset_input(0, input);
7      *unchanged = 0;
8  }
9
10 void req_body(int *params, Object **participants, Mask *unchanged)
11 {
12     int ac_id = ap_add_action(handle_body, NULL, 1, 0, NULL);
13     ap_create_user_input(ac_id, 150, "Lokiviesti\n");
14 }
```

4.4 Vuorontaja

Vuorontaja on toteutettu Linuxin kernel-säikeenä, joka on lukittu sovellussuorittimilta vapaaksi jäävälle ytimelle. Säie odottaa Linuxin *waitqueue*-rakenteessa, että suoritusjonossa olisi tilaa ja järjestelmässä aktiivisia aktioita. Sovellussuorittimien tekemät järjestelmäkutsut herättävät säikeen tarkistamaan järjestelmän tilan ja suorittamaan vuoronnuksen, jos se on mahdollista. Lisäksi vuorontaja herätetään säännöllisesti, jotta aktioiden ajastukset voidaan tarkistaa. Vuorontaja toimii ytimen tilassa ja siten eri muistiavaruudessa kuin sovellussuorittimet.

Järjestelmän käynnistyessä vuorontaja yrittää käynnistää sovellussuorittimet käyttäen Linuxin *usermode_helper*-tominnallisuutta. Sen avulla käyttöjärjestelmä voi

suorittaa käyttäjätilan ohjelmia. Jos sovellussuorittimien käynnistys ei jostain syystä onnistu, vuorontaja kirjaa virheviestin lokiin. Vuorontajan lokina käytetään Linuxin järjestelmälokia *syslog*. Sitä voi lukea esimerkiksi *dmesg*-ohjelman avulla.

4.4.1 Vuoronnus ja suoritusjono

Toteutettu vuorontaja valitsee ajettavat aktiot satunnaisesti aktiivisina olevien aktioiden joukosta. Koska aktion yhteinen vahti tarkistetaan vasta juuri ennen aktion rungon suoritusta, vuoronnuksessa ei voida olla varmoja aktion vireydestä. Siksi vuorontaja keskittyy aktiivisiin aktioihin. Se myös lukitsee aktion osallistujat, jolloin muut samoja objekteja käyttävät aktiot eivät voi päätyä suoritukseen.

Suoritusjono on toteutettu yksinkertaisena rengaspuskurina, johon sijoitetaan suoritettavien aktioiden tunnukset. Sovellussuorittimet hakevat aktioita järjestelmäkutsun *get_ action()* avulla. Se palauttaa suoritusjonon vanhimman alkion. Jotta sovellussuorittimet eivät saisi samaa aktiota suoritukseen, suoritusjonon käsittely on suojattu semaforilla. Suoritusjonon seuraavan aktion voi selvittää järjestelmäkutsulla *peek_ runqueue()*. Se ei poista palautettua aktiota suoritusjonosta.

Sovellussuorittimien sammutus tapahtuu laittamalla ne nukkumaan, jos ne yrittävät hakea aktiota tyhjästä suoritusjonosta. Samalla vuorontaja herätetään yrittämään vuoronnusta. Vuoronnuksen epäonnistuessa kutsun tehnyt sovellussuoritin jää nukkumaan. Sovellussuorittimet nukkuvat Linuxin *waitqueue*-rakenteessa.

4.4.2 Aktiot ja objektit

Aktioiden suorituksen loppuessa sovellussuoritin suorittaa jonkin taulukossa 4.3 olevan järjestelmäkutsun. Näiden avulla vuorontajan on helppo pitää kirjaa vuoronnusmahdollisuuksista. Jokainen kutsu herättää vuorontajan tarkistamaan, onko vuoronnus mahdollista. Jos vuoronnus on mahdollista, se suoritetaan.

Aktioita poistettaessa vuorontajasta täytyy varmistaa objektien lukitusten oikeellisuus. Käytännössä aktion osallistujien lukitus avataan, koska aktio on aina suorituksessa, kun se poistetaan. Vuorontaja kuitenkin varmistaa tilanteen olevan oletuksen mukainen, jotta ei synny ongelmia.

