Agenttialusta moniaistisiin käyttöliittymiin

Rami Saarinen

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos/tko
Pro gradu -tutkielma
Ohjaaja: Roope Raisamo
9.10.2006

Tampereen yliopisto
Tietojenkäsittelytieteiden laitos/tko
Rami Saarinen: Agenttialusta moniaistisiin käyttöliittymiin
Pro gradu -tutkielma, 88 sivua
Lokakuu 2006

Tiivistelmä

Teknologian kehittyessä on keksitty uusia laitteita ja lähestymistapoja sovellusten rakentamiseen. Tietoverkkojen yleistymisen ansiosta hajautetut järjestelmät ovat olleet kiivaan kehityksen alaisena. On myös syntynyt uusia ohjelmointimalleja sovellusten rakentamiseen. Yksi näistä, agenttiohjelmointi, jakaa sovelluksen toiminnan pieniksi itsenäisiksi komponenteiksi, jotka keskustelevat toistensa kanssa. Agenttiohjelmoinnin keskeisin ajatus on kapseloida toiminnallisuus pieniin osiin ja hajauttaa osat yhdelle tai useammalle tietokoneelle.

Vuosien varrella on havaittu että nykyaikaiset graafiset käyttöliittymät rajoittavat ihmisen interaktiomahdollisuuksia ja jättävät useita erityistarpeita omaavia käyttäjäryhmiä, kuten sokeat ja liikunnallisesti rajoittuneet, tietotekniikan piiristä pois. Vaihtoehtoisia laitteita ja tietokoneen ohjaustapoja on kehitetty aktiivisesti. Erityisesti puheentunnistus ja katseohjaus ovat nousseet keskeisiksi tutkimuksen kohteiksi erityisryhmien käyttöliittymissä. Perinteisten vaihtoehtoisten syöte- ja palautemekanismien rinnalle on tullut myös tuntopalaute.

Tässä tutkielmassa esitellään Suomen Akatemian rahoittamassa PROAGENTS-hankkeessa rakennettu kolmiulotteinen opetusohjelma sokeille lapsille. Ohjelman teemoja olivat sokeille hankalat käsitteet: aurinkokunta, Maa, Maan suhde aurinkoon ja vuodenajat, maan kerrokset ja ilmakehä. Puuttuva aisti korvattiin käyttämällä sekä tunto- että äänipalautetta. Opetusohjelma rakentuu projektissa rakennetun agenttialustan päälle, joka myös kuvataan tässä tutkielmassa. Lopuksi esitetään, miten agenttialustaa voitaisiin laajentaa siten, että se toimisi tehokkaana pohjana moniaististen sovellusten rakentamiseen.

Sisällys

1 Johdanto

Tietoverkkojen yleistyminen ja nopeutuminen on tuonut uusia mahdollisuuksia toteuttaa ohjelmistoja ja antanut mahdollisuuksia olemassa olevien ohjelmistokehityksen ongelmien ratkaisemisen. Tietoverkkojen yleistymisen ansiosta hajautetut järjestelmät ovat saaneet huomiota enenevissä määrin. Hajautetut ohjelmat toimivat usealla eri tietokoneella ja ohjelman eri osat ovat yhteydessä yhteisen kommunikaatiokanavan kautta. Hajautettujen järjestelmien etuna on prosessorikuorman jakaminen, suoritusajan nopeutuminen ja sovelluksen toiminnan jakaminen pienempiin osiin.

Hajautettuihin järjestelmiin perustuva uusi ohjelmointiparadigma, agenttiohjelmointi, syntyi 1990-luvun alkupuolella [Shoham, 1993]. Agenttiohjelmoinnissa keskitytään hajauttamaan sovelluksen toiminnallisuus pieniin itsenäisesti toimiviin osiin, agentteihin. Agentit voivat kommunikoida toisten agenttien kanssa. Ne voivat myös siirtyä tietokoneelta toiselle. Agenttiohjelmoinnin yleistyessä on rakennettu useita agenttialustoja, jotka mahdollistavat agenttien vakiintuneet rakenteet ja agenttien helpon käyttöönoton [Bellifemine *et al.*, 1999, Bigus *et al.*, 2002, Martin *et al.*, 1999].

Tietoverkkojen yleistyessä myös muu tietotekniikka on jatkanut kehityskulkuaan. Lähiverkoista on tullut heterogeenisia käyttöjärjestelmien, käytettävien ohjelmien ja ohjelmointikielten mukaan. Laitepuolellakin on edistytty. Tutut näppäimistö ja hiiri ovat saaneet täydennystä mm. laitteista, jotka tuottavat aidon tuntuista tuntopalautetta [SensAble Technologies Inc, 2006].

Vaihtoehtoiset syöte- ja palautelaitteet avaavat ovet tietotekniikan laajemmalle käytölle ja tuovat tietotekniikan sellaisten erityisryhmien ulottuville, kuten sokeat, joiden tietokoneen käyttö muuten olisi kovin rajallista. Viimeaikaisissa tutkimuksissa on pyritty pois tyypillisistä graafisista käyttöliittymistä kohti käyttöliittymiä, jotka hyödyntävät kokonaisvaltaisemmin ihmisen aisteja ja ihmiselle luontevia syötetapoja [Nigay & Coutaz, 1993, Schomaker *et al.*, 1995].

Nämä moniaistiset käyttöliittymät ovat normaaleja graafisia käyttöliittymiä monimutkaisempia ja ne vaativat huomattavasti enemmän resursseja. Hajautettuja järjestelmiä on käytettykin usein moniaististen käyttöliittymien toteutuksessa, koska niiden käyttäminen jakaa prosessorikuormaa useammalle koneelle samalla hajauttaen sovelluksen toiminnan helpommin hallittaviin osiin. Hajautetun rakenteen ja valmiin infrastruktuurin vuoksi agenttialustat ovat oivia pohjia moniaististen sovellusten rakentamiseen [Martin *et al.*, 1998].

Suomen Akatemian rahoittamassa PROAGENTS-hankkeessa Tampereen Yli-

opistolla rakennettiin kolmiulotteinen opetusohjelma sokeille lapsille. Ohjelman teemoja olivat sokeille hankalat käsitteet: Aurinkokunta, Maa, Maan suhde aurinkoon ja vuodenajat, maan kerrokset ja ilmakehä. Koska pääkohderyhmä oli sokeat lapset, keskityttiin puuttuvan aistin korvaamiseen muiden aistien, kuten tunto- ja kuuloaistin, avulla. Sovellusta voivat käyttää täysipainoisesti myös näkevät lapset, jotka hyötyvät moniaistisesta palautteesta kolmen aistin välityksellä.

Projektin pidemmän ajan päämääränä on mahdollistaa sokeiden ja näkevien lapsien yhteistyö jaetussa virtuaaliympäristössä, missä puuttuvia aisteja voitaisiin korvata muilla aistipalautteilla. Hankkeen yhteydessä rakennettiin agenttialusta, koska haluttiin luoda pohja pidemmän ajan päämäärän toteutumiselle, ja koska haluttiin jakaa suuret resurssivaatimukset useammalle koneelle. Alusta suunniteltiin käytettäväksi heterogeenisessa lähiverkossa, johon voi olla liitettynä erilaisia koneita ja käyttöjärjestelmiä [Saarinen *et al.*, 2005]. Tässä tutkielmassa esittelen agenttialustan ja itse opetusohjelman.

Luvuissa 2 ja 3 käsittelen tutkielman kannalta tärkeitä perusasioita. Luvussa 2 käsittelen agenttiteknologiaa ja siihen liittyviä käsitteitä. Tarkennan aluksi agenttikäsitettä. Sen jälkeen käyn läpi aihealueen yleisiä standardeja ja vertailen eri agenttialustoja. Luvussa 3 vuorostaan esittelen moniaisiset käyttöliittymät ja moniaistisuuteen liittyvät aihepiirit. Käsittelen aluksi moniaistisuuden käsitettä ja sen jälkeen esitän ohjelmistomallin moniaististen sovellusten tekemiseen.

Luvussa 4 käsittelen synkronointia, joka tulee erittäin tärkeäksi hajautetuissa moniaistisissa ohjelmistoissa. Erittelen aluksi aiheen perusongelmia ja teorioita. Luvussa käsitellään mm. erilaisia synkronointitapahtumia ja tarkastellaan eri modaliteettien synkronoinnissa sallittuja viiveitä, joiden yli mentäessä epäsynkronia on ihmisaistein havaittavissa. Lopuksi esittelen kolme erilaista synkronointiongelman ratkaisumallia ja pohdin synkronointiin liittyviä erityisongelmia.

Luvussa 5 esitän PROAGENTS-projektin yhteydessä rakennetun agenttialustan arkkitehtuurin ja toiminnallisuuden. Agenttialusta on hyvin samankaltainen luvussa 2 esiteltyjen alustojen kanssa ja noudattaa keskitettyä mallia, jossa viestintä pääosin kulkee jaetun viestipalvelimen kautta. Luvun lopuksi esitän lyhyen esimerkin agenttien toiminnasta.

Luvussa 6 vuorostaan kuvailen PROAGENTS-projektin varsinaisen sovelluksen, joka rakennettiin luvun 5 agenttialustan päälle. Sovellus on rakennettu sekä sokeille että näkeville lapsille erityisesti sokeille vaikeiden asioiden opiskeluun.

Luvussa 7 arvioin sekä rakennettua agenttiarkkitehtuuria että projektissa rakennettua sovellusta. Erottelen sekä alustan että sovelluksen heikkouksia ja vahvuuksia ja pohdin niiden vaikutuksia alustaan ja sovellukseen.

Lopulta luvussa 8 paneudun kahteen suurimpaan edellisessä kappaleessa löydettyyn ongelmaan: moniaistisuuden tehostamiseen ja synkronoinnin integroimiseen agenttipohjaiseen järjestelmään. Luvun lopuksi esitän suunnitelman, jossa yhdistän nämä kaksi esitettyä ratkaisua agenttialustaksi, joka tukee sekä modaliteettien yhdistämistä ja yhteistulkintaa että myös palautteiden antamista useampaa aistikanavaa hyväksikäyttäen. Luvussa 9 on tutkielman yhteenveto.

2 Agentit

Agentti voidaan määritellä useilla toisistaan poikkeavilla tavoilla. Termi esiintyy monissa eri merkityksissä koneautomatiikasta tietokantateknologioihin, eikä sille enää löydy yksiselitteistä määritelmää. Shoham käsittelee agenttiperustaisen ohjelmoinnin artikkeleissaan [Shoham, 1993, Shoham, 1997] agentin olemusta ja terminologiaa. Hän toteaa että agenttien määrittely vaihtelee lähtien tiukan formaaleista kuvauksista löyhiin sanallisiin kuvauksiin. Perimmältäänhän termi agentti tarkoittaa tahoa, joka toimii jonkin toisen tahon puolesta. Shoham väittää että tämän perimmäisen määreen ovat ainakin osittain unohtaneet jopa tekoälyn tutkijat, ellei oteta huomioon pientä joukkoa ns. älykkäitä käyttöliittymiä (intelligent user interfaces) tutkivia tutkijoita. Hän katsoo, että tekoälyn parissa puuhailevat agenteista puhuvat tahot voivat olla samaa mieltä agentin olemuksesta vain hyvin suppeassa mielessä: agentti on jatkuvasti toiminnassa oleva itsenäinen toimija, joka saattaa jakaa saman ympäristön muiden agenttien kanssa.

Shoham itse määrittelee agentin entiteetiksi, jonka tilan voidaan katsoa koostuvan mentaalisista tekijöistä, kuten uskomuksista, kyvyistä, valinnoista ja sitoumuksista. Tämä määrittely ei ole niin suppea kuin se aluksi näyttää, vaan Shohamin ajatus on, että minkä tahansa ohjelman tai laitteen voi katsoa olevan tämän määreen mukainen agentti; yksinkertaisen valokytkimen tila voidaan nähdä mentaalitilan omaavana agenttina, mutta sen agentiksi määritteleminen ei anna meille uusia työkaluja entiteetin toiminnan ymmärtämiseksi. Valokytkimen tapauksessa meillä on sille paljon tehokkaampi ja selkeämpi mekaaninen malli. Monimutkaisten ja ehkä jossain määrin ennalta arvaamattomien entiteettien – kuten robottien, ihmisten ja vaikkapa käyttöjärjestelmien – toiminnan ymmärtäminen voi olla helpompaa, kun niiden toimintaa tutkaillaan mentaalitilojen avulla. Mentaalitilojen käyttö on siis työkalu, joiden avulla monimutkaista toimintaa voidaan helpommin selittää.

Bradshaw antaa agentille toisen klassisen määritelmän. Agentilla on hänen mukaansa seuraavia ominaisuuksia: autonomisuus, reaktiivisuus, pysyvyys, persoonallisuus, kommunikointikyky, yhteistyökyky, päättelykyky, mukautuvuus ja liikkuvuus [Bradshaw, 1997]. Agentti on siis itsenäinen, omaa toimintaa ohjaava ja muiden agenttien kanssa kommunikoiva jatkuvasti käynnissä oleva ohjelma. Agentin katsotaan omaavan jonkinlaisia keinoja tarkkailla ja manipuloida ympäristöään, oli kyseessä sitten lämpöpatterin termostaattiagentti tai vaikkapa satojen agenttien yhteisö jollakin erityisellä agenttialustalla. Tässä tutkielmassa agentilla tarkoitetaan Bradshaw:n kuvauksen mukaisia entiteettejä.

Agentit voivat olla toiminnaltaan sekä reaktiivisia että proaktiivisia [Woolridge, 2000]. Reaktiivinen agentti kykenee tarkkailemaan ympäristöään ja sieltä tulevia ärsykkeitä ja reagoimaan niihin jollain tavalla. Agentteja, jotka yrittävät ennakoida ja ennustaa tulevia tilanteita ja ohjaavat toimintaansa ennustusten mukaan, kutsutaan proaktiivisiksi agenteiksi. Proaktiivisuudeksi katsotaan myös agentin kyky toimia päämääräsuuntautuneesti, eli agentilla on jokin päämäärä tai aikomus, jota se pyrkii noudattamaan. Proaktiivisuutta on myös kyky hyödyntää sattumuksia [Bradshaw, 1997]. Vähänkin hyödyllinen proaktiivinen agentti on aina myös jollain asteella reaktiivinen, kun taas reaktiivinen agentti ei välttämättä ole proaktiivinen.

2.1 Agenttien kommunikointikieliä ja standardeja

Verkkoliikenteen, ohjelmointikielten ja -paradigmojen ja hajautettujen järjestelmien kehittyessä on laadittu useita standardeja laite- ja ohjelmistoyhteensopivuuden aikaansaamiseksi. Nykyiset agenttialustat jossain määrin nojaavat tai ainakin ottavat kantaa joihinkin seuraavaksi esiteltäviin standardeihin.

2.1.1 OMG CORBA

CORBA (Common Object Request Broker Architecture) on OMG:n standardi hajautetuille järjestelmille [OMG CORBA, 2006]. CORBA:a käyttävä ohjelma voi operoida verkon yli toisen CORBA:a tukevan ohjelman kanssa. CORBA pyrkii yleistämään ohjelmien välisen liikennöinnin niin, että toteutuksen ohjelmointikieli tai alla oleva käyttöjärjestelmä ei vaikuta prosessiin. Ohjelmien välinen kommunikaatio tapahtuu IIOP-protokollalla (Internet Inter-ORB Protocol).

CORBA:n ohjelmakomponentit voidaan jakaa kahteen ryhmään: *palvelun toteuttajat* toteuttavat tietyn palvelun, mitä *asiakkaat* voivat käyttää. Palvelun toteuttajalle määritellään CORBA-rajapinta tarjottavista palveluista OMG IDL -kielen (Interface Definition Language) avulla. Esimerkissä 2.1 määritellään rajapinta yrityksen osakkeen hinnan kysymiseksi. Jokaiselle palvelun parametrille määrätään, välitetäänkö parametrin arvo palvelun tarjoajalle (in), laittaako palvelun tarjoaja arvon parametriin (out) vai käytetäänkö parametria sekä tiedon viemiseen että tuomiseen (inout).

IDL-määrittely ajetaan IDL-kääntäjän läpi ja tämä luo sekä rajapintatiedoston asiakkaan käytettäväksi että luurangon palvelun tarjoajan toteuttamiseksi. Tässä esimerkissä C++:lla toteutettu asiakas laajennetaan esimerkissä 2.2 olevan rajapinnan mukaan.

Esimerkki 2.1: OMG IDL -määrittely

```
interface stock_query{
  int getStock(in string company);
}
```

Esimerkki 2.2: Asiakkaan rajapinta

```
class stock_query: public virtual CORBA::Object
{
  static stock_query_Ref _bind(...);
  virtual int getStock(CORBA::String company);
}
```

Rajapinnan _bind operaatiota käytetään asiakkaan yhdistämiseen johonkin tiettyyn palvelun tarjoavaan ohjelmaan. Esimerkissä 2.3 yhdistetään asiakas käyttämään cs.uta.fi-koneella olevaa palvelun tarjoajaa.

Esimerkki 2.3: Asiakkaan ja palvelun tarjoajan yhdistäminen

```
stock_query* x =
          stock_query::_bind("oma_olio", "foo@cs.uta.fi");
x->getStock("ACME");
```

IDL-kääntäjä tekee palvelun tarjoajalle tarjottavat rajapinnat sisältävän rangon (stock_query_sk esimerkissä 2.4), jota laajentamalla ohjelmoija voi toteuttaa mainitut rajapinnat (stock_query_impl esimerkissä 2.4).

Esimerkki 2.4: Palvelun tarjoajan rajapinta

```
class stock_query_sk: public virtual stock_query{
  virtual int getStock  (CORBA::String company) = 0;
 ...
};

class stock_query_impl: public stock_query_sk{
  int getStock(CORBA::String company);
};
```

2.1.2 KQML ja KIF

Yhdysvaltojen puolustusvoimien ylläpitämän DARPA-tutkimusjärjestön (Defense Advanced Research Projects Agency) tukema Knowledge Sharing Effort on poikinut mm. standardit KIF ja KQML [Patil *et al.*, 1992]. KIF (Knowledge Information Format) kuvaa yhteisen kielen eri tahojen väliselle tiedon jakamiselle ja kuvaamiselle. KIF perustuu ensimmäisen asteen predikaattilogiikkaan. KQML [Finin *et al.*, 1992, Finin *et al.*, 1994] vuorostaan on kolmikerroksinen kommunikointiprotokolla, joka sisältää kommunikaatio-, , viesti- ja sisältötason. Kommunikaatiotasossa määritellään viestin alkeisominaisuuksia, kuten lähettäjän ja vastaanottajan tiedot. Viestitasolla kuvataan mm. se mikä kommunikatiotapahtuma on kyseessä. Sisällössä vuorostaan on itse välitettävä informaatio jollakin formaalilla tavalla kuvattuna (esimerkiksi juuri KIF). Myöhemmät sovellukset ja standardit, kuten FIPA:n ACL omaavat paljon yhteistä KQML:n kanssa.

Esimerkissä 2.5 on esimerkki tyypillisestä KQML-protokollasta, missä agentti pyytää tietoa toiselta agentilta. Välitettävä kysymys on KIF-muodossa, jossa Agent1-agentti pyytää Agent2-agenttia hankkimaan ACME-yhtiön osakkeiden hinnan.

Esimerkki 2.5: KQML-viesti

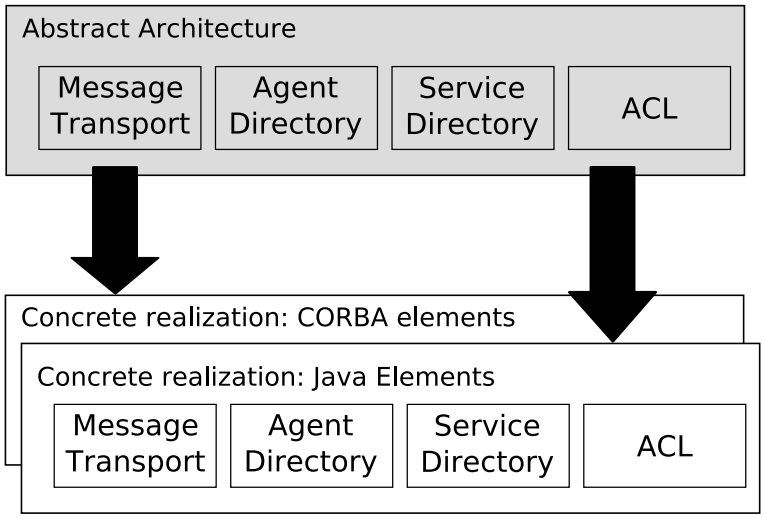
```
(ask
   :sender Agent1
   :receiver Agent2
   :language KIF
   :ontology Task-ontology
   :content (GetStockPrice ACME)
)
```

2.1.3 FIPA

FIPA:n (Foundation of Intelligent Physical Agents) tarkoituksena oli laatia standardit agenttijärjestelmien väliselle kommunikaatiolle ja täten se ei ota kantaa agentin, tai edes agenttialustan, sisäiseen rakenteeseen. FIPA:n ensimmäinen virallinen spesifikaatio julkaistiin vuonna 1997 [FIPA, 2006a, FIPA, 2006b, FIPA, 2006c], johon itsensä FIPA-yhteensopiviksi ilmoittavat alustat yleensä viittaavat. Tosin vuonna 1998 ilmestyi uusi spesifikaatio ja vuonna 2000 ilmestyi FIPA:n Abstraktin Arkkitehtuurin määrittely [FIPA, 2006d]. FIPA määrittelee vuosien

1997 ja 1998 määrittelyt vanhentuneiksi. Vuoden 2000 määrittely poikkeaa aiemmista määrittelyistä ja uusilta FIPA-alustoilta vaaditaan yhteensopivuutta vuoden 2000 määrittelylle.

Abstrakti Arkkitehtuuri [FIPA, 2006d] sai alkunsa vuonna 1999, kun FIPA päätti omaksua holistisen näkökohdan agenttialustan määrittelyyn. Sen sijaan, että aikaisempaa määritelmää olisi jatkettu, päätettiinkin edetä tarpeellisten arkkitehtuurin elementtien tunnistamisella ja yleisten, jaettujen, ominaisuuksien kartoituksella. Laadittiin Abstrakti Arkkitehtuuri, mikä toimii suuntaa-antavana spesifikaationa konkreettista agenttialustaa luodessa. Määritelmässä kuvataan mm. useita pakollisia ominaisuuksia ja palveluja, jotka konkreettisen alustan tulee toteuttaa ollakseen Abstraktin Arkkitehtuurin mukainen (kuva 2.1 [FIPA, 2006d]). Kuten aina, FIPA keskittyy agenttien ja agenttialustojen yhteistoiminnan mahdollistamiseen, eikä se täten ota kantaa itse agenttien tai alustan rakenteeseen. Abstrakti Arkkitehtuuri käyttää hyväkseen aiempia määrittelyjä. Esimerkiksi FIPA97:n ACL ja kommunikointiprotokollat ovat mukana määrittelyssä.



Kuva 2.1 Abstraktin Arkkitehtuurin osat

Abstraktissa Arkkitehtuurissa jokaisen agenttialustan on toteutettava agenttien hakemistopalvelu. Hakemistopalvelun tehtävänä on ylläpitää tietoa agenteista ja niiden ominaisuuksista. Kirjautuessaan palvelimelle, juureen, agentti välittää ainutlaatuisen nimensä ja agentin paikallistimen. Paikallistin sisältää yhden tai useamman viestinvälityskuvauksen, joka koostuu viestinvälitysmekanismin tyypistä, agentin osoitteesta ja mahdollisesti mekanismin lisäominaisuuksista. Tämän lisäksi agentin kuvaus voi sisältää esimerkiksi tietoa agentin tarjoa-

mista palveluista tai agentin käytön rajoituksista. Agentti voi muuttaa tietojaan myös ajon aikana.

Agentit voivat etsiä toisia agentteja agenttien hakemistopalvelusta. Agentin ei ole pakko kirjautua hakemistopalveluun, vaan agentti voi esimerkiksi tarjota paikallistintaan yksityisesti suoraan toiselle agentille.

Palvelujen hakemistopalvelussa pidetään kirjaa mahdollisista palveluista, joita on tarjolla agenttiympäristössä. Siinä missä agenttien hakupalvelu tarjoaa keinot löytää tiettyjä agentteja, antaa palvelujen hakemistopalvelu samat mahdollisuuden palvelujen suhteen.

Agentit ja toiset palvelut voivat suorittaa hakuja palveluhakemistoon. Palveluhakemiston palvelut ovat erillisiä sovellustason palveluita ja agenttien ei odoteta kirjaavan omia palveluitaan palveluhakemistoon. Palvelulle määritellään sen ainutlaatuinen nimi, tyyppi ja paikallistin. Palvelua ei ole pakko ilmoittaa palveluhakemistolle. Myös palvelun määrittelyä voidaan muokata ajon aikana.

Käynnistyessään agentille pitää antaa palveluiden keskitetyn palvelimen, juuren, osoite. Juuren avulla agentti pääsee käsiksi muihin agentteihin ja palveluihin.

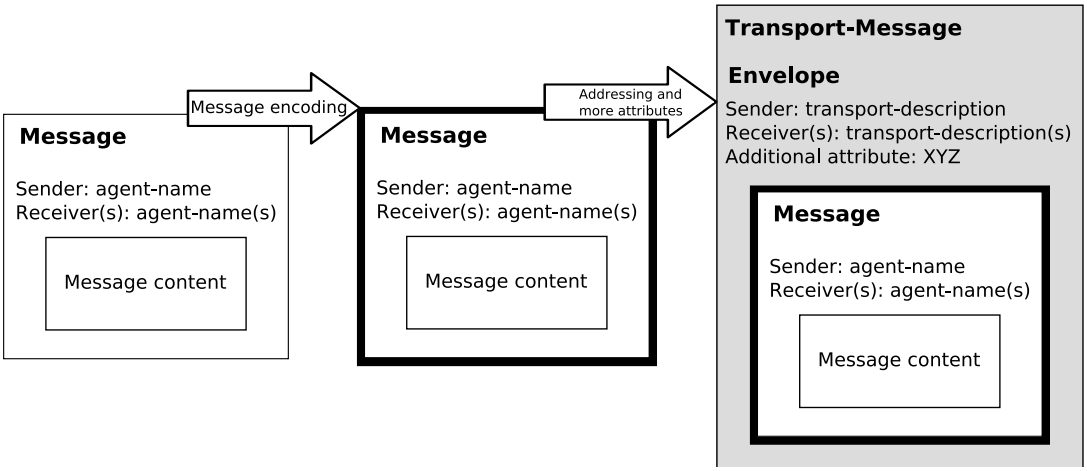
Agenttien välinen viestintä tapahtuu FIPA ACL -kielellä. ACL-viesti sisältää mm. lähettäjän nimen, vastaaottajan tai vastaanottajien nimet, viestin sisällön ontologiakuvauksen ja itse viestin sisällön. Esimerkissä 2.6 on vastaava ACME-yhtiön osakekysely XML-muotoisena FIPA ACL:nä. Viestin sisällön kielenä käytetään tässä esimerkissä KIF:iä.

Esimerkki 2.6: Esimerkki FIPA ACL -viestistä XML-muodossa

```
<fipa-message act="query-ref">
  <sender> agent-name </sender>
  <receiver> agent-name </receiver>
  <content> (GetStockPrice ACME) </content>
  <language> KIF </language>
  <encoding> encoding </encoding>
  <ontology> ontology </ontology>
  <protocol> protocol </protocol>
  <reply-with> identifier </reply-with>
  <in-reply-to> identifier </in-reply-to>
  <reply-by> time </reply-by>
  <reply-to> agent-name </reply-to>
  <conversation-id> ID </conversation-id>
```

Lähettävällä agentilla on vastaanottavan agentin viestinvälityskuvaukset, joi-

ta agentti käyttää viestin muuttamiseen lähetyskelpoiseksi. Tämä tapahtuu pyytämällä koodauspalvelua muodostamaan viestistä viestinvälitystapaan nähden asianmukaisella koodauksella varustettu paketti. Tämän jälkeen pakettiin liitetään kirjekuori, joka sisältää mm. tietoa käytetystä koodaustavasta ja lähettäjän ja vastaanottajan viestinvälityskuvaukset. Agentti antaa täten muodostetun viestin viestinkuljetuspalvelun hoidettavaksi (kuva 2.2 [FIPA, 2006d]).



Kuva 2.2 Viestin muodostus Abstraktissa Arkkitehtuurissa

Viestin varsinaiseen sisältöön FIPA ei ota kantaa, mutta määrää sisällön olevan jonkin sisältökielen, esimerkiksi FIPA-SL:n, FIPA-RD:n, FIPA-KIF:n tai FIPA-CCL:n, mukainen. Agenttialustan tulee toteuttaa koodauspalvelu, joka vastaa agenttien välisten viestien muuttamisesta ja muutosten purkamisesta valitun viestintätavan mukaan. Koodauspalvelu voi myös tarjota viestien salaus- ja varmennuskeinoja.

Viestinkuljetuspalvelu toimittaa viestejä agentilta toisille agenteille. Palvelun ei välttämättä tarvitse olla erillinen, vaan esimerkiksi vain yhtä kuljetustapaa tukevassa alustassa voidaan käyttää käyttöjärjestelmän valmiita rakenteita, kunhan palvelun perusominaisuudet toteutetaan.

2.2 Nykyiset agenttijärjestelmät

Järjestelmät, jotka sisältävät monia toistensa kanssa kommunikoivia agentteja, on rakennettu yleensä niin, että agentit sijaitsevat agenttialustalla eli agentit eivät ole itsenäisiä tietokoneella ajossa olevia prosesseja. Alustan käytöstä seuraa monia etuja. Se esimerkiksi mahdollistaa yleisen tavan kontrolloida agentin elinkaarta

(käynnistä, pysäytä, jatka ja lopeta). Alusta voi myös tarjota erilaisia, esimerkiksi viestintään tai eri kommunikointiprotokolliin liittyviä, palveluita agenteille.

Alusta on yleensä ajossa yhdellä koneella ja täten kaikki alustan agentit ovat myös suorituksessa samalla koneella. Jotkin alustat mahdollistavat agentin siirtymisen alustalta toiselle.

Agenttijärjestelmissä, joissa on useampia kuin yksi agentti ja joiden agenttien pitäisi keskustella keskenään, tulee ratkaista se ongelma, miten agentit löytävät toisensa ja kuinka ne voivat keskustella toistensa kanssa. Monissa järjestelmissä tämä on ratkaistu keskuspalvelimen avulla, joka osaa välittää agenttien viestejä, ja joka tarjoaa erilaisia palveluita agenttien ja alustojen käytettäväksi. Keskuspalvelin tietää agenttien sijainnin, niiden kommunikointiprotokollat ja tarjoaa esimerkiksi agentille keinon etsiä ja löytää alustan muut agentit.

Seuraavaksi esittelen kolme erilaista agenttialustaa, jotka ovat saavuttaneet vakiintuneen aseman agenttiohjelmoijien keskuudessa. Sekä JADE että ABLE ovat uudempia Javalla toteutettuja arkkitehtuureita, kun taas vanhemmalle alustalle, OAA:lle, on useita eri ohjelmointikielillä tehtyjä alustoja ja agentteja.

2.2.1 JADE

JADE [Bellifemine *et al.*, 1999, Bellifemine *et al.*, 2001] on Javalla toteutettu agenttialusta. Alustan 1. versio on FIPA 97 -yhteensopiva ja toteuttaa standardin vaatiman hakemistopalvelun (DF), ylläpitopalvelun (AMS) ja kommunikaatiopalvelun (ACC). Alustan uudempi versio on FIPA 2000 -yhteensopiva.

Yksittäinen alusta voi olla hajautettuna usealle koneelle lähiverkossa. Jokaisella alustaan kuuluvalla koneella on käynnistettynä agenttisäiliö, joka tarjoaa agenttien ajoympäristön, säiliön agenttien hallintapalvelut ja viestipalvelut alustan sisällä. Kaikki kommunikaatio käydään viestien välityksellä ja viesti on alustan sisäisessä liikenteessä Java-olio. Alustojen välisessä liikenteessä viesti muunnetaan FIPA ACL:ksi. Jokaisella säiliöllä on käytössään useita eri kommunikaatiokeinoja ja protokollia, joista se pyrkii käyttämään parasta mahdollista.

Erityinen säiliö, Front End, toimii yhteyspintana muihin agenttialustoihin ja vastaa myös hakemistopalveluista ja ylläpitopalveluista. Tavallisia säiliöitä voidaan lisätä ja poistaa alustasta heti kun Front End on käynnistynyt. Viestintä alustojen välillä käydään CORBA IIOP:lla. Alustan sisällä viestin välittämiseen käytetään Javan RMI:tä (Remote Method Invocation) [Java RMI, 2006] vastaanottavan agentin ollessa eri säiliössä kuin lähettävä agentti. Jos taas vastaanottaja ja lähettäjä ovat samassa säiliössä, käytetään Javan tapahtumaa. Tapahtuma

sisältää ACLMessage-olion.

JADE:ssa on valmiina muutamia erityisiä agentteja. Etätarkkailuagentti (Remote Monitoring Agent, RMA) on graafisen käyttöliittymän tarjoava agentti, jonka avulla alustan tilaa ja sisältöä voidaan tarkkailla ja kontrolloida. Nuuskija-agentilla (Sniffer) voidaan vuorostaan tarkkailla agenttien välistä liikennöintiä graafisesta käyttöliittymästä.

JADE:n agentit ovat tavallisia Java-luokkia. Agentti luodaan periyttämällä oma agentti JADE:n tarjoamasta abstraktista agentista. Jokainen agentti omaa yhden viestijonon, johon agentille tarkoitetut viestit saapuvat. Agentti ajetaan omassa säikeessään. Yleisenä sääntönä on, että agentilla saa olla vain yksi säie, vaikka mikään ei estä tämän säännön rikkomista. Säikeitä lisättäessä täytyy olla varovainen, sillä säikeiden runsas määrä saattaa johtaa tehokkuusongelmiin.

Säikeiden sijasta agentille määritellään useita käyttäytymistapoja (behaviours). Yksi käyttäytymistapa voi olla suorituksessa kerrallaan ja agentti vaihtaa ajettavaa käyttäytymistapaa automaattisesti. Agentti voi lisätä ja poistaa käyttäytymistapoja ajon aikana esimerkiksi uuden dialogin käymiseksi. Jokaisella agentilla on valmiina käyttäytymistavat agentin rekisteröintiin, rekisteröinnin purkamiseen ja viestien lähettämiseen.