Vuoronnuksen yksinkertaistamiseksi aktioiden tilasta pidetään kirjaa vuorontajan

Taulukko 4.3 *Suorituksen lopettavat järjestelmäkutsut.*

Järjestelmäkutsu	**Kuvaus**
activate_action(action_id, action_state)	Kirjaa aktion aktiiviseksi tilasta action_state
deactivate_action(action_id, action_state)	Kirjaa aktion ei-aktiiviseksi tilasta action_state
remove_action(action_id)	Poistaa aktion järjestelmästä

tietorakenteissa. Tällöin ei tarvitse käsitellä sovellussuorittimien käyttäjätilan muistia ytimen tilasta käsin. Vuorontajan tietorakenteisiin tallennetaan vain aktion tunniste, tila ja osallistujat. Osallistujista tallennetaan tunniste, lukitustila ja viittauslaskuri. Vuorontajassa sekä aktioita että objekteja säilytetään B-puussa. Sen toteutus on osa Linuxia ja valittiin yksinkertaisen rajapintansa vuoksi. B-puu on nopea hakurakenne, jossa voidaan käyttää kokonaislukuja avaimina [1].

Objekteja voisi säilyttää myös pelkästään aktioiden yhteydessä kuten sovellussuorittimet (luku 4.3) tekevät. Vuorontajan on kuitenkin pidettävä huolta lukituksista, ja B-puusta voidaan hakea objektitietueet nopeasti. Näin osallistujia lisättäessä voidaan helposti ja nopeasti selvittää onko kyseinen objekti jo olemassa. Lukitsemisen nopeuttamiseksi aktiolla on kuitenkin osoittimet suoraan osallistujiensa tietueisiin.

Objektien käsittelyyn on kaksi järjestelmäkutsua: *add_ object* ja *remove_ object*. Molemmat ottavat parametrikseen aktion ja objektin tunnisteen. Ensimmäinen lisää annetun objektin aktiolle, minkä jälkeen aktiota ei oteta suoritukseen kyseisen objektin ollessa lukittuna. Toinen vastaavasti poistaa objektin aktiolta.

Objektia lisättäessä ensin tarkistetaan objektien B-puusta, onko samalla tunnisteella jo luotu objekti. Jos objekti löytyi, se liitetään aktioon. Muussa tapauksessa luodaan uusi objekti. Objektin tietue vapautetaan, kun se poistetaan viimeiseltä siihen viittavalta aktiolta. Ennen vapauttamista tietue poistetaan B-puusta.

Vuorontajan näkökulmasta aktio voi olla neljässä tilassa. Tilat on kuvattu taulukossa 4.4. Tämä jako on epätarkempi kuin sovellussuorittimilla, mutta riittävä vuorontajalle. Tila *ACTION_ STATE_ NEW* kertoo vuorontajalle, että aktioita ei löydy sen tietorakenteista. Tila tallennetaan kokonaislukuna.

Vuorontajassa ei erikseen ole aktion luontikutsua, vaan sovellussuoritin deaktivoi tai aktivoi aktion tilasta *ACTION_ STATE_ NEW*. Silloin aktiolle luodaan tietue, joka lisätään aktioiden B-puuhun. Objekteja voidaan lisätä vain olemassaoleville

Taulukko 4.4 *Aktion tilat vuorontajassa.*

Tilan nimi	Kuvaus	Lukuarvo
ACTION_STATE_NEW	Uusi aktio, lisättävä kirjanpitoon	1
ACTION_STATE_ACTIVE	Aktio voidaan ottaa suoritukseen	2
ACTION_STATE_INACTIVE	Aktiota ei voida ottaa suoritukseen	4
ACTION_STATE_RUNNING	Aktio on suorituksessa	8

aktioille, jolloin luodaan ja tallennetaan objektin tietue.

Ainoastaan sovellusten alustusaktiot aktivoidaan luotaessa. Muut aktiot deaktivoidaan, jotta niille voidaan lisätä osallistujat ja osallistujat voidaan alustaa haluttuun tilaan. Alustusaktiot voidaan aktivoida heti, koska niillä ei ole osallistujia.

Aktioiden ajastuksia varten vuorontaja pitää kirjaa aktion seuraavasta mahdollisesta suoritusajankohdasta. Jos aktion seuraava suoritusaika on tulevaisuudessa, se käsitellään kuin se ei olisi aktiivinen. Järjestelmäkutsu *add_wait_time* asettaa aktion seuraavan suoritusajan parametrinaan saamansa odotusajan verran tulevaisuuteen. Odotusaika tulkitaan sekunteina. Aikojen laskentaan käytetään Linuxin *jiffies*-muuttujaa, jota kasvatetaan yhdellä jokaisessa ajastinkeskeytyksessä [7].

Toteutettu vuoronnusalgoritmi on hyvin yksinkertainen. Ensin vuorontaja valitsee positiivisen satunnaisluvun, joka on korkeintaan aktioiden määrä järjestelmässä. Sen jälkeen vuorontaja käy läpi aktioita ja ohittaa arvotun luvun osoittaman määrän tekemättä mitään. Kun aktioita on ohitettu tarpeeksi, vuorontaja tarkistaa jäljelle jääneiden aktioiden osallistujat ja aktiivisuuden. Ensimmäinen aktiivinen aktio, jonka osallistujat eivät ole lukossa ja jolle asetetut viiveet ovat kuluneet, valitaan suoritukseen. Jos ohituksen jälkeen ei löydy yhtäkään sopivaa aktiota, vuoronnus aloitetaan alusta ilman ohitusta. Näin pyritään varmistamaan vuoronnuksen onnistuminen, jos järjestelmässä on aktiivisia aktioita. Käytetyn B-puun rajapinnan vuoksi aktiot käydään läpi suurimmasta tunnisteesta alkaen.

4.4.3 Käyttäjälle tarjotut toiminnot

Järjestelmän ulkopuolelle tarjotaan erikoiskansio, jossa sijaitsevat järjestelmän tilaa kuvaavat virtuaalitiedostot: *scheduler*, *actions*, *objects* ja *processors*. Taulukko 4.5 kertoo, mitä tietoja tiedostoista voi lukea. Tiedostot ovat osa Linuxin *proc*-virtuaalitiedostojärjestelmää ja sijaitsevat kansiossa */proc/actionsys*. Samasta kan-

siosta löytyy aliluvussa 4.3 mainittu järjestelmän syötetiedosto *input*. Tiedostoista luettaessa tilat ovat numeromuotoisia.

Vuorontajan aktioiden tiloille käyttämät lukuarvot ovat taulukossa 4.4. Objektien lukitustila on 0 kun objekti on vapaa ja 1 kun se on lukittu. Tiedot tulevat tiedostosta riveinä. Yksi rivi vastaa yhtä aktiota, objektia tai sovellussuoritinta. Esimerkiksi tiedostosta *actions* yksi rivi voisi näyttää tältä:

```
1          8          1,3,4.
```

Esimerkissä aktio tunnisteella 1 on suorituksessa ja sen osallistujien tunnisteet ovat 1, 3 ja 4.

Taulukko 4.5 *Virtuaalitiedostojen sisältö*

Tiedoston nimi	**Tiedoston kuvaus**	**Kerrotut tiedot**
actions	järjestelmässä olevat aktiot	tunniste, tila, osallistujat
objects	järjestelmässä olevat objektit	tunniste, lukituksen tila, viittaukset
processors	sovellussuorittimien tiedot	PID, tila, aktio
scheduler	vuoronnuksen tila	suoritusjonossa olevat aktiot
input	syötettä odottavien aktioiden viestit	aktion viesti

Aliluvun 4.3.6 mukaisesti tiedosto *input* toimii aktiopohjaisen syötetoiminnallisuuden rajapintana käyttäjille. Kaikki tiedostoon kirjoitettu data ohjataan syötettä odottaville aktioille. Jokainen kirjoituskerta ohjataan eri objektin puskuriin.

Objektien puskurit ovat sovellussuorittimien varaamia ja ne eivät yleensä suorita syötteen antamista. Tästä syystä käyttäjän antama syöte on kopioitava toisen prosessin muistiavaruuteen. Tähän käytetään Linuxissa toteutettua *access_process_vm* funktiota. Se mahdollistaa tietyn prosessin muistiavaruuden käsittelyn toisesta prosessista.

Sovellussuorittimet rekisteröivät aktioiden syötepyynnöt *request_input* järjestelmäkutsulla. Järjestelmäkutsu ottaa parametrikseen pyytävän aktion tunnisteen, syötepuskurin, puskurin koon, viestin, viestin pituuden sekä osoitteen lippumuuttujaan. Lippumuuttujana käytetään syöteobjektin kenttää *processed*. Muuttujan arvo asetetaan ykköseksi kun syöte on tallennettu syötepuskuriin. Samalla syötettä pyytänyt aktio asetetaan aktiiviseksi, jotta sen vahdit voidaan tarkistaa.

Jos syötetiedostoon kirjoitetaan, kun yksikään aktio ei odota syötettä, syöte jäte-

tään huomiotta. Näin pyritään vähentämään ongelmallisten syötteiden päätymistä aktioille, sekä vältetään muistin kuluttamista turhaan. Syötejonon ollessa tyhjä käyttäjä tai järjestelmä eivät yleisessä tapauksessa tiedä varmasti, mikä aktio pyytää syötettä seuraavaksi vai pyytääkö mikään. Muistia säästyy, kun syötetiedostoon ei tarvitse liittää erillistä puskuria, vaan voidaan käyttää suoraan aktioille varattuja puskureita. Samalla toiminta nopeutuu, koska syötettä ei tarvitse kopioida ensin järjestelmän puskuriin ja sitten aktion puskuriin.