Jokainen käyttäytymismalli määrittelee action()-metodin, jossa mallin toiminnallisuus sijaitsee. Käyttäytymismallit jakaantuvat kahteen eri luokaan: yksinkertaisiin ja yhdistettyihin käyttäytymismalleihin. Yksinkertaiset mallit määrittelevät vain yhdenlaisen toiminnallisuuden action()-metodissa. Yksinkertaiset käyttäytymismallit jakaantuvat vielä kahteen aliryhmään: kertaluonteiset ja sykliset käyttäytymismallit. Kertaluonteiset mallit suoritetaan vain kerran ja sitten ne poistetaan käyttäytymismallien joukosta. Syklisiä malleja toistetaan normaaliin tapaan.

Yhdistetyt mallit voivat sisältää useampia käyttäytymismalleja. Mallin luonne määrää, miten niiden sisältämiä malleja suoritetaan. Yhdistelmämallit jakaantuvat kolmeen aliryhmään. Ensinnäkin jaksoittaisen käyttäytymismallin lapsimallit suoritetaan peräkkäin. Rinnakkaisessa käyttäytymismallissa lapsimallit suoritetaan samanaikaisesti. Lisäksi on tilakonemalli, jonka avulla voidaan muodostaa tilakoneen kaltainen käyttäytymismalli, eli se on aina jossain tilassa ja tilojen välillä on siirtymiä. Siirtymä tilasta toiseen tapahtuu tiettyjen ehtojen toteuduttua.

2.2.2 OAA

OAA:n (Open Agent Architecture) pääsuunnitteluperiaattena on ollut agenttialustan mukauttaminen heterogeeniseen ympäristöön [Martin *et al.*, 1999]. Tämän lisäksi agenttialustaan haluttiin rakentaa joitakin liitutaulutekniikan (Blackboard), liikkuvien agenttien ja kommunikoivien agenttien vahvimpia puolia. Liitutaulutekniikoissa agentit jakavat yhteisen työtilan, liitutaulun, jonka avulla ne voivat tehdä yhteistyötä ja tiedon jakamista.

Arkkitehtuuria suunniteltaessa keskityttiin joustavuuteen sekä ajon aikaisilta ominaisuuksiltaan, sovelluskehityksen aikana että agenttiympäristön eri tahojen yhteistyöinteraktioiden rakenteessa. Arkkitehtuurissa haluttiin määrätä agenttien rakennetta sopivasti, muttei liikaa. Esimerkiksi KQML:n käyttäminen kommunikointiin ei vaatisi agentin rakenteelta juuri mitään. Toisaalta toisen merkittävän agenttimetodologian BDI:n [Rao & Georgeff, 1995] katsottiin pakottavan agentit rakenteeseen, joka oli joihinkin tarkoituksiin liian vaativa. Lopulta suunnittelijat päätyivät ratkaisuun, joka sijaitsee KQML:n ja BDI:n välimaastossa.

Käyttäjän osa on korostettu. Agenttialustan tulisi tarjota käyttäjälle luontevat tavat kanssakäymiseen useiden hajautettujen komponenttien kanssa, niin että hänen ei tarvitse tietää tarkemmin agenteista tai niiden ominaisuuksista. Käyttäjä katsotaan myös yhdeksi, joskin etuoikeutetuksi, agentiksi.

Agenttien välinen liikennöinti käydään OAA:n omalla kielellä ICL:llä (Inter-agent Communication Language), joka muistuttaa rakenteeltaan Prolog-kielen predikaattikutsua. Yleisin viestintämuoto on tapahtuma, jota käytetään aina agenttien aktiviteettien ilmaisusta agenttien väliseen kommunikointiin saakka. Yleensä tapahtuma nähdään päämääränä, joka tulisi saavuttaa.

Tapahtumatyypit ja niiden parametrit määrittelevät ICL:n keskusteluprotokollan, joka muistuttaa pitkälti Prolog-kielen rakenteita. Dialogiin liittyvien asioiden määrittely tapahtuu tapahtumatyypin ja parametrijoukon valinnalla. Parametreja on kahdenlaisia. Palauteparametreilla palvelua pyytävä taho voi saada tietoja siitä, miten annettu tavoite on suoritettu. Pyytäjä voi esimerkiksi saada tietoon, mitkä agentit hoitivat tavoitteen suoritusta ja kuinka kauan suoritus kesti. Neuvoa-antavat parametrit taas asettavat rajoitteita tai ohjeita jonkin tavoitteen suorittamiselle.

Tapahtuman sisältö voi koostua erilaisista tavoitteista, laukaisimista ja välitettävästä tiedosta (oli se missä muodossa tahansa). Sisältö on suunniteltu Prolog-kielen laajennukseksi hyödyntäen näin Prologin erityispiirteitä. Sisältönä voi olla myös ns. yhdistelmätavoite, joka on erillisten tavoitteiden joukko. Lopuksi jo-

kaisella alitavoitteella voi olla osoiteattribuutti ja alitavoitteeseen liittyvien parametrien joukko. Osoite tarkentaa yhden tai useamman agentin, joille alitavoitteen suoritus sallitaan.

OAA on perusrakenteeltaan keskitetty järjestelmä, jossa kaikki liikennöinti tapahtuu yhden tai useamman välittäjän (Facilitator) kautta. Välittäjän tehtävänä on koordinoida agenttien välistä kommunikointia ja yhteistyössä tapahtuvaa ongelman ratkaisua. Välittäjä pitää kirjaa myös agenttien tarjoamista palveluista. Lähettävän agentin ei välttämättä tarvitse tietää palvelua tarjoavasta agentista mitään, vaan välittäjän tehtävänä on löytää oikeat tahot agentin lähettämän pyynnön toteuttamiseksi.

Välittäjä pystyy myös käsittelemään yhdistelmätavoitteita. Yhdistelmätavoitteiden käsittelyssä välittäjä sekä delegoi osatavoitteita oikeille tahoille, optimoi osatavoitteita mahdollisimman tehokkasti ratkaistavaksi ja tulkitsee ratkaistujen osatavotteiden tulokset muodostaen niistä kokonaisuuden, joka palautetaan palvelua vaatineelle taholle.

Välittäjää käytetään usein myös agenttien välisenä globaalina tietovarastona esimerkiksi liitutaulutekniikan [Buschmann *et al.*, 1996] mahdollistamiseen. Liitutaulutekniikassa agentit voivat laittaa liitutaululle (Blackboard) ongelmia, tai siis ICL-rakenteita, toisten agenttien ratkaistavaksi. Palvelun tarjoajat voivat tutkia liitutaulua ja ratkaista siellä olevia ongelmia. Ratkaistuaan ongelman, agentti laittaa vastauksen liitutaululle, josta ongelman esittäjä sen hakee.

OAA-arkkitehtuurissa on yleisesti määritelty kolme erilaista agenttiluokkaa. Luokittelu tosin ei ole kuin neuvoa-antava, eikä sitä ole arkkitehtuuriin kiinnitetty. Sovellusagentit tarjoavat palveluiden joukkoja, kuten sähköposti-, puheentunnistus- ja matkanvarauspalveluja. Meta-agentit avustavat välittäjää toisten agenttien toimintojen koordinoinnissa välittäen sovellus- ja aihekohtaisia tietoja ja päättelytapoja. Viimeisenä on käyttöliittymäagentit, jotka toimivat mm. eri modalitteettien tulkinnassa ja yhdistämisessä.

Agentit eivät pelkästään käytä välittäjän palveluita hyväkseen vaan myös tarjoavat palveluita muiden agenttien käytettäväksi. Käynnistyessään agentti kirjaa tarjoamansa palvelunsa välittäjälle. Agentti voi lisätä, poistaa tai muokata palveluitaan suorituksen aikana. Agentti voi esimerkiksi määritellä palvelun yksityiseksi, jos se ei halua palvelua käytettävän muualta kuin agentista itsestään käsin.

Agentit, ja täten myös käyttäjät, voivat asettaa laukaisimia jonkin tietyn tapahtuman seuraamiseen. Laukaisimen voi asettaa itseensä, toiseen agenttiin tai välittäjään. Asettaessaan laukaisinta agentti määrittelee jonkin reunaehdon ja

tapahtuman, joka suoritetaan reunaehdon täytyttyä. Esimerkiksi käyttäjä voisi asettaa sähköpostiinsa laukaisimen, joka ilmoittaisi puhelimitse tietyn sähköpostin saapumisesta.

Laukaisimia on neljänlaisia. Kommunikaatiolaukaisin sallii minkä tahansa ulos- tai sisäänpäin menevän tapahtuman tarkkailun. Informaatiolaukaisimen avulla voidaan tarkkailla jotain tietovarastoa ja sen toimintaa. Sillä voidaan tarkkailla esimerkiksi jonkin tiedon lisäämistä, poistamista tai muuttamista. Tehtävälaukaisimen avulla agentti voi tarkkailla jotakin sisääntulevaa tapahtumaa määrittelemällä ICL:n mukaisen tavoitteen. Lopuksi ajastinlaukaisimet tarjoavat keinot tarkkailla ajallisia tilanteita esimerkiksi "klo. 15.00" tai "kolmen minuutin välein".

2.2.3 ABLE

IBM:n ABLE-agenttialusta (Agent Building and Learning Environment) [Bigus *et al.*, 2002, Meyer, 2006] on toteutettu JavaBean-teknologian avulla. ABLE:n agentin rakennetta ei ole erikoistettu helpottamaan jonkin tietyn agenttimallin rakentamista, vaan agentin koostumus ja rakenne on jätetty avoimeksi. Arkkitehtuuri koostuukin laajasta apukirjastosta, jossa on paljon yleisiä agenttien rakennuksessa tarvittavia komponentteja.

Agentteja ohjelmoidaan pääsääntöisesti laajentamalla AbleDefaultAgent luokkaa, joka tarjoaa oletustoteutuksen ABLE:n agenteille. Agentit sijaitsevat AbleBeanContainer-oliossa. Agentti itsekin on AbleBeanContainer, mistä seuraa se, että agentti voi sisältää toisia AbleBean olioita kuin myös toisia agenttejakin.

Agenttia suoritetaan omassa säikeessään, ja ne voivat olla yhteydessä toisiinsa kolmella eri tavalla. Ensinnäkin agentit voidaan kytkeä ketjuun siten, että toisen agentin prosessoima data ohjataan toiselle agentille prosessoitavaksi. Näin agentteja voidaan ketjuttaa vastaavalla tavalla kuin on kuvattu tietovuoarkkitehtuurissa (Pipes & Filters) [Buschmann *et al.*, 1996]. Toisekseen agentti voi ilmoittautua toisen agentin kuuntelijaksi tarkkailija-suunnittelumallin (Observer) tavoin [Buschmann *et al.*, 1996]. Agentit voivat myös jakaa joitakin ominaisuuksia. Tällöin ominaisuuden muuttuessa toisessa agentissa sen uusi arvo päivittyy myös toiseen agenttin. Metaforaltaan agenteilla katsotaan olevan sensoreita ja manipulaattoreita. Sensoreiden avulla agentti saa tietoa ympäröivästä maailmasta ja se toimii manipulaattoreiden avulla. Sensorit ja manipulaattorit ovat erillisiä olioita, jotka on liitetty agenttiin.

ABLE:n apukirjasto voidaan jakaa karkeasti kolmeen osaan: datakirjasto, op-

pimiskirjasto ja sääntökirjasto. Apukirjaston kaikki komponentit ovat täysiverisiä JavaBean-olioita.

Datakirjastoon kuuluvat kaikki ne komponentit, joita käytetään datan hankkimiseen, tulostamiseen tai käsittelyyn. Kirjastosta löytyvät mm. komponentit tiedostojen lukemiseen, tietokantojen käsittelyyn ja datan muokkaamiseen. Oppimiskirjasto vuorostaan koostuu koneoppimisessa käytettävistä komponenteista. Kirjastoon kuuluvat komponentit mm. neuroverkkojen-, itseohjautuvien karttojen- ja päätöspuiden luomiseen. Sääntökirjasto sisältää komponentit erilaisten sääntökoneiden luomiseen. Komponenttien avulla voi esimerkiksi rakentaa sekä eteen- että taaksepäin ketjuttavia sääntökoneita, sumealla logiikalla varustettuja sääntökoneita ja hahmontunnistuskoneita.

ABLE on täysin FIPA 97 -yhteensopiva ja tämän lisäksi tarjoaa joitakin erillisiä palveluita, kuten seuraavat:

- Käynnistyspalvelun ja ensimmäisen yhteyspalvelun, jotka helpottavat agenttialustan käynnistämistä, uusien agenttien luomista ja niiden ensimmäistä yhteyttä agenttialustaan.

- Nimeämispalvelun, joka antaa jokaiselle agentille ainutlaatuisen nimen.

- Tiedonvälityspalvelut, jotka tarjoavat useita eri protokollia ja mekanismeja agenttien väliseen kommunikointiin.

- Elinkaaripalvelun, joka tarjoaa mahdollisuuden lisätä ja poistaa agentteja ajon aikana. Ylikäyttäjä voi myös pysäyttää agentteja ja jatkaa pysäytettyjen agenttien suoritusta.

ABLE:n mukana tulee graafinen kehitysympäristö agenttien luomiseen ja agenttien välisen interaktion määrittelyyn. Lisäksi agenttialustalla on graafinen käyttöliittymä, josta näkee helposti järjestelmän tilan ja jonka avulla voi manipuloida agentteja helposti.

2.3 Agenttialustojen arviointia

Erityisesti Java-pohjaisia agenttialustoja on syntynyt tiheään tahtiin viime vuosien aikana. Syynä tähän lienee Javan helppo saatavuus, Javalla ohjelmoinnin helppous ja sen erityiset ominaisuudet kuten JavaBeans-teknologia, RMI [Java RMI, 2006] ja Reflection API [Java Reflection API, 2006]. Edellä esitellyistä kolmesta alustasta kaksi on tehty Javalla; Sekä JADE että ABLE ovat vakaalla

pohjalla seisovia menestyksekkäitä alustoja, jotka ovat vakiinnuttaneet asemansa agenttiohjelmoinnin piirissä. Vaikka päällepäin näyttääkin, että alustojen ratkaisut ovat kovin erilaisia, voidaan nopeasti huomata, että tietynlainen perustoiminnallisuus löytyy molemmista. Vaikka JADE:n agentti käyttää käyttäytymismalleja ja ABLE:n agentti valmiita JavaBean-komponentteja, periaate on kuitenkin sama: agentti koostetaan toisista komponenteista.

ABLE on hyvin monipuolinen ja varmasti myös yrityskäyttöön mainiosti sopivalta alusta, jonka perustoiminnallisuudessa on pyritty ottamaan huomioon tyypillisimmät agenttialustan tarpeet. Alustaan liittyvä apuohjelmisto ja työkalut ovat korkealuokkaisia ja monipuolisia.

JADE on hyvin selkeä ja helposti ylläpidettävä alusta. Toiminnallisuuden jakaminen käyttäytymismalleihin kapseloi ne mukavasti. Käyttäytymismallien käyttäminen toisaalta tarkoittaa myös, että uutta toiminnallisuutta lisättäessä on aina tehtävä uusi käyttäytymismallin aliluokka. JADE:n arkkitehtuurin rakenne on selkeä ja helposti ymmärrettävä.

OAA puolestaan erottuu joukosta monestakin syystä. Ensinnäkin OAA on pitkälti ohjelmointikielivapaa ja OAA-agentteja onkin tehty mm. Prologilla, Javalla ja C/C++:lla. OAA:n agenttien vuorovaikutus on lähtokohtaisesti aivan erilaista kuin aiemmin mainituissa alustoissa. Jokainen viesti on Prolog-tavoitetta vastaava rakenne, jossa voidaan välittää tehtävän tavoite, sen parametrit ja mahdollisesti myös paluuarvoina palautuvat parametrit. Jos tavoite koostuu useammasta osatavoitteesta, voidaan tavoite jakaa osiin ja osat ratkaista samanaikaisesti eri agenteilla. Niinkuin monessa muussakin agenttialustassa, voi OAA:n kompastuskiveksi nopeasti muodostua keskitetty liikennöinti välittäjän kautta.

Kaikille edellä mainituille alustoille on yhteistä keskitetyt palvelut viestien lähettämiseksi, toisten agenttien löytämiseksi ja elinkaaripalvelut agenttien toiminnan kontrolloimiseksi. Suurin osa eroista löytyykin agentin toteutuksesta.

Myöhemmin esittelen rakentamani agenttialustan. Ennen kuin aloin rakentamaan alustaa arvioin näiden edellämainittujen alustojen sopivuuden projektin puitteisiin. ABLE:n hylkäsin nopeasti, sillä vaikka se näyttikin ominaisuuksiltaan erittäin lupaavalta, se vaatii JavaBean-palvelimen, jonka toimintakuntoon saattaminen ja ylläpitäminen olisi ollut liiallinen rasite. Sekä ABLE:n että JADE:n jouduin hylkämään myös siksi, että projektissa käytettiin sekä kaupallista C++-kirjastoa tuntopalautelaitteen kontrollointiin että muita valmiita C/C++-pohjaisia kirjastoja agenttien toiminnan rakentamiseen. Vaikka Javassa on JNI (Java Native Interface), vaatisi sen käyttäminen jokaisen C++-kirjaston käärimistä JNI:n mukaan. Oli myös epäselvää, kuinka hyvin Java:n agentit pelaisivat

yhteen C++-pohjaisen tuntopalautelaitteen kanssa sitten, kun JNI-rajapinnat oltaisiin saatu valmiiksi. Tuntopalautelaite vaatii kaksiprosessorikoneen, jonka toinen prosessori pyhitetään tuntopalautteen antamiselle. Tuntopalauteen toteuttaminen vaatii paljon resursseja ja Javan lisääminen näin kriittiseen kohtaan voisi aiheuttaa häiriöitä tuntopalautteessa. Kuten seuraavassa luvussa havaitaan, on JADE:lla ollut suuri vaikutus oman agenttialustani rakenteeseen.