Seuraavassa esimerkissä näytetään yksi tapa syötetiedoston lukemiseen ja syötteen antamiseen. Esimerkkissä syötettä odottaa ohjelmassa 4.5 esitelty aktio. Ensimmäiseksi tarkistetaan minkälaista syötettä aktio haluaa. Tämän jälkeen voidaan haluttu syöte antaa aktiolle. Esimerkissä on käytetty ulkoisia komentoja *cat* ja *echo*. Merkintä $: tarkoittaa Linuxin komentokehotetta.

```
1 $: cat /proc/actionsys/input
2 Lokiviesti
3 $: echo "Hei maailma" > /proc/actionsys/input
```

Toteutettu syötetoiminto ei takaa oikean muotoista syötettä, joten aktion on varauduttava virheellisiin syötteisiin. Vaikka syöte käsitellään merkkijonona, on mahdollista, että se ei ole C:n standardin mukainen merkkijono. Tämä tarkoittaa, että merkkijonon viimeinen merkki ei välttämättä ole *NULL*. Syötetietueessa olevaa puskurin kokotietoa voidaan käyttää puskurin lukemisen rajoittamiseen.

4.4.4 Sovellussuorittimien hallinta

Sovellussuorittimien suorituksessa olevista aktioista pidetään kirjaa linkitetyssä listassa. Siihen tallennetaan sovellussuorittimen PID ja suorituksessa olevan aktion tunniste. Tunniste 0 tarkoittaa sovellussuorittimen olevan pois käytöstä. Silloin se odottaa uutta aktiota suoritukseen.

Sovellussuorittimien hallinnan helpottamiseksi on toteutettu kaksi järjestelmäkutsua: *register_approc* ja *unregister_approc*. Näillä kutsuilla käynnistyvä sovellussuoritin kertoo olemassaolostaan ja sammuva poistumisestaan. Ne auttavat varmistamaan kirjanpidon oikeellisuuden ja siten mahdollistavat signaalien lähetyksen. Rekisteröintiin käytettävä *register_approc* ottaa parametrikseen sovellussuorittimen PID:n. Poistoon käytetty *unregister_approc* ottaa PID:n lisäksi parametrikseen virhekoodin, joka aiheutti poiston. Jos virhettä ei ole tapahtunut, annetaan koodi 0.

Sovellussuorittimen rekisteröinti lisää sovellussuorittimien listaan uuden alkion, jossa pidetään kirjaa uuden sovellussuorittimen tilasta. Rekisteröinnin poisto vastaavasti poistaa alkion listasta. Näin vuorontajan ei tarvitse selvitellä sovellusuorittimien tunnisteita. Tunnisteita ei saada suoraan käynnistyksen yhteydessä, koska sovellusuorittimet käynnistetään yhtenä sovelluksena. Tämä sovellus jakautuu useammaksi prosessiksi, jotka toimivat sovellusuorittimina. Samalla se alustaa *action_loader* apuohjelman tarvitseman FIFO-tiedoston.

5. ARVIOINTI JA TULEVA KEHITYS

Toteutettu järjestelmä sisältää aktiosuorittimen perusosat ja voi siten toimia pohjana jatkokehitykselle. Laajuuden rajaamiseksi mahdollisia ominaisuuksia on jätetty toteuttamatta. Järjestelmä pyrittiin rakentamaan niin, että sitä on helppo laajentaa tarpeen mukaan.

Järjestelmän avulla voidaan paremmin selvittää aktiosuorittimen suorituskykypotentiaalia, koska järjestelmästä saadaan muiden ohjelmien määrä minimoitua helposti. Käytetty laitteisto mahdollistaa tulosten keräämisen laajemmalta yleisöltä, mikä auttaa realistisen kuvan saamista. Samalla voidaan saada kokemuksia paradigmasta ja parannusehdotuksia aktioille tarjottuun rajapintaan.

Kehittäjäyhteisöltä kokemusten ja ehdotusten kerääminen toimii hajautettuna tutkimuksena. Siinä kaikki asiasta kiinnostuvat voivat helposti osallistua kehitykseen ja esitellä omia havaintojaan tutkimuksen avuksi. Tällainen tutkimustapa poikkeaa yleisesti käytetyistä menetelmistä toimien siten kokeiluna uudelle tutkimusmenetelmälle.

5.1 Rajapinnat

Vuorontajan rajapinta saatiin pidettyä melko yksinkertaisena. Siihen kuuluu kaikkiaan kaksitoista järjestelmäkutsua, joista neljä ei suoranaisesti ole osa vuorontajaa. Toteutetut virtuaalitiedostot on liitetty vuorontajaan, koska ne toteutettiin osaksi muokattua Linuxia. Niiden tarjoama toiminnallisuus on lähinnä virheenetsintää helpottavaa tilannetietoa, eikä siten järjestelmän toiminnan kannalta välttämätöntä. Syötetiedostoa vastaava toiminnallisuus on tarpeellinen, mutta sen ei tarvitse kuulua vuorontajaan.

Vuorontajaan kuulumattomat järjestelmäkutsut ovat suoritusjonosta aktion haku, ajossa olevan aktion haku, seuraavan suoritukseen tulevan aktion tarkistus ja syöt-

teen pyytäminen. Tarvittaessa suoritusjono voidaan eriyttää vuorontajasta siten, että vuorontajalle tarjotaan kirjoitusrajapinta ja sovellussuorittimille lukurajapinta. Syötetoiminto voidaan siirtää osaksi sovellussuorittimia. Toteutetussakin järjestelmässä se voitaisiin määritellä sovellussuorittimien osaksi, mutta toteutuksen ollessa vuorontajan ohjelmakoodissa, se oli loogista kuvata osana vuorontajaa.

Myös loppukuittauksiin käytettävät 3 järjestelmäkutsua voitaisiin yhdistää yhdeksi kutsuksi, joka ottaa enemmän parametreja. Loppukuittaus ei ole välttämätön, mutta vähentää vuorontajan kuormaa, kun sen ei tarvitse aktiivisesti tarkkailla järjestelmää. Ajastuksien takia vuorontajan täytyy kuitenkin käytännössä ajoittain suorittaa vuoronnus, vaikka varsinaisia kuittauksia ei olisi tullut. Jos järjestelmää muutetaan siten, että vuorontaja käsittelee aktiosäiliötä suoraan, vuorontajan rajapinnasta voidaan poistaa objektien käsittely.

Aktioille tarjottu rajapinta oli toissijainen suunnittelutavoite, joka jäi osittain saavuttamatta. Nyt toteutettu rajapinta riittää todennäköisesti melko pitkälle, mutta sen käyttö voi olla haastavaa. Samalla todennäköisesti on useita tilanteita, joissa rajapintaan tarvittaisiin lisäyksiä.

5.2 Suorituskyky

Varsinaista suorituskykytestausta ei tämän työn puitteissa suoritettu. Tässä aliluvussa esitetyt ongelmat ovat toteutuksesta pääteltyjä mahdollisia heikkouksia. Niiden todellista vaikutusta on tutkittava tarkemmin, jotta kehitystä voidaan kohdentaa järkevästi.

Vuoronnusalgoritmin tehostaminen ja mahdollinen laajentaminen paremmilla reaaliaikaominaisuuksilla vaatii lisätutkimusta. Nykyiseen järjestelmään on melko helppo toteuttaa uusia vuoronnusalgoritmeja, jos ne eivät tarvitse uutta tietoa sovellussuorittimilta. Lisätietojen saaminen vaatii joko uusia järjestelmäkutsuja tai sovellussuorittimien muistin suoraa käsittelyä. Jälkimmäistä tapaa on käytetty syötetoiminnallisuuden toteutuksessa ylimääräisten puskurien välttämiseksi.

Mahdollisena suorituskykyongelmana järjestelmässä on Linuxin prosessipohjaisuus. Sen takia järjestelmässä on nyt kaksi vuorontajaa. Ensimmäinen on Linuxin prosessivuorontaja, joka vuorontaa sovellussuorittimia, aktiovuorontajaa ja muita käynnistyviä prosesseja. Toinen on luvuissa 3.3 ja 4.4 kuvattu aktiovuorontaja.

Kahden vuorontajan olemassaolo aiheuttaa ylimääräistä kuormaa, eikä prosessivuorontaja ole tarpeellinen puhtaassa aktiojärjestelmässä. Nykyinen rakenne kuitenkin mahdollisti nopean testauksen ja toteutuksen monien tarpeellisten työkalujen ollessa jo olemassa. Suurimpana hyötynä on valmis laitteistotuki ja käyttöliittymät, joiden avulla järjestelmää pystyy käyttämään. Linuxin vuorontaja on kuitenkin tehokas [7], joten sen ei pitäisi aiheuttaa suuria suorituskykyongelmia.

Järjestelmässä on vielä varaa suorituskykyoptimoinnille. Järjestelmää kehitettäessä keskityttiin enemmän rajapintoihin ja tomivuuteen kuin suorituskykyyn. Tästä johtuen suorituskykytulokset ovat lähinnä suuntaa antavia, mutta kuitenkin parempia kuin olemassaolevilla testaustyökaluilla on saatu. Toteutuksessa kuitenkin pyrittiin valitsemaan suorituskykyisiä tietorakenteita ja toimintatapoja.