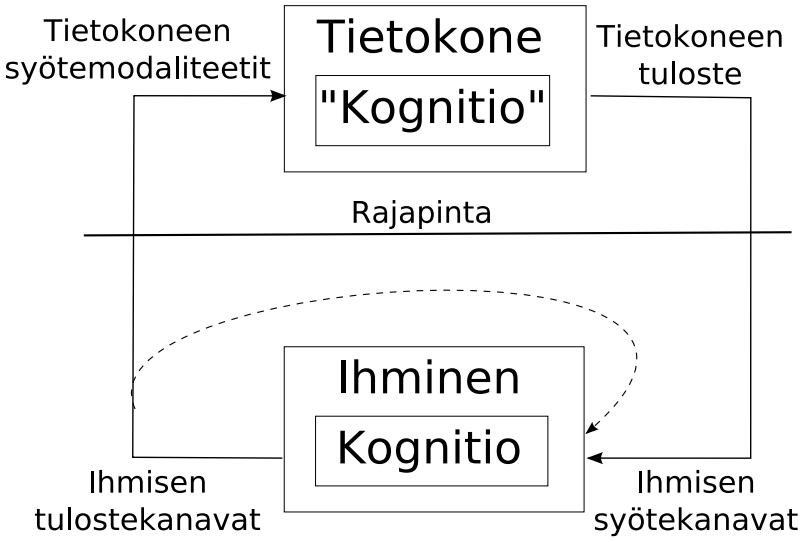
OAA vuorostaan, vaikka onkin vakaa ja monissa sovelluksissa käytetty, on perusteiltaan hyvin vanha ja olemukseltaan vaikealta ja tarpeettoman monimutkaiselta vaikuttava ohjelmisto. Projektiin nähden oli suotavaa, että agenttialusta olisi helposti toimintakuntoon saatettavissa ja uusien agenttien tekeminen ja järjestelmään liittäminen olisi mahdollisimman helppoa. Tulevaisuuden projekteja ajatellen olisi myös suotavaa, jos alustaa voitaisiin käyttää jatkossakin hyväksi. OAA:n Prolog-tyylinen ICL-protokolla vaikutti kryptiseltä ja vaikeasti ymmärrettävältä.

OAA:n päälle olisi voitu rakentaa toiminnallisuutta helpottavia agentteja, jotka käärisivät OAA:n toiminnallisuuden helpommin ymmärrettäviin rajapintoihin. Projektissa haluttiin kuitenkin säilyttää mahdollisimman suuri mahdollisuus vaikuttaa suoraan agentteihin ja itse infrastruktuuriin. Tuntopalautelaite ja sen tarjoama kolmiulotteinen haptinen virtuaalimaailma haluttiin integroida järjestelmään mahdollisimman saumattomasti.

3 Moniaistisuus

Ihmisen ja tietokoneen vuorovaikutusta tutkivat tahot pitävät nykyajan vallitsevia käyttöliittymiä, missä käyttäjä antaa syötettä hiirellä ja saa palautetta toimistaan visuaalisesti, liian rajoittavina. Buxton [1987] toteaa, että nykyajan tietokoneen käyttöliittymät eivät ole yhdenmukaisia niiden käyttäjien fyysisiin ominaisuuksiin nähden. Buxtonin mukaan tietokoneiden käyttöliittymissä käytetään hämmästyttävän vähän ihmisen aistikanavia hyväksi ja hän toteaakin että lähestulkoon kaikkien muiden ihmiselle tarkoitettujen laitteiden, aina autosta suihkuun saakka, käyttöliittymät ovat paremmin suunniteltuja ja ottavat ihmisen aistit paremmin huomioon.

Schomaker *et al.* [1995] tutkivat ihmisen ja tietokoneen vuorovaikutusta ja esittivät multimodaalisen vuorovaikutuksen taksonomian. Taksonomiassa on kaksi toimijaa, ihminen ja tietokone (kuva 3.1). Ihminen ja tietokone voivat jakaa tietoa erilaisten tiedonvälityskanavien avulla yhteisen rajapinnan kautta. Vuorovaikutusmallissa ihmisen tulostekanava ohjataan rajapinnan kautta tietokoneen syötemodaliteetteihin ja tietokoneen tuloste vuorostaan ohjataan rajapinnan kautta ihmisen syötekanaviin (aisteihin). Silmä-käsi-koordinaatiosta seuraa, että ihmisellä on myös yhteys omasta tulosteesta syötteeseensä; ihminen siis tarkkailee omaa toimintaansa.



Kuva 3.1 Ihmisen ja tietokoneen vuorovaikutuksen peruselementit

Ihmisen syötekanavina toimivat aistit, kun taas tulostekanavat ovat pääsääntöisesti kehon eri osien käyttöön perustuvia. Taulukossa 3.1 on listattu aistit,

aisteihin liittyvät aistielimet ja näitä vastaavat modaliteetit.

Aisti	**Aistielin**	**Modaliteetti**
Näköaisti	Silmät	Katse
Kuuloaisti	Korvat	Ääni
Tuntoaisti	Iho	Haptinen
Hajuaisti	Nenä	Olfaktorinen
Makuaisti	Kieli	Gustatoorinen
Tasapainoaisti	Tasapainoelin	Vestibulaarinen

Taulukko 3.1: Aistit, aistielimet ja modaliteetit

Multimodaaliset järjestelmät tukevat useamman (multi) aistin ja vuorovaikutuskanavan (modaliteetti) käyttöä. Palautetta käyttäjälle voidaan antaa ihmisen aistien avulla, ja ihminen voi vuorostaan ohjata järjestelmää eri toimiensa avulla. Tyypillisimmillään järjestelmää ohjataan käden motoriseen toimintaan pohjaavilla laitteilla, kuten hiirellä tai näppäimistöllä. Näiden lisäksi voidaan esimerkiksi käyttää ääntä tai katsetta. Multimodaalisuutta on myös syötteen antaminen useaa samaakin aistia hyödyntävää kanavaa apuna käyttäen; järjestelmää voidaan esimerkiksi ohjata kahdella hiirellä. Moniaistisuus on multimodaalisuuden osajoukko, joka sisältää aistien käytön, mutta poissulkee saman aistin samanaikaisen käytön. Tässä tutkielmassa keskitytään moniaistisuuteen.

Nigay ja Coutaz [1993] määrittelevät multimodaalisuudeksi sen, että järjestelmä pystyy kommunikoimaan käyttäjän kanssa usean eri kommunikaatiokanavan avulla ja on kykenevä erottamaan ja johtamaan kommunikaation tarkoituksen automaattisesti. Multimodaalisuus on kahdensuuntaista: järjestelmälle voidaan antaa komentoja usealla eri tavalla ja käyttäjälle tarjotaan informaatiota useampaa aistikanavaa apuna käyttäen.

Multimodaalisuudesta puhuttaessa puhutaan usein aistien fuusiosta ja fissiosta. Fissiolla tarkoitetaan esitettävän datan kuvaamista käyttäjälle eri palautetapojen avulla. Fuusiolla vuorostaan tarkoitetaan järjestelmän kykyä yhdistää eri kommunikaatiokanavien kautta saatuja havaintoja ja komentoja. Coutaz ja Nigay [1993] jakavat multimodaalisuuden neljään eri luokkaan fuusion tason ja modaliteettien käyttötavan mukaan:

- Modaliteettien käyttäminen peräkkäin fuusion yhdistäessä syötteet. Tämä kuvaa multimodaalisuutta, missä tehtäviä voidaan tehdä usealla modaliteetilla siten, että niiden pitää tapahtua peräkkäin. Fuusio yhdistää eri modaa-

lisyötteitä ketjuiksi. Syötteet siis ovat osa syötteiden ketjusta muodostuvaa käskyä.

- Modaliteettien käyttäminen peräkkäin ilman fuusiota kuvaa multimodaalisuutta, missä vain yhtä modaliteettia voidaan käyttää ekslusiivisesti kerrallaan. Eri modaliteetit liittyvät erillisiin tehtäviin.

- Modaliteettien käyttäminen rinnakkain fuusion yhdistäessä ne, tarkoittaa synegististä multimodaalisuutta, missä komentoja voidaan antaa samanaikaisesti usealla modaliteetilla ja niiden fuusiosta muodostuu yksittäinen käsky.

- Kun modaliteetteja käytetään rinnakkain ilman fuusiota, on kyseessä multimodaalisuus, missä komentoja voidaan antaa samanaikaisesti usealla modaliteetilla. Komennot itsessään ovat erillisiä ja muista komennoista riippumattomia.

Eri modaliteetteja voidaan käyttää toistensa tukemiseen ja komennon tulkinnan virhemarginaalin pienentämiseen. Multimodaalisissa järjestelmissä on usein myös modaliteetteja, jotka ovat dominantteja muihin modaliteetteihin nähden. Multimodaalisissa sovelluksissa keskitytään usein fuusioon.

Nykyaikaiset käyttöliittymät ovat yhä enenevissä määriin suoravaikutteisia. Suoravaikutteinen käyttöliittymä pyrkii olemaan mahdollisimman näkymätön ja pyrkii antamaan käyttäjälle mahdollisuuden manipuloida ohjelman elementtejä suoraan eri syötelaitteiden avulla. Esimerkiksi piirto-ohjelmassa käyttäjä voi suoraan liikuttaa ja muuttaa piirroksen elementin ominaisuuksia. Suoravaikutteisuus ja multimodaalisuus ovat vahvasti yhteensopivia käsitteitä. Siinä missä suoravaikutteisuus pyrkii tarjoamaan mahdollisimman luontevan rajapinnan ihmisen ja tietokoneen välille, luo multimodaalisuus siihen tekniset mahdollisuudet.

Eri modaliteettien käyttöä on kiivaasti tutkittu jo hyvän aikaa. Pääosa tutkimuksesta keskittyy puheen tunnistukseen ja katseella ohjaamiseen. Tuntopalaute on myös jossain määrin ollut tutkimuksen kohteena. Modaliteettien lisäksi on tutkittu vaihtoehtoisia käyttöliittymiä, esimerkiksi kosketeltavia käyttöliittymiä, lisättyä todellisuutta ja jokapaikan tietotekniikkaa. Myös erilaiset biopalauteet (aivokäyrä, lihasjännitykset, pulssi, biosähkö) ovat nousseet viime aikoina tutkimuksen kohteiksi.

3.1 Moniaistisuutta tukeva ohjelmistoarkkitehtuuri

PAC-Amodeus on agenttipohjainen arkkitehtuurimalli multimodaalisten sovellusten tekemiseen. Se yhdistää PAC-mallin [Coutaz, 1987] Arch-malliin [Bass *et al.*, 1992]. Siinä missä Arch kuvaa sovelluksen jakoa käsitteellisiin komponentteihin, keskittyy PAC-malli rinnakkaiseen toiminnallisuuteen ja agenttihierarkioihin. Seuraavaksi käsitellään Arch- ja PAC-mallit ja tämän jälkeen katsotaan, miten ne on yhdistetty PAC-Amodeukseksi.

3.1.1 Arch

1990-luvun alussa interaktiivisille ohjelmistoille suunniteltu Arch-malli [Bass *et al.*, 1992] ja siitä yleistetty Slinky-metamalli jakavat arkkitehtuurin osiin sen mukaan minkälaista dataa kussakin osassa käsitellään. Mallin suunnittelijat toteavat, että interaktiivisessa järjestelmässä käsitellään ainakin kolmen tyyppistä dataa: sovelluskohtaista dataa (esim. tietokannan nimi- ja tyyppiarvoja), syöte- ja tulostelaitteiden dataa (esim. pikseleitä ruudulla) ja erilaisia datan välimuotoja (esim. data kahdessa sarakkeessa yhden elementin valinnalla).

Data liikkuu kahdensuuntaisesti sovelluskohtaisten komponenttien ja syöte- ja tulostelaitteiden välillä. Sovelluskohtaiseen dataan kohdistetaan muutosoperaatioita, kunnes jossain vaiheessa saavutetaan tulostelaitteiden taso. Vastavuoroisesti syötelaitteilta tulevaa dataa muokataan, kunnes saadaan aikaiseksi sovelluskohtaista dataa. Käsiteltävää dataa järjestellään monella eri tasolla. Esimerkiksi käyttäjän yksittäiset toimet voidaan ymmärtää osana isompaa tehtävää ja näitä toimia täytyy järjestää tehtävän mukaisella tavalla. Samoin syötelaitteilta tulevaa dataa täytyy käsitellä ja yhdistää. Esimerkiksi hiiren raahaaminen samaan aikaan, kun näppäin on painettu pohjaan, tulee osata yhdistää samaan tapahtumaan.

Näistä puitteista lähtien mallin suunnittelijat listaavat minimijoukon pakollisia operaatioita, jotka jokaisen interaktiivisen järjestelmän pitää toteuttaa. Operaatioiden tärkeys vaihtelee sovelluskohtaisesti.

- Hallinnoi, manipuloi ja nouda sovelluskohtainen data ja suorita muut sovelluskohtaiset toiminnot.

- Uudelleenjärjestä sovelluskohtainen data käyttölittymään soveltuvaan muotoon.

- Tarjoa tehtäväkohtainen järjestäminen. Yksi tehtävä voi koostua joukosta yksittäisiä toimia (esimerkiksi kappaleen raahaaminen paikasta toiseen) tai tehtävään voi kuulua multimodaalista vuorovaikutusta kuten puheen, katseen ja eleen yhteistulkintaa.

- Tarjoa monen näkymän välinen yhteneväisyys.

- Tee päätöksiä syöte- ja tulostemedioiden suhteen.

- Valitse interaktioelementit.

- Tarjoa fyysinen käyttöliittymä käyttäjälle.

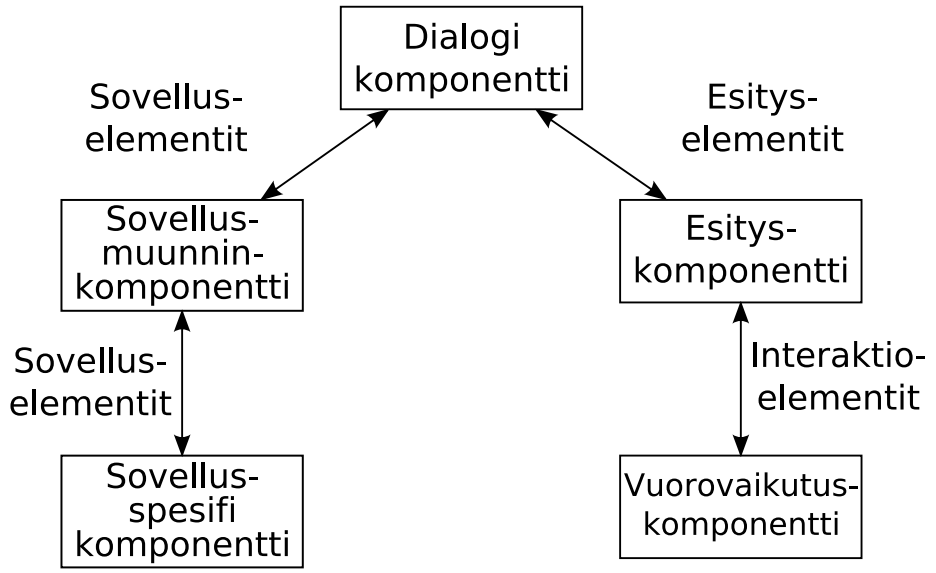
- Muodosta muunto sovelluskohtaisesta formalismista käyttäliittymän formalismiin.

Arch-malli lähtee liikkeelle siitä ajatuksesta, että käyttöliittymäsuunnittelijat usein joutuvat työskentelemään tilanteessa, missä varsinainen sovellus ja käytettävä käyttöliittymäkirjasto asettavat selkeät rajoitteet käyttöliittymien suunnittelulle. Tästä seuraa, että kaarimallin (arch) peruspilarit ovat sovellusspesifi komponentti eli sovelluksen ydin ja vuorovaikutuskomponentti eli syöte- ja tulostelaitteet (kuva 3.2). Sovellusspesifi komponentti hallinnoi, manipuloi ja noutaa sovelluskohtaista dataa ja suorittaa muut sovelluskohtaiset toiminnot. Vuorovaikutuskomponentti mahdollistaa fyysisen vuorovaikutuksen käyttäjän kanssa.

Kaaressa on kolme muuta osaa. Dialogikomponentti vastaa tehtäväkohtaisesta järjestämisestä. Se käsittelee käyttäjän toimien järjestämisen sekä sen osan sovelluskohtaisesta järjestämisestä, mikä riippuu käyttäjän toimista. Dialogikomponentti vastaa myös monen näkymän välisestä yhteneväisyydestä ja datan muuntamisen sovelluskohtaisesta formalismista käyttöliittymäkohtaiseen formalismiin ja toiseen suuntaan.

Esityskomponentti on välittäjä tai puskuri dialogikomponentin ja vuorovaikutuskomponentin välimaastossa. Se muodostaa ja välittää yleisluontoisia elementtejä dialogikomponentille. Tällainen elementti on esimerkiksi valintaelementti, joka kuvaa käyttöliittymässä valikon kautta tai valintapainikkeiden avulla tehtyjä valintoja. Esityskomponentti myös päättää palautteiden esitystavasta.

Sovellusmuunnin-komponentti toimii vuorostaan välittäjänä sovellusspesifin komponentin ja dialogikomponentin välissä. Tässä komponentissa toteutetaan ne sovelluskohtaiset tehtävät, joita tarvitaan käyttäjän toimien tukemiseen, mutta



Kuva 3.2 Arch-malli

joita ei ole sovelluspesifissä komponentissa. Komponentti välittää myös sovelluksen ytimestä lähtevät dialogitehtävät dialogikomponentille, uudelleenjärjestää sovelluskohtaista dataa ja havaitsee ja raportoi semanttiset virheet.

Vaikka sovelluselementtejä käytetään sekä sovellusspesifissä komponentissa että sovellusmuunnin-komponentissa, on niiden käyttötapa erilainen. Sovellusspesifissä komponentissa elementit sisältävät sovelluskohtaista dataa ja operaatioita, jotka eivät suoraan liity käyttöliittymään. Sovellusmuunnin-komponentissa taas elementti sisältää operaatioita, jotka on tarkoitettu sovelluskohtaisen datan muuntamiseksi käyttöliittymälle. Sovellusmuunnin-komponentissa voidaan esimerkiksi yhdistää listaksi sovellusspesifistä komponentista saatuja tietoalkioita.

Interaktioelementit kuvaavat syötteitä (näppäimistö, hiiri, puhe) ja tulosteita (pikseleitä ruudulla, tuntopalautetta, ääntä). Esityskomponentilta saatu lista voidaan esimerkiksi näyttää ruudulla pudotusvalikkona tai listarakenteena.

Esityselementit ovat virtuaalisia interaktioelementtejä, jotka kuvaavat käyttäjälle näytettävää dataa ja käyttäjän laukaisemia tapahtumia. Malli ei ota kantaa esityksen mediamuotoihin eikä tapahtumien luontiin. Esimerkiksi sovellusmuunnin-komponentilta saatu lista voi olla esityselementtinä 'yksisarakkeinen, otsikoitu ja lajiteltu lista, josta voi valita yhden rivin kerrallaan'.

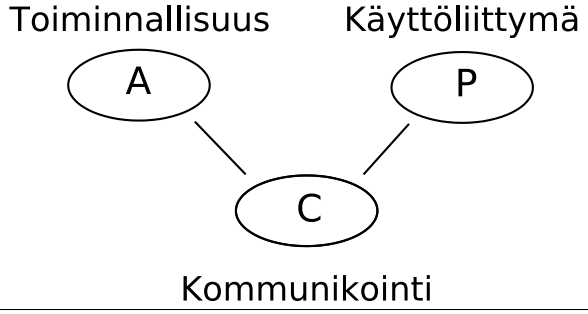
Siinä missä edellä kuvattu Arch-malli on suunniteltu minimoimaan tulevien muutosten vaikutus eri komponentteihin, yleistää Slinky-metamalli Arch-mallia siten, että eri komponenttien tärkeys ja niiden sisältämä toiminnallisuus voi vaih-

della sovelluksesta toiseen. Joissakin sovelluksissa on paljon toiminnallisuutta vuorovaikutuskomponentissa, kun taas sovellusspesifi komponentti voi olla hyvinkin pieni. Toisissa sovelluksissa voi sovellusspesifi komponentti olla sovelluksen isoin komponentti ja dialogikomponentti lähes olematon. Painopiste kaaressa voi siis liikkua eri komponenttien välillä.

3.1.2 PAC

PAC-suunnittelumallin [Coutaz, 1987, Buschmann *et al.*, 1996] nimi tulee sanoista Presentation, Abstraction, Control. Malli pohjaa agentteihin, jotka tässä tarkoituksessa ovat yksittäisiä ja selkeän toiminnallisuuden omaavia komponentteja, eivätkä välttämättä itsenäisiä ja omassa säikeessä ajettavia. Nykytermistöllä PAC-agentit voidaan ehkä mieltää olioiksi ennemmin kuin agenteiksi sanan varsinaisessa merkityksessä. Suunnittelumalli eriyttää tehokkaasti toiminnallisuuden pienempiin erillisiin yksiköihin ja mahdollistaa järjestelmän joustavan muuttamisen ja uuden toiminnallisuuden lisäämisen.

Malli erottaa käyttöliittymäosan, toiminnallisen ytimen ja kommunikaatio-osan toisistaan. Käyttöliittymäosa (Presentation) toteuttaa agentin käyttäjälle näkyvän osan. Toiminnallinen ydin (Abstraction) vastaa agentin sisäisestä tietomallista ja tähän kohdistuvista operaatioista. Kommunikaatio-osa (Control) yhdistää käyttöliittymäosan ja toiminnallisen ytimen ja mahdollistaa kommunikaation toisten agenttien kanssa (kuva 3.3).



Kuva 3.3 PAC-agentti

Yksittäinen PAC-agentti on vain pieni osa varsinaista PAC-mallia. PAC-mallin mukaan agentit tulee järjestää hierarkkisiin suhteisiin toisiinsa nähden. Hierarkiassa on yksi ylimmän tason PAC-agentti, sekä useita alimman tason agentteja ja välitason agentteja. Ylimmän tason agentti toteuttaa toiminnallisen ytimen tai on yhteydessä siihen. Ylimmän tason agentti myös sisältää ne käyttöliittymän

osat, kuten valikon tai valintaikkunat, joita olisi vaikea jäsentää jonkin alemman tason agentin vastuulle.

Ylimmän tason agentin kommunikaatio-osa mahdollistaa alemman tason agenttien pääsyn jaettuun tietoon. Alemmalta tasolta tulevat palvelupyynnöt ohjataan joko agentin toiminnalliselle ytimelle tai käyttöliittymäosalle. Kommunikaatio-osa vastaa myös käyttäjän toimien oikeellisuuden tarkistamisesta ja se voi myös sisältää tehtyjen toimien historian ja peruuta-kumoa-toiminnallisuuden. Lisäksi kommunikaatio-osa koordinoi alemman tason agenttien toimintoja ja pitää agenttien välisistä yhteyksistä kirjaa.

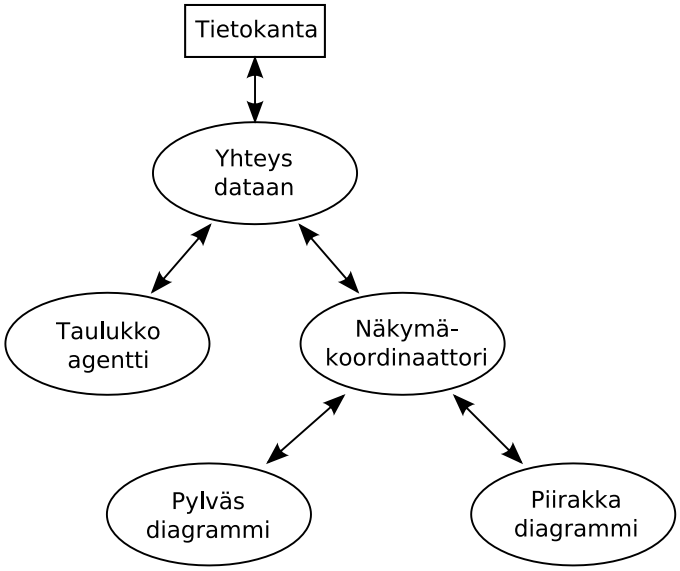
Alimman tason agentit taas edustavat muista riippumattomia komponentteja, joita käyttäjä voi useimmissa tapauksissa manipuloida. Esimerkiksi erilaiset kaaviot, taulukointinäkymät ym. vuorovaikutteiset elementit ovat alimman tason agentteja. Agentti tarjoaa ja toteuttaa tavat, millä käyttäjä voi manipuloida agenttia. Käyttäjälle näkyvän toiminnallisuuden toteutus tapahtuu luonnollisesti agentin käyttöliittymäosassa. Alimman tason agentteina voi olla myös käyttöjärjestelmän palveluita käyttäviä agentteja sekä agentteja, jotka mahdollistavat yhteydet toisiin PAC-hierarkioihin tai ulkopuolisiin palveluihin.

Välitason agentit vuorostaan toimivat alemman ja ylemmän tason agenttien välisen liikenteen välittäjinä. Ne myös toimivat alemman tason agenttien välisten suhteiden ja riippuvuuksien kuvaajina ja vastaavat niiden välisestä yhdenmukaisuudesta.

Otetaan esimerkiksi sovellus, jossa käsitellään tietokannassa sijaitsevaa demografista tietoa. Käyttäjällä on useita näkymiä tietoon. Käyttäjä voi esimerkiksi tarkastella tietoa pylväs- ja piirakkadiagrammien avulla ja muokata tietokannan tietoja taulukkomuodossa. Alimman tason agentteja on kolme. Taulukkoagentti, pylväs- ja piirakkadiagrammi. Välitason agentteja on vain yksi: näkymäkoordinaattori, jonka alla diagrammit ovat (kuva 3.4).

Esimerkin ylimmän tason agentin toiminnallinen ydin vastaa yhteydestä tietokantaan ja tarjoaa operaatiot tietokannan käsittelemiseen ja tiedon eheyden ylläpitämiseen. Kommunikaatio-osa järjestää alemman tason agenttien välistä liikennöintiä ja yhteistyötä. Jos käyttäjä muuttaa tietoja taulukkoagentissa, tieto muutoksesta menee ylimmän tason agentille, joka päivittää tietokannan tiedot ja lähettää näkymäkoordinaattorille viestin muutoksista. Viesti etenee luonnollisesti diagrammeille saakka, jotka päivittävät näkymänsä asianmukaisella tavalla. Tämän esimerkin ylimmän tason agentilla ei ole käyttöliittymäosiota.

Alimman tason agenttien toiminnallinen ydin tallentaa demografista dataa käyttötarkoitukseen soveltuvaan muotoon ja pitää yllä agenttikohtaista tietoa da-



Kuva 3.4 PAC-agenttien hierarkia

tasta (esim. esitysjärjestys). Käyttöliittymäosion vastuuna on luoda tallennetusta datasta havaittava representaatio (ääni, kaavio, kuva) ja toteuttaa mahdolliset datan manipulaatiomenetelmät. Kommunikaatio-osa mahdollistaa agentin yhteyden ylemmän tason agenttiin, diagrammin tapauksessa näkymäkoordinaattoriin ja taulukkoagentin tapauksessa ylimmän tason agenttiin. Myös agentin sisäiseen malliin kohdistuvat muutokset kulkevat kommunikaatio-osan kautta.

Esimerkkimme välitason agentin, näkymäkoordinaattorin, käyttöliittymäosa voisi tarjota käyttäjälle kontrollipaneelin, mistä voi avata tai sulkea erilaisia demografista tietoa näyttäviä diagrammeja. Toiminnallinen ydin vuorostaan pitäisi yllä tietoa aktiivisista näkymistä. Kommunikaatio-osa vastaa alemman tason agenttien koordinoinnista välittäen esimerkiksi tietoa jaetun tietomallin muutoksista alemman tason agenteille. Välitason agentti välittää myös alemman tason agenttien jaettuun malliin tekemät muutokset ylemmälle tasolle. Agentin kommunikaatio-osa vastaa niiden alemman tason agenttien luonnista ja poistamisesta, jotka ovat seurausta käyttöliittymäosan kontrollipaneelin käytöstä.

PAC-arkkitehtuuri on helposti mukautettavissa monisäikeiseksi ja se on hajautettavissa lähiverkon kautta usealle koneelle. Toisaalta malli saattaa lisätä järjestelmän monimutkaisuutta ja tehottomuutta, jos agenttien määrää ei osata kontrolloida. Agentin kolmiosainen rakenne lisää ohjelmiston monimutkaisuutta ja agentin kommunikaatio-osan toteutus saattaa muodostua monimutkaiseksi ja työlääksi.

3.1.3 Arch, PAC ja PAC-Amodeus

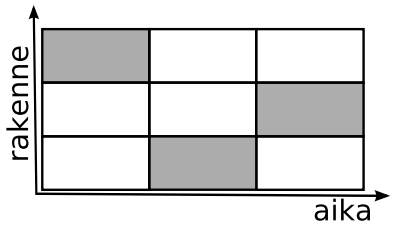
Arch-malli ei ota kantaa siihen, miten dialogikomponentissa tapahtuva tehtävien järjestäminen voitaisiin tehdä. Coutaz ja Nigay esittelivätkin PAC-Amodeus mallin [Nigay & Coutaz, 1993, Nigay & Coutaz, 1995], joka yhdistää PAC-mallin Arch-mallin dialogikomponenttiin. PAC-Amodeus-mallin lähtökohtana on multimodaalisen vuorovaikutuksen mahdollistaminen sovelluksessa. Kuten aiemmin mainittiin, multimodaalista vuorovaikutusta on monenlaista, mutta eri modalitettien yhdistäminen eli fuusio antaa käyttäjälle enemmän ilmaisuvoimaa ja lisää käyttäjän vuorovaikutusmahdollisuuksia järjestelmän kanssa. PAC-mallin upottaminen dialogikomponenttiin mahdollistaa modaliteettien yhdistämisen luontevalla tavalla.

Nigay ja Coutaz [1993] lajittelevat fuusion kolmeen eri ryhmään: leksikaalinen, semanttinen ja syntaktinen fuusio. Leksikaalinen fuusio tapahtuu laitetason rajapinnassa eli vuorovaikutuskomponentissa. Esimerkiksi Shift-näppäimen alas painaminen samanaikaisesti, kun hiirellä valitaan listasta useampia elementtejä, voidaan laskea leksikaaliseksi fuusioksi ja se tulee käsitellä vuorovaikutuskomponentissa.

Syntaktinen ja semanttinen fuusio taas tapahtuu dialogikomponentissa. Syntaktisessa fuusiossa koostetaan syötteitä kokonaisen komennon aikaansaamiseksi. Esimerkiksi käyttäjän osoittaessa kohtaa dokumentissa ja sanoessa 'laita kommentti' muodostaa hän yhden komennon, joka koostuu kahdesta erillisestä toiminnosta ja modaliteetista. Syntaktinen fuusio kokoaa nämä kaksi toimintoa yhdeksi järjestelmän ymmärtämäksi komennoksi.

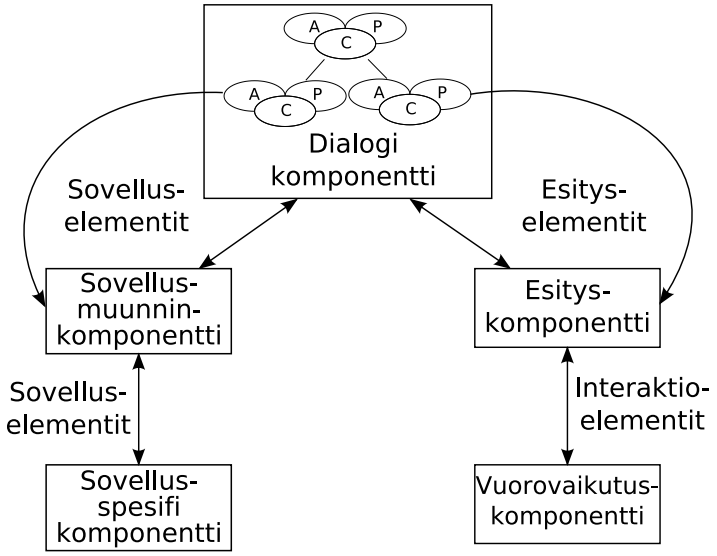
Semanttinen fuusio yhdistää komentoja uusiksi tapahtumiksi. Käyttäjä voi esimerkiksi samanaikaisesti piirtää viivaa ja määritellä viivan väriä, jolloin semanttisen fuusion avulla tuloksena on monivärinen viiva.

Komentojen tai toimintojen yhdistäminen vaatii yhteisen ja yleistetyn tietomallin. PAC-Amodeuksessa yhteisen tietomallin virkaa hoitaa 'sulatusuuni' (melting pot). Sulatusuuni on kaksiulotteinen rakenne, jonka x-akselilla kuvataan aikaa ja y-akselilla kuvataan käyttäjän laukaisemat tapahtumat. Sulatusuunin y-akselin tapahtumat ovat dialogikomponentin ymmärtämässä yleistetyssä rakenteellisessa muodossa [Nigay & Coutaz, 1991]. Tapahtuman sijainti y-akselilla määrittää sen suhteen muihin sulatusuunin tapahtumiin. Esimerkiksi valitulla kohteella ja sen määränpäällä on omat sijaintinsa sulatusuunin y-akselilla. Muunnos käyttäjän laukaisemista tapahtumista yleiseen muotoon tapahtuu sekä vuorovaikutus- että esityskomponentissa (kuva 3.5).