5.3 Jatkokehitysmahdollisuuksia

Aktioiden ja objektien kopioinnin toteutukset jäivät teknisten ongelmien ja aikatauluhaasteiden vuoksi tässä yhteydessä toteuttamatta. Niiden toteuttaminen järkevällä tavalla on järjestelmän käytettävyyden kannalta tärkeää. Esimerkiksi aktiopohjaisten tietorakenteiden toteuttaminen on haastavaa tai jopa mahdotonta ilman sopivaa kopiointitoimintoa.

Aktioiden osallistujien määrärajoituksen poistaminen on yksi mahdollinen kehityskohde. Tähän pitää valita sopiva tietorakenne, jolla osallistujat välitetään aktioille. Aktion on pystyttävä saamaan haluamansa osallistuja helposti. Nykyisessä toteutuksessa aktion on tiedettävä missä järjestyksessä osallistujat lisätään. Yksi mahdollisuus on liittää osallistujille nimet, joiden perusteella aktio pyytää niitä järjestelmältä. Nimien tulisi olla joko yksikäsitteisiä tai aktiokohtaisia. Aktiokohtaisessa nimeämisessä objektia lisättäessä aktiolle objektille annettaisiin nimi, joka kuvaa objektin roolia aktiossa. Nimien ja aktioiden tunnisteiden perusteella voitaisiin tehdä hakurakenne, josta objektit haetaan. Nimien käyttö tosin lisää muistinkulutusta.

Toiminnaltaan yksinkertaisin ratkaisu määrärajan poistoon olisi muuttuvankokoisen taulukon käyttö objektien tallennukseen. Silloin aktiot edelleen tunnistaisivat osallistujansa järjestyksen perusteella. Tietorakenteen lisäksi on huomioitava muutosten seuranta tai järjestelmän suorituskyky kärsii. Jos objektien muuttumista ei voida merkitä jotenkin, järjestelmän on aina tarkistettava kaikkien aktion osallistujien vahdit. Osallistujien määrän kasvaessa tämä on entistä suurempi ongelma,

koska vahtien tarkistus on kestoltaan suoraan verrannollinen osallistujien määrään.

Kuitenkin ennen kuin osallistujamäärää lähdetään kasvattamaan olisi hyvä suorittaa lisätutkimusta. Tällä hetkellä ei vielä ole selvää kuinka suuri ongelma määrärajoitus on. Myöskään osallistujamäärän muuttamisen vaikutuksia suorituskykyyn ei ole selvitetty tarkasti. On mahdollista, että käytännön sovellukset saadaan toteutettua ongelmitta nykyisellä määrällä. Testauksessa käytetyissä sovelluksissa aktioilla on ollut korkeintaan viisi osallistujaa.

Yksi toteuttamatta jätetyistä ominaisuuksista on reaaliaiaikatuki. Toteutetut ajastukset ovat vain osa täyttä tukea. Niiden lisäksi on toteutettava aikarajat eli viimeiset mahdolliset suoritusajat. Jos aikarajoista halutaan tarkkoja, vuorontajan täytyy pystyä priorisoimaan aktioita aikarajojen perusteella.

Aktioparadigmaan sopivan laitteistotuen jatkokehitys on tärkeä osa tulevaa kehitystä. Sen tarkka toteutusmalli täytyy suunnitella kunnolla. Parhaiten aktioparadigmaan sopii malli, jossa laitteistoa kutsuva aktio on eri kuin vastauksen käsittelevä aktio. Näin saadaan vastausta odoteltaesa helposti suoritettua muita aktioita, eikä tarvitse erikseen palata aktion suoritukseen. Järjestelmään toteutettu syötteen pyytäminen seuraa aktiopohjaista mallia ja sitä voidaan käyttää jatkokehityksen lähtökohtana.

Toteutettu sovellussuorittimien rajapinta riittää monenlaisiin sovelluksiin, mutta sen käytettävyydessä on vielä parannettavaa. Erityisesti laitteistotuki ja reaaliaikaominaisuudet ovat puutteellisia. Rajapinta on käytännössä melko matalan tason rajapinta, koska aktiot ja objektit ovat järjestelmän perusosia. Tästä syystä kehityksessä on hyvä harkita, kuuluuko jokin toiminnallisuus tähän rajapintaan vai voidaanko se toteuttaa olemassaolevien toimintojen yhdistelmänä korkeammalla tasolla. Tässä pohdinnassa kannattaa huomioida myös toiminnon suorituskykytarpeet. Jokin korkeamman tason toiminto voi kannattaa tuoda suoraan järjestelmän rajapintaan jo siksi, että se on paljon nopeampi toteuttaa omana kokonaisuutenaan.