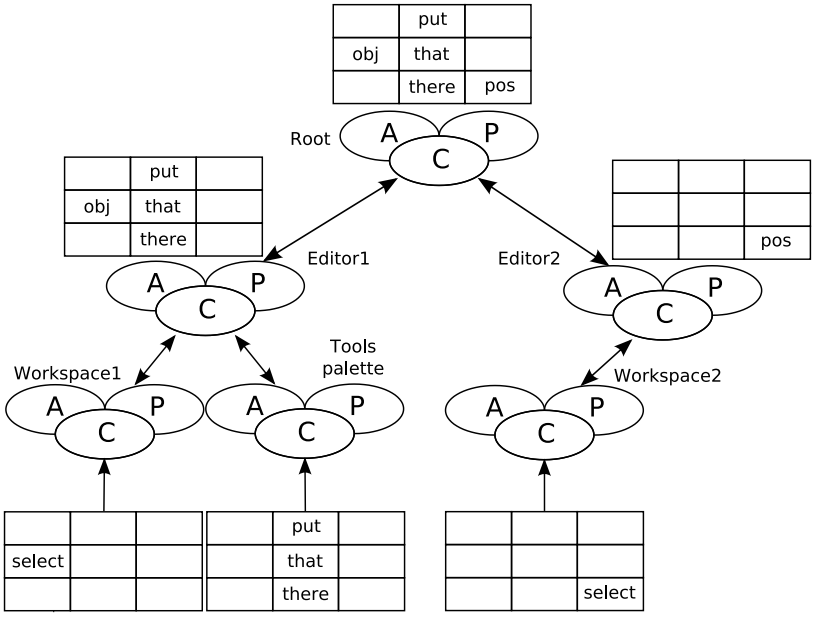
Kuva 3.5 PAC-Amodeuksen sulatusuuni

PAC-Amodeus-mallissa dialogikomponentti sisältää useita PAC-agenttien hierarkioita (kuva 3.6). PAC-agentti on yhteydessä sovellusmuunninkomponenttiin toiminnallisen ytimensä kautta (Abstraction) ja esityskomponenttiin vuorovaikutusosansa kautta (Presentation). Yhdellä agentilla voi olla useita yhteyksiä edellä mainittuihin komponentteihin. Dialogikomponentissa tapahtuu ylimmän tason fuusio sulatusuunielementtien avulla. Fuusioon vaikuttavat sekä sulatusuunien muodostama aikaikkuna että sulatusuunien tapahtumien välinen suhde.



Kuva 3.6 PAC-Amodeus-malli

Esimerkkinä mainitaan grafiikkaeditori, missä on tuki puheentunnistukselle ja hiiren tapahtumille [Nigay & Coutaz, 1991]. Sovelluksessa on useita erillisiä piirtoalueita eli työpöytiä. Työpöytien lisäksi käyttäjällä on käytössään työkalupaletti. Kullakin työpöydällä, niin kuin työkalupaletillakin, on oma alimman tason PAC-agenttinsa dialogikomponentissa. Agenttihierarkiassa on myös välitason agentteja, Editoreita, jotka toimivat alemman tason agenttien koordinoijina ja niiltä tulevien sulatusuunielementtien yhdistäjinä (kuva 3.7).



Kuva 3.7 Esimerkin dialogikomponentin PAC-agenttien hierarkia

Käyttäjä siirtää valitsemansa kohteen työpöydältä 1 työpöydälle 2 sanomalla "laita tuo tuonne" ja samanaikaisesti valitsemalla kohteen ja sen määränpään. Agentti Työpöytä1 saa sulatusuunielementin esityskomponentilta ja muuntaa siinä olevan valintatoiminnon viitteeksi valittuun elementtiin. Samanaikaisesti agentti Työpöytä2 saa myös esityskomponentilta sulatusuunielementin ja muuntaa siinä olevan valinta-komennon vastaamaan sijaintia työpöydällä. Työkalupaletti saa käyttäjän antaman käskyn sulatusuunielementtinä esityskomponentilta ja lähettää sen eteenpäin yhdistävälle agentille Editori1. Agentti Editori1 yhdistää agentilta Työpöytä1 saadun elementin työkalupaletin elementin kanssa ja lähettää yhdisteen ylemmäksi agenttihierarkiassa. Samanaikaisesti agentti Editori2 saa agentti Työpöytä2 sulatusuunielementin ja koska muuta syötettä ei ole tulossa, ohjaa se sen hierarkian ylemmälle tasolle. Juuritasolla kaksi saatua elementtiä yritetään yhdistää. Jos yhdistetty elementti noudattaa fuusion sääntöjä, muodostuu siitä kokonainen käsky, joka lähetetään toiminnalliselle ytimelle.

4 Palautteiden synkronointi

Hajautetuissa ympäristöissä eri palautetta antavat laitteet voivat olla jopa maantieteellisesti toisistaan erillään. Lähiverkon yli kommunikoidessa ohjelmien toimintanopeus vaihtelee jatkuvasti. Viestien lähettäminen, välitys ja purkaminen ovat sidoksissa mm. viestin sisältöön, lähettäjän ja vastaanottajan sijaintiin ja verkkoliikenteen senhetkiseen määrään. Enää ei riitä, että kuva ja ääni suoritetaan peräkkäin ja toivotaan että samanaikaisuuden illuusio säilyy.

Tarvitaan siis jonkinlaista mekanismia mediaelementtien synkronointiin. Synkronointi ei tarkoita vain samanaikaisesti toistettavien elementtien kohdistusta, vaan erilaisia synkronointitapauksia on useita. Tässä kappaleessa esitän synkronointiin liittyviä ongelmia ja muutamia niiden ratkaisumalleja. Synkronointia käsitellessä oletetaan, että kyseessä ei ole reaaliaikainen käyttöjärjestelmä, joka pystyy paremmin mm. antamaan prosessille aikaa sen tarpeiden mukaan. Oletuksena on siis, että prosesseja ajetaan tavallisissa tietokoneissa ja käyttöjärjestelmissä, missä ei ole laatutakeita esimerkiksi prosessin suorituksen suhteen.

Medialähteiden synkronointia on tutkittu paljon jo 80-luvun lopulta saakka. Onhan kyseessä ongelma, joka on yhä akuutimpi Internetin kasvun myötä. Synkronointi jaetaan yleisesti kahteen osaan: mediasäikeiden sisäiseen (inter) ja mediasäikeiden väliseen (intra) synkronointiin. Mediasäikeen sisäisessä synkronoinnissa samassa tietovirrassa tulee useampaa mediaa ja ko. lähteen toistamiseksi tarvitsee suorittaa näiden säikeiden synkronointi. Säikeiden välinen synkronointi taas tarkoittaa sitä, että mediavirrat tulevat mahdollisesti eri paikoista erillisinä virtoina ja synkronointi yhdistää virrat niin, että ne muodostavat yhdenmukaisen kokonaisuuden. Yleisimmät synkronointitapahtumat koskevat yleensä kuvaa ja ääntä, mutta viime aikoina on tutkittu myös uusien modaliteettien, kuten tuntopalautteen, synkronointia [Wongwirat & Ohara, 2006].

Tässä luvussa perehdytään vain säikeiden väliseen synkronointiin, sillä on todennäköistä, että moniaistisissa järjestelmissä eri palautteet tulevat erillisinä palautevirtoina. Myöhemmin tässä tutkielmassa esitän yhden mallin synkronoinnin lisäämiseksi seuraavassa luvussa esitettävään agenttialustaan. En myöskään erityisemmin ota kantaa tässä luvussa erilaisiin epäsynkronian korjaustapoihin, eikä niitä myöskään käsitellä tarkemmin esimerkkien yhteydessä.

Synkronointitavat voidaan jakaa kolmeen kategoriaan [Manvi & Venkataram, 2005]. *Pistesynkronoinnissa* mediasäikeiden käynnistys tai lopetus synkronoidaan, mutta muuta synkronointia ei tapahdu. *Tosiaikaisessa synkronoinnissa* mediayksikkö synkronoidaan joko ajan tai toisen mediayksikön kanssa. Esimerkiksi videon

27. ruudun tulisi ilmestyä videon toiston 27. ruudun kohdalla. Tämä synkronointimalli korjaa epäsynkronian esimerkiksi pudottamalla toistosta mediaelementtejä, kunnes synkronia on jälleen saavutettu. Lopuksi *mukautuva synkronointi* yrittää selviytyä mahdollisimman vähällä mediaelementtien hylkäämisellä laskemalla säännöllisin väliajoin synkronointiin liittyviä laskennallisia arvoja, kuten verkon arvioitua viivettä.

Synkronointiin liittyy joukko yleisiä ongelmia, jotka on sitä suunniteltaessa ratkaistava:

- Verkosta johtuva tiedonsiirtoviiveen vaihtelu. Verkossa olevaa kuormitusta on mahdoton ennustaa. Tämän lisäksi tiedonsiirron nopeus paikasta toiseen riippuu maantieteellisestä etäisyydestä ja yhteyden välillä olevista tietoliikennettä ohjaavista laitteista.

- Verkossa olevilla tietokoneilla on eri kellotaajuus. Tämä on ongelma vain, jos mediaa suoratoistetaan (Streaming). Suoratoistossa jatkuvaa mediaa, kuten videota tai ääntä, lähetetään tasaisena virtana verkon yli. Vastaanottava kone puskuroi tulevan datan ja näyttää sen synkronoituna käyttäjälle.

- Äkilliset prosessorikuormitukset. Ei-reaaliaikaiset järjestelmät eivät voi koskaan luvata tiettyä ajankohtaa prosessin suoritukselle. Tästä johtuu, että esimerkiksi äkilliset prosessorikuormitukset voivat aiheuttaa epäsynkroniaa. Tämäkin ongelma koskee lähinnä suoratoistettavaa dataa.

- Verkko ja prosessorikuormat. Joillakin synkronointiin osallistuvista koneista voi olla jatkuva raskas prosessorikuormitus, joka haittaa synkronointiprosessin suoritusta. Tämän lisäksi verkkoliikenne voi olla ruuhkaista ja aiheuttaa ennalta-arvaamattomia viiveitä.

- Globaalin kellon puuttuminen. Synkronointi olisi helppo suorittaa, mikäli eri koneiden synkronointiprosessit jakaisivat saman kellon, eli synkronointi voitaisiin suorittaa suoraan kellonaikojen avulla. Globaalia jaettua kelloa ei tosin ole yleisesti olemassa (paitsi ntp-palvelimet). Lisäksi koneiden kellot yleisesti jätättävät tai edistävät jatkuvasti, joten niitä on tarve säätää tietyin väliajoin.

4.1 Eri medioiden epäsynkronian toleransseista

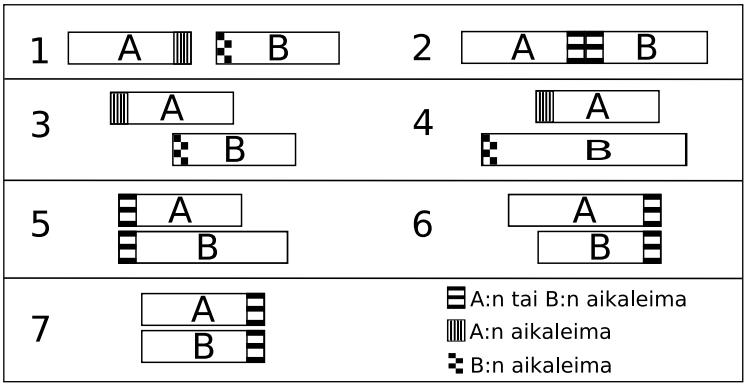
Ralf Steinmetz tutki eri medioiden synkronointia ja pyrki selvittämään, kuinka paljon mediat saavat olla epäsynkroniassa keskenään, ennen kuin katsoja havaitsee sen [Steinmetz, 1996]. Steinmetz suoritti tutkimuksessaan kaksi eri testiä. Ensinnäkin hän tarkkaili huulisynkroniaa kolmella eri kuvakoolla. Toisekseen hän mittaa epäsynkronian havaitsemista esitystilanteessa, missä luennoitsija korostaa puhettaan osoittamalla kursorin puheen kohteeseen. Artikkelin lopuksi Steinmetz esittää koosteen eri medioista ja niiden välisistä epäsynkroniatoleransseista. Taulukossa 4.1 on esitetty Steinmetz:n löytämät toleranssirajat synkronoitaessa videota tai ääntä jonkin muun median kanssa.

Media		**Tyyppi**	**Toleranssi**
Video	Animaatio	Korrelointi	± 120 ms
	Ääni	Huulisynkronia	± 80 ms
	Kuva	Päällekkäin	± 240 ms
	Kuva	Ei päällekkäin	± 500 ms
	Teksti	Päällekkäin	± 240 ms
	Teksti	Ei päällekkäin	± 500 ms
Ääni	Animaatio	Tapahtuman korrelointi	± 80 ms
	Ääni	Tiukasti yhdistetty (stereo)	± 11 ms
	Ääni	Löyhästi yhdistetty (dialogi)	± 120 ms
	Ääni	Löyhästi yhdistetty (taustamusiikki)	± 500 ms
	Kuva	Tiukasti yhdistetty (musiikki ja nuotit)	± 5 ms
	Kuva	Löyhästi yhdistetty (kalvoesitys)	± 500 ms
	Teksti	Tekstin selitykset	± 240 ms
	Kohdistin	Ääni yhteydessä näytettyyn kohteeseen	± 500 ms

Taulukko 4.1: Epäsynkronian toleranssiarvoja

4.2 Synkronointitavat

Hajautettujen multimediasovellusten synkronointia käsittelevässä artikkelissaan Nipun Agarwal ja Sang H. Son [Agarwal & Son, 1996] listaavat C. Hamblinin tutkimukseen perustuvat seitsemän erilaista synkronointitapahtumaa (kuva 4.1). Alkuperäinen tutkimus listaa 13 tapahtumaa, mutta kuusi niistä on alla esitetyn listan tapausten peilikuvia, minkä vuoksi ne on jätetty kuvasta pois.



Kuva 4.1 Seitsemän synkronointitapahtumaa

1. A ennen B:tä. Tapahtuma A päättyy ennen kuin tapahtuma B alkaa. Tapahtumien välissä on aikaa. Synkronointiin tarvitaan A:n lopun aikaleima ja B:n alun aikaleima.

2. A kohtaa B:n. Tapahtuma B alkaa välittömästi tapahtuman A jälkeen. Synkronointiin tarvitaan joko A:n lopun aikaleima ja/tai B:n alun aikaleima.

3. A on B:n kanssa päällekkäin. Tapahtuma B alkaa tapahtuman A ollessa käynnissä, mutta ei samanaikaisesti. Tapahtuma B kestää pidempään kuin tapahtuma A. Synkronointiin tarvitaan sekä A:n että B:n alun aikaleimat.

4. A B:n aikana. Tapahtuma A alkaa ja päättyy tapahtuma B:n aikana. A:n tulee alkaa B:n alkamisen jälkeen ja päättyä ennen B:n päättymistä. Synkronointiin tarvitaan sekä A:n että B:n alun aikaleima.

5. A aloittaa B:n. A ja B alkavat samanaikaisesti. Synkronointiin tarvitaan joko A:n tai B:n alun aikaleima, tai molemmat.

6. A lopettaa B:n. A ja B päättyvät samanaikaisesti. Synkronointiin tarvitaan joko A:n tai B:n lopun aikaleima, tai molemmat.

7. A samanaikaisesti B:n kanssa. A ja B alkavat ja päättyvät samanaikaisesti. Synkronointiin tarvitaan joko A:n tai B:n lopun aikaleima, tai molemmat.

4.3 Synkronointiesimerkkejä

Suurin osa synkronointiratkaisuista käyttää jonkinlaista jaettua keinoa ajankohtien määrittelyyn, esimerkiksi normalisoitua kelloa tai synkronointipaketteja. Ennen synkronisaatiota verkon toiminnallisuus testataan jollain tavalla ja yleensä selvitetään ainakin, kuinka kauan paketin kestää kulkea vastaanottajalle ja takaisin. Seuraavaksi esittelen lyhyesti muutamia tyyppillisimpiä synkronointiratkaisuja. Kaikki esimerkit koskevat suoratoistettavan median synkronointia.

4.3.1 Multimediauutiset

Multimediauutisissa [Lamont *et al.*, 1996] käyttäjä voi hakea palvelimelta uutisia, jotka voivat koostua kuvasta, grafiikasta, äänestä, videosta, animaatiosta ja tekstistä. Eri medioille on oma palvelimensa. Mediapalvelinten lisäksi järjestelmässä on tietokantapalvelin, joka sisältää uutisten ja esityspakettien tiedot. Uutisen esityspaketti on kooste uutiseen kuuluvista medioista ja niiden välisistä ajallisista suhteista.

Mediauutisten arkkitehtuuri on järjestetty siten, että suurin osa synkronoinnista tapahtuu vastaanottajan päässä olevassa skeduloijassa. Aluksi skeduloija saa tietokantapalvelimelta esityspaketin, jonka perusteella skeduloija luo jokaiselle tarvittavalle mediaelementille vastaanottajaolion. Vastaanottajaolion tehtävänä on vastaanottaa dataa ja antaa se eteenpäin skeduloijalle.

Skeduloija tekee esityspaketin perusteella myös lähetysaikataulun, joka sisältää aikataulun jokaiselle tarvittavalle mediaelementille. Aikataulun tarkoituksena on säätää asiakkaalle lähetettävien mediaelementtien lähetysajat siten että ne tulevat ajallaan, mutta eivät liian aikaisin, jotta puskuroimiseen ei tarvita liikaa tilaa. Lähetysaikataulun laskemiseen tarvitaan tieto verkon viiveestä, sen varianssista ja muista lähetykseen liittyvistä rajoituksista. Palvelun laatuun ja muihin lähetykseen vaikuttavien tekijöiden laskeminen tapahtuu palvelun laadusta vastaavan komponentin kautta, joka aloittaa lähetyksen laatuneuvottelut mediapalvelinten, vastaanottajan sovelluksen ja lähetyskanavan välillä.

Aluksi palvelunlaatukomponentti tutkii esityspaketin ja koostaa siitä palveluehdotuksen huomioiden käyttäjän asetukset, vastaanottajan koneen ja käyttöjärjestelmän aiheuttamat rajoitukset, lähetyskanavan tiedot ja mediapalvelinten rajoitukset. Lähetyskanavan tiedot sisältävät mm. yksittäisen datapaketin maksimikoon, datan lähetysnopeuden, keskimääräisen päästä-päähän verkkoviiveen ja viiveen varianssin. Palveluehdotus lähetetään mediapalvelimille, jotka avaavat yhteyden vastaanottajaan yrittäen saada aikaiseksi yhteyden, joka noudattaisi

lähetyskanavan laatutietoja. Jos mediapalvelin ei saa avattua sopivaa yhteyttä, hylkää se palveluehdotuksen. Palvelunlaatukomponentti voi tehdä uuden ehdotuksen ja suorittaa neuvottelun uudelleen.

Mediaelementin datapaketille lasketaan kokonaisviive, mikä vastaa sitä aikaa mikä kuluu siitä, kun paketti lähetetään palvelimelta siihen, kun paketti esitetään käyttäjälle. Kokonaisviive koostuu kolmesta eri viiveestä: aika mikä kuluu paketin lähettämisestä sen saapumiseen vastaanottajalle (verkkoviive), aika mikä kuluu paketin käsittelyyn ja purkamiseen (mahdollinen videon purku yms.) ja aika mikä kuluu paketin puskuroimiseen. Puskurin koolla on suuri merkitys synkronoinnissa, ja mediauutisissa puskurille määritelläänkin koko sen mukaan, mikä on sallittu prosentuaalinen todennäköisyys datapaketin myöhästymiselle.

Kun palveluehdotus on hyväksytty, luo skeduleri aikagraafin esityksestä. Aikagraafia apuna käyttäen skeduleri koostaa lähetysaikataulun. Lähetysaikataulussa on jokaiselle mediaelementille laskettu lähetysaika, joka pohjaa kyseisen median datapakettien laskettuun kokonaisviiveeseen. Mikäli esitys alkaa useammalla medialla, varautuu skeduleri mahdolliseen alkuviiveeseen laskemalla mukaan synkronointiviiveen, mikä on ensimmäisenä esitettävien medioiden nopeimman paketin viiveen ja hitaimman paketin viiveen välinen aika. Lähetysaikataulu lähetetään jokaiselle esityksessä tarvittavalle mediapalvelimille, jotka lähettävät mediaelementit lasketun aikataulun mukaan.

4.3.2 Aikaleimat

Ernst Biersack ja kumppanit rakentavat tutkimuksessaan synteettisille (talletetuille) videovirroille synkronoinnin käyttäen aikaleimoja [Biersack *et al.*, 1996]. Heidän ratkaisunsa painottuu vastaanottajan päähän, eikä vaadi yhteisen kellon olemassaoloa. Ratkaisu käyttää monia synkronoinnin korjaustapoja, kuten puskurin koon dynaamista muuttamista ja mediayksiköiden pudottamista. Tutkimuksessa rakennetaan kolme mallia, jotka rakentavat edellisen mallin varaan ja täten lisäävät synkronointiprotokollaan asteittain uusia ominaisuuksia.

Video on jaettu palvelinten kesken niin, että yksittäinen videon ruutu on lohkottu niin moneen osaan kuin palvelimia on. Jokainen palvelin saa oman osansa videon ruuduista. Eri mediasäikeiden samaan aikaan toistettavat mediayksiköt yhdistetään synkronointijoukoksi. Synkronointijoukko siis vastaa yhtä videon ruutua, joka on jaettu palvelinten kesken, ja mediayksikkö vastaa yhtä videon ruudun osaa. Synkronoinnin avuksi jokaiseen mediayksikköön liitetään lähettävän tahon aikaleima ja mediayksikön järjestysluku mediavirrassa.

Ensimmäinen malli on hyvin yksinkertainen ja sisältää vain aloitusajan synkronoinnin. Mallissa oletetaan, että verkkoviive on vakio ja viiveiden huojuntaa ei tapahdu verkkoliikenteessä tai synkronointitehtävään liitetyillä koneilla. Synkronointi katsotaan onnistuneeksi, kun vakuututaan siitä, että kaikki synkronointiryhmään kuuluvat mediayksiköt saapuvat vastaanottajalle samaan aikaan. Helpoin tapa synkronoinnin saavuttamiseksi tässä tapauksessa olisi säilyttää mediayksiköitä kunnes viimeinenkin ryhmään kuuluva yksikkö on saapunut. Tämä kuitenkin saattaisi vaatia kohtuuttoman suuria puskureita mediayksiköiden tallettamiseen. Ensimmäisessä mallissa mediayksiköitä ei halutakaan säilöä vastaanottajan päässä liian pitkään, vaan synkronointi halutaan järjestää niin, että mediayksiköt saapuisivat vastaanottajalle parhaimpaan mahdolliseen aikaan. Tämä ongelma ratkaistaan hyvin tyypillisellä tavalla, eli jokaiselle mediavirralle lasketaan verkkoviive ja median lähetysaikaa säädetään viiveen mukaan.

Synkronointiprosessi lähtee käyntiin aloitusprotokollalla, joka koostuu kahdesta vaiheesta. Arviointivaiheessa jokaiselle mediavirralle lasketaan verkkoviive ja synkronointivaiheessa mediavirtojen aloitusajat lasketaan ja tieto välitetään palvelimille. Protokollassa lähetetään kahdenlaisia viestejä palvelimille. Ensinnäkin vastaanottaja voi pyytää palvelimelta mediayksikköä ja toisekseen se voi lähettää palvelimelle mediavirran lähetyksen käynnistysajan. Protokolla suoritetaan kaikkien synkronointiin osallistuvien palvelinten kanssa.

Protokollan aloitusaika, t_{start}, on ensimmäisen vaiheen aloitusaika. Kuten mainittiin, ensimmäinen malli olettaa, että verkkoviive tulee olemaan vakio ja että huojuntaa ei tapahdu. Tällöin voidaan toisen vaiheen aloitusajaksi, t_{ref}, määritellä ensimmäisen vaiheen viimeisen saapuvan mediayksikön saapumisaika. Aloitusprotokollan ensimmäinen vaihe alkaa mediayksikön, m_i, pyynnöllä palvelimilta ajanhetkellä t_{start}. Palvelin saa pyynnön paikallisella ajanhetkellä s_i, joka liitetään vastaanottajalle välittömästi lähetettävään mediayksikköön. Vastaanottaja saa mediayksikön omalla ajanhetkellä a_i. Vastaanottajan saatua viimeisen mediayksikön, ja täten kokonaisen videon ruudun, on vastaanottajan paikallinen aika t_{ref}. Vastaanottaja laskee lopuksi mediayksikön viiveen ja verkon maksimiviiveen. Mediayksikön viive, d_i, on mediayksikön saapumishetken ja protokollan aloitusajan erotus $(a_i - t_{start})$. Maksimiviive, d_{max}, taas on suurin esiintynyt mediayksikön viive. Merkitään suurimman viiveen omaava mediayksikkö muuttujalla m_v.

Protokollan toinen vaihe käynnistyy heti ensimmäisen vaiheen jälkeen ajankohdalla t_{ref}, jolloin vastaanottaja laskee aikaisimman mahdollisen videon toistoajan. Toistoaika on ajankohdan t_{ref} ja maksimiviiveen d_{max} summa. Lisäksi

lasketaan jokaiselle synkronointiryhmän mediayksikön m_i saapumisajan ja suurimman viiveen omaavan mediayksikön m_v saapumisajan erotus $\delta_{vi} = a_v - a_i$. Mediayksikön m_i lähettämisaika, s_{ci}, on palvelimen paikallisen ajan, maksimiviiveen ja saapumisajan erotuksen summa, eli $s_{ci} = s_i + d_{max} + \delta_{vi}$. Vastaanottaja lähettää aloitusajan palvelimelle, joka aloittaa mediaelementtien lähettämisen sen mukaan.

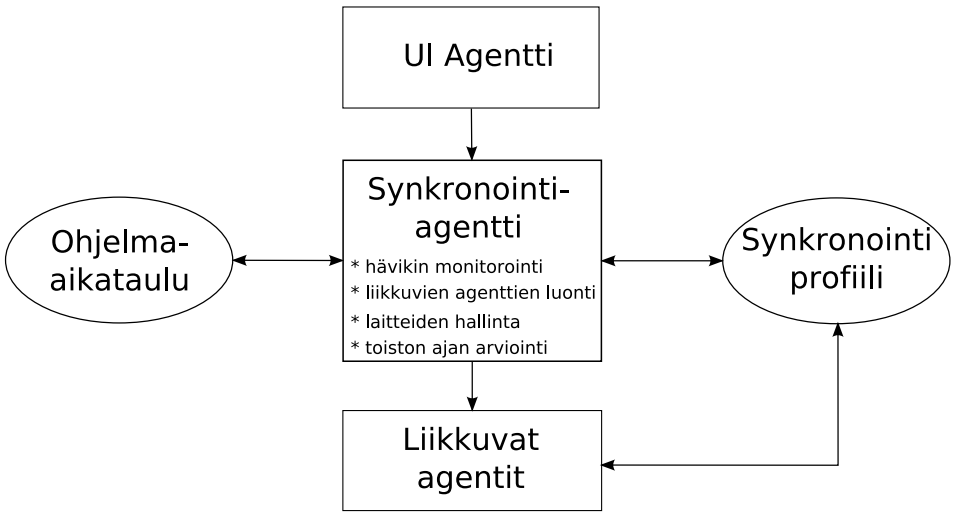
Ensimmäinen malli näyttää videon ruudut välittömästi, eikä täten käytä puskuria mediayksiköiden tallettamiseen. Ensimmäinen malli toimiikin vain jos järjestelmässä ei ole muuttuvia tekijöitä. Toinen malli laajentaa ensimmäistä mallia ottamalla huomioon sekä verkkoviiveen että synkronointitehtävään osallistuvien koneiden ajalliset huojunnat (esimerkiksi prosessorikuormat). Malli ottaa mediayksiköiden puskurit käyttöön ja laskee kullekin mediavirralle k huojunnan Δ_k, joka on virran maksimiviiveen ja minimiviiveen erotus ($\Delta_k = d_{max_k} - d_{min_k}$). Tästä seuraa, että mediavirran toistoa kannattaa viivyttää Δ_k sekuntia, jotta huojunta ei vaikuttaisi median toistoon. Jos virtoja on useampia, etsitään se virta, jonka Δ_k on isoin ja tätä aikaa käytetään kaikkien virtojen toiston viivyttämiseen. Malli olettaa että huojunta on rajoitettua, mikä ei välttämättä pidä paikkaansa ei-reaaliaikaisissa käyttöjärjestelmissä ja lähiverkoissa, joissa ei taata palvelun laatuominaisuuksia.

Tutkimuksen kolmas malli laajentaa toista mallia ja luopuu rajoitetun huojunnan ehdosta. Huojunnan vaihtelun lisäksi malli ottaa huomioon kellojen edistämisen, verkkoviiveen vaihtelun ja palvelinten katoamisen. Biersack ja kumppanit toteavat, että näiden uusien tekijöiden vaikutus näkyy mediavirran puskurissa ja rakentavatkin kolmannen mallin tarkkailemaan puskureiden tilaa. Epäsynkroniasta selvitään suorittamalla siirtyminen mediavirrassa. Mallissa on mekanismi, joka osaa tunnistaa puskurissa tapahtuvan muutoksen aiheuttavan tekijän ja osaa sen mukaan käynnistää mediavirtaan tarvittavan muutoksen. Esimerkiksi, jos puskuria uhkaa ylivuoto verkkoviiveen pienenemisen myötä, aktivoi mekanismi mediayksiköiden väliinjättämisen. Jos taas puskuria uhkaa tyhjeneminen, toistaa mekanismi viimeistä ruutua, kunnes puskuri on saanut täytettä.

4.3.3 Synkronointiagentit

S.S. Manvi ja P. Venkataram kehittivät tutkimuksessaan agenttipohjaisen järjestelmän mediasäikeiden synkronointiin [Manvi & Venkataram, 2005]. Synkronointi tapahtuu vastaanottavassa päässä, missä sijaitsee käyttöliittymäagentti (kuva 4.2). Käyttöliittymäagentti luo synkronointiagentin. Synkronointiagentti koostuu

neljästä osakokonaisuudesta. Ensinnäkin agentti tekee saapuviin tietovirtoihin tiedonhukan kartoituksen ja säätää toimintaansa sen mukaan. Toisekseen synkronointiagentti vastaa saapuvien mediavirtojen toistosta. Lisäksi synkronointiagentti sisältää esitykseen kuuluvien mediaelementtien toistosta luodun ohjelma-aikataulun, jota käytetään eri mediasäikeiden elementtien pyytämiseen palvelimilta oikeina ajankohtina.



Kuva 4.2 Agenttipohjainen synkronointijärjestelmä

Viimeiseksi synkronointiagentti luo jokaiselle esitykseen kuuluvalle mediasäikeelle oman liikkuvan agentin. Näiden liikkuvien agenttien tehtävänä on matkata vastaanottajan ja asianmukaisen palvelimen välissä, laskea matkaan kuluvia viivetekijöitä, mukauttaa ohjelma-aikataulua mahdollisten viiveiden mukaan ja asettaa tiedot median lähettämisajankohdista palvelimelle.

Tutkimuksessa vertaillaan kaikkia kolmea synkronointitapaa. *Pistesynkronoinnissa* synkronointiagentti toimii seuraavasti:

1. Luo mediasäikeille liikkuvat agentit, jotka laskevat koneiden kellojen eroavaisuudet ja verkkoviiveen määrän. Nämä tiedot päivitetään synkronointiprofiiliin.

2. Tämän jälkeen agentit laskevat aikavääristymän, mikä kyseisen mediasäikeen mediaelementin saapumisessa on verrattuna muiden säikeiden saman ajankohdan mediaelementtien saapumisiin verrattuna.

3. Sitten synkronointiagentti laskee mediasäikeille eri viivetekijöistä riippuvan aloitusajan ja lähettää liikkuvat agentit takaisin palvelimille.

4. Lopuksi synkronointiagentti laskee mediasäikeen esittämisen aloitusajan ja päivittää synkronointiprofiilin.

Pistesynkronoinnissa on siis kyse primitiivisimmästä ja elintärkeimmästä synkronoinnista. Jokaisessa synkronointiprotokollassa pitää olla keino saada eri mediasäikeet aloittamaan ja lopettamaan säikeen toisto tiettynä ajankohtana. Tämä on erittäin tärkeätä myös mediaa suoratoistettaessa, sillä jos dataa lähetetään liian aikaisin, kasvaa vastaanottajan päässä tarvittava puskuri kohtuuttoman isoksi. Jos taas datan lähetys aloitetaan liian myöhään, saattaa säikeiden synkronointi epäonnistua täysin tai sitten mediasäikeen puskurin täyttöaste saattaa jäädä liian pieneksi, josta voi seurata esimerkiksi verkkoviiveiden vaihtelun vuoksi puskurin alivuotoa ja mediayksiköiden myöhästymistä.

Mikäli pistesynkronoinnissa mediaelementti saapuu niin myöhässä, että se ei enää mahdu aikavääristymän tolerenssirajojen sisään, käytetään vääristymän korjaamiseksi esimerkiksi videon tapauksessa viimeisen mediaelementin toistamista tai toiston pysäyttämistä, jos mediaelemetti puuttuu. Myöhässä saapuvat mediaelementit jätetään väliin. Tutkimuksessa [Manvi & Venkataram, 2005] esitetään aikavääristymien toleranssirajoiksi 100-200 ms, mutta uskon että ko. toleranssirajat ovat modaliteettiriippuvaisia ja täten toleranssirajat noudattavat paljolti Steinmetz:n tutkimuksen tuloksia [Steinmetz, 1996].

Tosiaikainen synkronointi laajentaa pistesynkronointia siten että lopuksi synkronointiagentti laskee viitteellisen toistoajankohdan jokaiselle mediasäikeen elementille. Toiston alettua synkronointiagentti näyttää kaikki ne mediaelementit, jotka saapuvat arvioidun esitysajan sisällä. Mahdollisia vääristymiä korjataan vastaavasti kuin pistesynkronoinnissa.

Viitteellinen toistoajankohta on mediavirran ensimmäisen elementin lähetysajan, lähettäjän ja vastaanottajan kellojen erotuksen, verkkoviiveen, mediaelementin luomiseen kuluvan ajan ja edellisten mediaelementtien luomiseen kuluneen ajan summa. Mikäli kyseessä oleva mediasäikeen verkkoviive ei ole suurin kaikista säikeistä, lasketaan summaan vielä säikeen ko. ajankohdan aikavääristymä.

Myös *mukautuva synkronointi* laajentaa pistesynkronointia ja laskee viitteellisen toistoajankohdan jokaiselle mediavirran elementille. Tässä tapauksessa mediavirta jaetaan jaksoihin ja jokaisen jakson jälkeen synkronointiagentti lähettää liikkuvat agentit viiveiden laskemiseksi. Jaksojen välissä päivitetään myös synkronointitilastoja ja lasketaan edellisen jakson aikana esiintynyt mediaelementtien kato. Täten voidaan tehokkaammin huomioida järjestelmän väliaikaiset viiveet.

Mediavirran elementille lasketaan jaksoittain viitteellinen toistoajankohta seuraavasti:

- Jos kyseessä on mediavirran alku, on viitteellinen toistoajankohta mediavirran ensimmäisen elementin lähetysajan, lähettäjän ja vastaaottajan kellojen erotuksen, edellisen aikajakson verkkoviiveen, mediaelementin luomiseen kuluvan ajan ja edellisten mediaelementtien luomiseen kuluneen ajan summa. Mikäli kyseessä oleva mediasäikeen edellisen elementin verkkoviive ei ole suurin kaikista säikeistä, lasketaan summaan vielä säikeen edellisen aikajakson aikavääristymä.