Järjestelmän hajauttaminen on myös mahdollinen tutkimuskohde. Siihen vaaditaan objektien ja aktioiden tilojen synkronointi osien välillä, vuorontajan hajautus tai muu tapa tukea hajautusta, hajautuksen virheenkäsittely ja kommunikointitoimintojen lisäys sovellussuorittimiin. Tilojen synkronointiin on mahdollista käyttää hajautettuja aktio- ja objektisäiliöitä. Sellaisten toteuttaminen on kuitenkin vaativa tehtävä.

Koska aktiosuoritin on matalan tason kokonaisuus, tulevaisuudessa kannattaa pyrkiä tarjoamaan korkeamman tason ohjelmointityökaluja aktiosuorittimelle. Korkeamman tason työkalut nostavat paradigman houkuttelevuutta ja helpottavat ohjelmointia. Yksi potentiaalinen malli on professori Hannu-Matti Järvisen esittelemä subjektipohjainen ohjelmointi[3]. Uusilla työkaluilla voidaan myös vähentää aktio-paradigman ja ihmisten ajattelun erojen aiheuttamia ongelmia.

Tärkein kehityskohde on vajaaksi jäänyt aktioille tarjottu rajapinta. Vaikka sen päälle kehitettäisiin korkeamman tason työkaluja, sen täytyy olla riittävän kattava kaikkien sovellusten toteuttamiseen. Tarvittaessa myös vuorontajan rajapintaa voidaan tässä vaiheessa laajentaa, mutta laajennuksia harkitessa tulee pitää mielessä vuorontajan mahdollinen laitteistototeutus.

6. YHTEENVETO

Aktioparadigma tarjoaa tavan mallintaa ohjelmien suoritusta. Se on vaihtoehto prosessipohjaiselle suoritusmallille, joka on nykyisin käytössä laitteistotasolla. Aktioparadigmaa noudattava järjestelmä eli aktiosuoritin mahdollistaa monien rinnakkaisesta suorituksesta johtuvien ongelmien ratkaisemisen. Ohjelmoijan ei tarvitse huolehtia poissulkemisesta tai viestinvälityksestä ja synkronointi onnistuu helposti.

Aktiosuoritin koostuu vuorontajasta, sovellussuorittimista, aktiosäiliöstä ja objektisäiliöstä. Aktiot ovat suorituksen perusyksikkö ja korvaavat prosessimallin säikeet. Objektit ovat järjestelmän datasäiliöitä ja toimivat aktioiden välisenä jaettuna muistina. Jokainen objekti on kerrallaan yhden aktion käytössä, joten aktio voi käsitellä niitä vapaasti.

Aktio voi päästä suoritukseen ollessaan vireessä. Vireessä olevan aktion vahti on tosi. Lisäksi aktion osallistujien tulee olla vapaina. Aktioiden välinen poissulkeminen hoidetaan lukitsemalla aktion osallistujat, kun aktio on suorituksessa.

Vuorontaja huolehtii aktioiden valitsemisesta suoritukseen ja objektien lukituksista. Sovellussuorittimet suorittavat aktiot siinä järjestyksessä, kun ne tulevat vuorontajalta. Järjestelmä pystyy suorittamaan yhtä monta aktiota rinnakkain kuin käytössä on sovellussuorittimia ja aktioita on vireessä. Sovellussuorittimien määrää voidaan muuttaa ilman vaikutusta aktioiden toimintaan. Muutos voidaan tehdä jopa järjestelmän ollessa käynnissä. Silloin säästyy virtaa, kun osa laitteistosta voidaan laittaa lepotilaan.

Toteutettu aktiosuoritin on tarkoitettu pääasiassa jatkotutkimuksen pohjaksi, mutta toimii myös itsenäisenä järjestelmänä. Sen avulla aktioparadigmaa voidaan esitellä paremmin ja paneutua paradigman haasteisiin. Samalla saatiin pohja laitteistopohjaiselle vuorontajalle toteuttamalla sille rajapinta, jonka mukaan se voi toimia.

LÄHTEET

[1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, pp. 173 – 189, 1972.

[2] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel.* O'Reilly, 2005, iSBN 0-596-00565-2.

[3] H.-M. Järvinen, "Actions, objects, and subjects," in *The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2013.

[4] D. Kahneman, *Thinking Fast and Slow.* Macmillan, 2011, iSBN 978-1429969352.

[5] R. Kurki-Suonio, *A Practical Theory of Reactive Systems.* Springer, 2005, iSBN 3-540-23342-3.

[6] L. Lamport, "The temporal logic of actions," *Research Report 71*, 1991.

[7] R. Love, *Linux Kernel Developement.* Sams, 2004, iSBN 0-672-32512-8.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to algorithms.* MIT Press, 2001, iSBN 0-262-03383-7.

LIITE A. AKTIOILLE TARJOTUN RAJAPINNAN ESITTELY

```
1  #define _GNU_SOURCE
2  #ifndef ACTION_INTERFACE_H
3  #define ACTION_INTERFACE_H
4
5  #include <stdbool.h>
6  #include <stdlib.h>
7
8  #define MAX_PARTICIPANTS 32
9  #define MESSAGE_SIZE 64
10
11 struct ObjectClass;
12 struct Object;
13
14 typedef int Mask;
15
16 struct UserInput {
17     bool processed;
18     size_t size;
19     char* buffer;
20     char message[MESSAGE_SIZE];
21 };
22
23 typedef void (*ObjectDestructor)(void* value);
24 typedef void (*ObjectConstructor)(void* value);
25 typedef void (*ObjectCopyConstructor)(void* old_value, void* new_value);
26
27 typedef bool (*CommonGuardFunc)(int parameters[],
28     struct Object* participants[]);
29 typedef bool (*LocalGuardFunc)(struct Object* obj);
30 typedef void (*ActionBody)(int parameters[],
31     struct Object* participants[],
32     Mask* unchanged);
33
34 #define ap_participant_changed(num, unchanged)\
35     *(unchanged) = (*(unchanged) & ~(1 << (num)));
36
37 typedef struct ActionTimings {
38     unsigned rate;
39     unsigned relative_deadline;
```

```
40      unsigned local_timeouts[MAX_PARTICIPANTS];
41      unsigned local_deadlines[MAX_PARTICIPANTS];
42      bool one_shot[MAX_PARTICIPANTS];
43  } ActionTimings;
44
45  typedef ActionTimings (*TimingFunc)(int parameters[]);
46
47  unsigned ap_add_action(ActionBody body,
48      CommonGuardFunc common,
49      unsigned participant_count,
50      unsigned parameter_count,
51      int parameters[]);
52
53  struct ObjectClass* ap_create_object_class(char name[],
54      size_t data_size,
55      ObjectConstructor constructor,
56      ObjectDestructor destructor,
57      ObjectCopyConstructor copy_constructor);
58
59  struct Object* ap_add_new_object(struct ObjectClass* cls,
60      unsigned action_id,
61      LocalGuardFunc local);
62
63  void ap_add_existing_object(struct Object* obj,
64      unsigned action_id,
65      LocalGuardFunc local);
66
67  void ap_add_static_timings(unsigned action_id, ActionTimings timings);
68
69  void ap_add_dynamic_timings(unsigned action_id, TimingFunc fun);
70
71  void ap_remove_participant(struct Object* obj, unsigned action_id);
72
73  void ap_delete_object(struct Object* obj);
74
75  void ap_remove_action(unsigned action_id);
76
77  void ap_log_message(const char* message, ...);
78
79  void *ap_object_data(struct Object* object);
80
81  unsigned ap_object_id(struct Object* object);
82
83  void ap_add_user_input(unsigned action_id,
```

```
84      size_t buffer_size,
85      const char* message);
86
87  void ap_reset_input(unsigned action_id, struct Object* input_object);
88
89  #endif
```

LIITE B. VUORONTAJAN JÄRJESTELMÄKUTSURAJAPINTA

```
1  #define ACTION_STATE_NEW 1
2  #define ACTION_STATE_ACTIVE 2
3  #define ACTION_STATE_INACTIVE 4
4  #define ACTION_STATE_RUNNING 8
5
6  long sys_get_action_id(void);
7
8  long sys_activate_action(unsigned id, int state);
9
10 long sys_deactivate_action(unsigned id, int state);
11
12 long sys_remove_action(unsigned id, int state)
13
14 long sys_add_object(unsigned object_id, unsigned action_id)
15
16 long sys_remove_object(unsigned object_id, unsigned action_id)
17
18 long sys_get_running_action()
19
20 long sys_peek_runqueue()
21
22 long sys_set_wait_time(unsigned action_id, unsigned wait_time)
23
24 long sys_request_input(unsigned action_id, char* buffer,
25     size_t buffer_size, const char* message, size_t message_size,
26     int* flag_addr);
27
28 long sys_register_approc(pid_t tid)
29
30 long sys_unregister_approc(pid_t tid, int error)
```