- Jos edellisen aikajakson mediaelementtien kato ylittää sallitun rajan ja kyseessä ei ole mediavirran alku, on viitteellinen toistoajankohta edellisen aikajakson viimeisimpänä näytetyn mediaelementin ajankohdan, nykyisen ja edellisen aikajaksojen verkkoviiveiden erotuksen, nykyisen ja edellisen aikajaksojen aikavääristymien erotuksen ja aikajakson aiempien elementtien luomiseen kuuluvien viiveiden summa.

- Muussa tapauksessa viitteellinen toistoajankohta on edellisen aikajakson viimeisimpänä näytetyn mediaelementin ajankohdan ja aikajaksossa jo esiintyneiden elementtien luomiseen kuluvan ajan summa.

Vaikka Manvin ja Venkataramin toteutus tuokin uusia katsantokantoja synkronointitehtäviin, on se mielestäni ylimitoitettu ja jossain määrin virheellinen. Ensinnäkään en usko, että yksinkertaiseen verkkoviiveiden laskemiseen tarvitaan liikkuvia agentteja; se hoituu paljon helpommin ja vähemmillä resursseilla, jos viiveiden mittaamiseen käytetään perinteisiä menetelmiä. Toisekseen agenttien siirtyminen paikasta toiseen on jossain määrin prosessointia vaativa operaatio. Ennen siirtymistä agentti pitää sarjallistaa ja se pitää palauttaa takaisin agentiksi vastaanottavassa päässä. Agentin siirtymisen prosessointiin kuluu siis aikaa ja resursseja, joita ei ilmeisesti oteta huomioon esimerkiksi verkon viiveitä laskettaessa. Itse media kuitenkin siirtyy ilman agentteja.

4.4 Synkronoinnin erityisongelmia

Yleensä synkronointia käsiteltäessä oletetaan, että mediavirtojen vastaanottaja on aina sama. Mediavirrat esimerkiksi saapuvat käyttäjän tietokoneelle, virrat synkronoidaan ja toistetaan tietokoneen multimedialaitteita apuna käyttäen.

1990-luvulla keskityttiin mediavirtojen synkronointiin tilanteessa, missä käyttäjä saa mediaa lähiverkon yli omalle koneelleen. Tämä skenaario olikin ajanmukainen ja toimiva, olihan Internet kehittymässä kovaa vauhtia ja verkostoituneelle medialle aukesi aivan uusia ulottuvuuksia. Tämä oletus ei kuitenkaan ole pätevä nykyaikaisissa käyttötilanteissa, missä jokapaikan teknologia on yleistynyt ja kodit elekronisoituvat kovaa vauhtia.

Jopa tietotekniikan tavalliset julkaisut, kuten Tietokone-lehti [Falck, 2004], käsittelevät nykyajan tietotekniikkaa ja kodin viihde-elektroniikan mahdollisuuksia ja ongelmia. Enää ei olla kaukana tilanteesta, jossa kodin perusvarustukseen kuuluu useita multimedian käsittelyyn tarkoitettuja laitteita: videolaitteita, musiikkisoittimia, kauko-ohjaimia ja kannettavia päätelaitteita.

Liitettävyydestä on tullut nykyajan keskeisimpiä mantroja. Enää ei riitä että laite osaa hoitaa asiansa, vaan sen on pystyttävä liittymään lähiverkkoon, hyödyntämään lähiverkon palveluja ja sen kanssa on pystyttävä keskustelemaan lähiverkon kautta. Joissain laitteissa on jo nykyään kyky jakaa mediaa lähiverkon kautta muille laitteille.

Edellä mainitussa skenaariossa mediaa käsittelevät tahot on hajautettu eri laitteille. Jossain vaiheessa on pyrittävä luomaan medialaitteille yhteinen kommunikointitapa. Tuleva ratkaisu voi hyvinkin olla agenttipohjainen, sillä se ominaisuuksiensa puolesta on kuin luotu vastaaviin tilanteisiin. Joka tapauksessa synkronointiongelma pahenee tilanteessa, missä mediaa vastaanottavatkin eri tahot; laitteiden on pystyttävä synkronoimaan suorituksensa toisten laitteiden kanssa. Tilanne vaikeutuu entisestään, jos halutaan mediaympäristön mukautuvan esimerkiksi käyttäjän sijaintiin. Tällöin medialaitteiden pitäisi pystyä vaihtamaan toistavaa tahoa esimerkiksi silloin, kun käyttäjä liikkuu huoneesta toiseen.

5 Agenttialusta

Agenttipohjainen rakenne oli PROAGENTS-projektin sovellukseen sopiva, koska näin voitiin jakaa suoritukseltaan raskaat prosessit lähiverkossa oleville koneille. Tämä oli tarpeellista, sillä tuntopalautelaite vaatii suuren osan kaksoisprosessorikoneen suorituskyvystä. Raskas kuorma koneella, johon tuntopalautelaite on kytketty, aiheuttaa tuntopalautteen heikentymistä ja esimerkiksi haptisten pintojen pettämistä. Agenttimaisuus edesauttaa myös projektin pidemmän tähtäimen päämäärää jaetusta virtuaalimaailmasta.

Alusta haluttiin pitää helposti muokattavana ja helppotajuisena. On mahdollista rakentaa yksinkertainen hajautettu agenttijärjestelmä muutamassa tunnissa perehtymättä alustan rakenteeseen tarkemmin. Agenttien välinen kommunikaatio noudattaa löyhästi FIPA:n spesifikaatioita [FIPA, 2006a, FIPA, 2006b, FIPA, 2006c] siten, että Agenttialustan käyttämä viestin XML-rakenne vastaa FIPA:n ACL-määrittelyä. Samoin alustan palveluiden perustoiminnallisuus on mukailtu FIPA 97 -spesifikaatiosta.

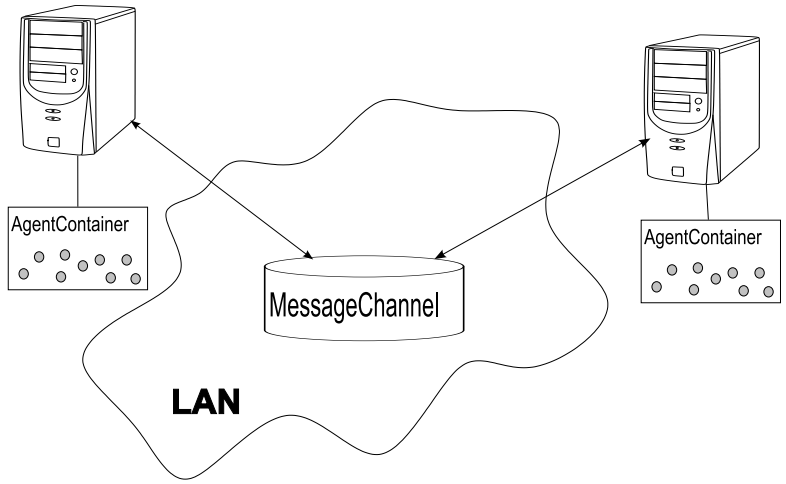
Agenttialusta on mahdollisimman riippumaton sovelluskohteista. Alusta itsessään ei ota kantaa agenttien välissä liikkuvaan tietoon, vuorovaikutustekniikoihin tai erilaisten modaliteettien käyttöön. Tarkoitus on, että eri projekteissa käytettäisiin olemassa olevia agentteja ja mahdollisesti lisättäisiin uutta, multimodaalistakin, toiminnallisuutta uusien agenttien muodossa. Tässä luvussa tutustutaan PROAGENTS-agenttialustaan ja esitellään sen toiminnalliset osat.

Osuuteni projektissa oli agenttialustan arkkitehtuurin suunnittelu ja toteutus. Myöhemmin suunnittelin ja toteutin myös projektin opetusohjelmassa käytetyt komponentit, agentit ja niiden väliset kommunikointiprotokollat. Lopuksi tein opetusohjelman äänijärjestelmän viallisen järjestelmän tilalle ja toteutin opetusohjelmassa tarvittavat erityiset haptiset palautteet, kuten kynän liikuttamisen tiettyihin koordinaatteihin ja kynän täristämisen. Opetusohjelmasta, agenteista ja haptisista palautteista kerron tarkemmin seuraavassa luvussa.

5.1 Alustan toiminnalliset elementit

Loogisesti agenttialusta jakaantuu kolmeen eri toiminnalliseen osaan (Kuva 5.1): agentit, agenttisäiliöt (AgentContainer) ja viestikanava (MessageChannel). Agentit ovat agenttijärjestelmän toiminnallinen ydin. Kaikki muu on agenttien toimintaa avustavaa infrastruktuuria. Agentteja ajetaan agenttisäiliöissä, mikä vastaa agenttien ylläpidosta ja tarjoaa agenteille tiettyjä palveluja. Agenttisäiliöitä voi

olla useita ja ne yleensä sijaitsevat lähiverkkoon yhdistetyillä tietokoneilla. Agenttialustoja yhdistää viestikanava, joka toimii keskitettynä tietoliikenteen palvelimena agenttisäiliöiden välissä.



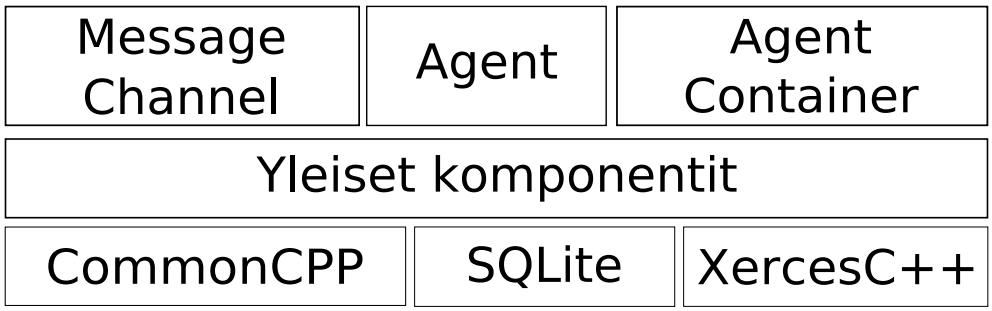
Kuva 5.1 Agenttialustan toiminnalliset osat

Viestintä tapahtuu pääosin keskitetysti viestikanavan kautta, ellei viestin vastaanottaja ole samassa säiliössä. Viestiessään agentti luo viestiolion ja antaa sen agenttisäiliölle lähetettäväksi. Agenttisäiliö muuttaa viestin XML-muotoon ja lähettää sen viestikanavalle. Viestikanava lähettää viestin edelleen agenttisäiliölle, jossa vastaanottaja sijaitsee. Vastaanottava agenttisäiliö muuntaa viestin takaisin viestiolioksi ja toimittaa sen vastaanottavan agentin postilaatikkoon. Saman säiliön sisällä viesti siirtyy viestioliona. Yleinen XML-viestimuoto mahdollistaa eri ohjelmointikielillä tehtyjen ja myös eri käyttöjärjestelmissä ajossa olevien Agenttisäiliöiden yhteyden samaan viestikanavaan. Tällä hetkellä Agenttisäiliöstä on sekä C++- että Python-toteutukset.

Monimutkaisemmissa järjestelmissä tulee tarpeelliseksi antaa agentille mahdollisuus omien tietoliikenneyhteyksien muodostamiseen, sillä viestikanavasta muodostuu nopeasti järjestelmän pullonkaula. Arkkitehtuuri mahdollistaakin yhteyden toiseen agenttiin sekä TCP/IP- että UDP-tietoliikenneprotokollien avulla. Arkkitehtuuri ei ota kantaa siihen, miten yhteys saadaan aikaiseksi. Agentti voi esimerkiksi aloittaa dialogin toisen agentin kanssa viestikanavan avulla ja sopimukseen päästyään agentit voisivat keskustella suoraan keskenään.

5.2 Arkkitehtuurin rakenne

Agenttialustaa suunniteltaessa pyrittiin käyttämään mahdollisimman paljon hyväksi valmiita kirjastoja. Arkkitehtuuri rakentuu kolmen avoimeen lähdekoodiin perustuvan kirjaston päälle. Ensinnäkin järjestelmän kaikki tietokantaominaisuudet on rakennettu SQLite tietokantakirjaston päälle [SQLite, 2006]. SQLite on kevyt ja helposti mukautettava kirjasto, joka mahdollistaa SQL-tietokantojen luomisen sekä levylle että muistiin. XercesC++ on vuorostaan kirjasto XML-koodattujen tiedostojen tekemiseen ja käsittelyyn [Xerces C++, 2006]. Suurimman osan tarvittavasta perustoiminnallisuudesta toteuttaa CommonCPP kirjasto, josta löytyy käyttöjärjestelmäriippumattomat toteutukset mm. säikeille ja verkkoliikenteelle [COMMON C++ library, 2006]. Kuvassa 5.2 on arkkitehtuuri kuvattu tasomallina.



Kuva 5.2 Agenttiarkkitehtuuri

5.3 MessageChannel

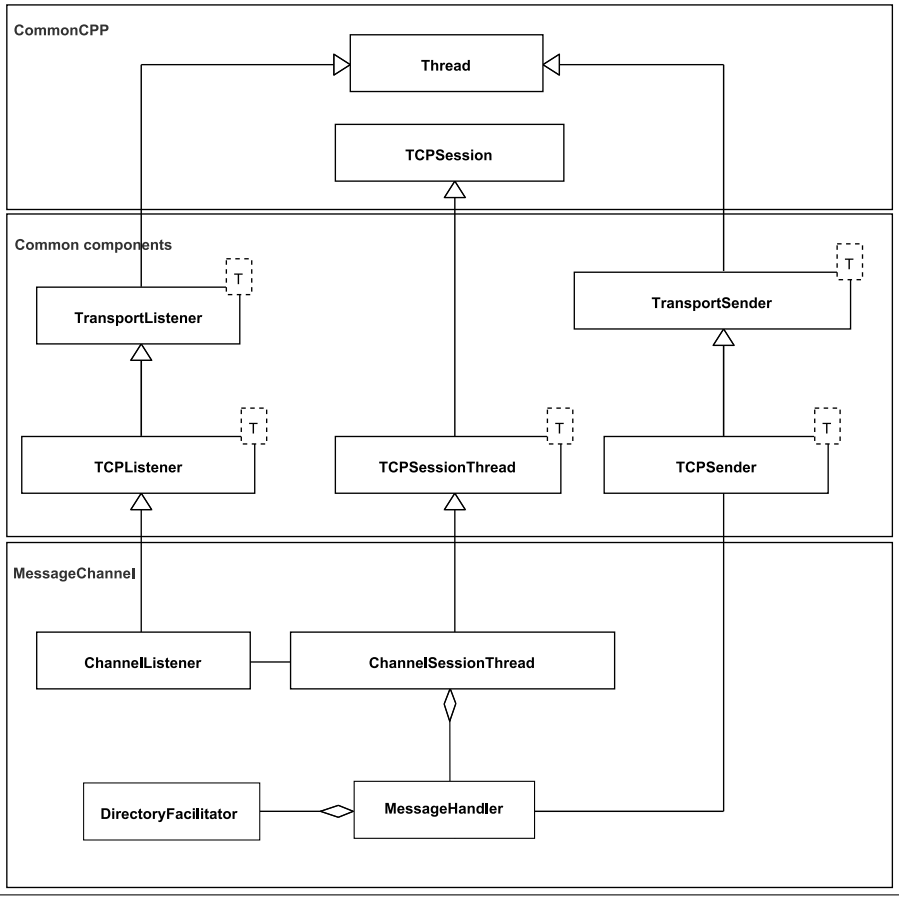
Viestikanava (MessageChannel) on agenttien välisen viestinnän keskeisin elementti. Viestikanava toimii keskitettynä viestien välittäjänä ja vastaa jaetuista hakemistopalveluista. Hakemistopalvelujen avulla agentit voivat esimerkiksi etsiä agentteja, jotka tarjoavat jotain tiettyä palvelua. Keskeinen ongelma hajautetuissa järjestelmissä on se, miten eri elementit saavat yhteyden toisiinsa mahdollisimman vähillä ennakkotiedoilla muiden elementtien sijainnista. Yleinen ratkaisu on käyttää keskitettyä viestinnän mahdollistajaa, johon kaikki elementit ottavat yhteyden. Täten elementtien ei tarvitse tietää kuin keskitetyn viestipalvelimen yhteystiedot. Viestikanava toimii juuri tällaisena keskitettynä viestipalvelimena.

Viestikanava pitää kirjaa olemassa olevista säiliöistä, säiliöiden tukemista tietoliikenneprotokollista ja agenteista, agenttien sijainnista ja niiden tarjoamista

palveluista. Yleisin viestikanavan tarjoama toiminnallisuus on tietyn palvelun tarjoavien agenttien kysyminen. Eli agentti voi kysyä viestikanavalta, mitkä agentit tarjoavat esimerkiksi lokipalvelua. Vastauksena agentti saa listan palvelun tarjoavista agenteista, jota se voi käyttää jatkossa esimerkiksi kommunikaatiotapahtumia käynnistettäessä.

5.3.1 Viestikanavan rakenne

Viestikanava koostuu tietoliikennettä kuuntelevasta komponentista (ChannelListener), joka käynnistää jokaiselle yhteydenotolle oman säikeensä yhteydenoton käsittelyyn (Kuva 5.3). ChannelListener on TCPListener-luokan aliluokka. TCPListener vuorostaan on abstraktin TransportListener-luokan aliluokka. TransportListener kuvaa yhteisen rajapinnan eri tietoliikenneprotokollien kuuntelemista varten. Tällä hetkellä toteutettuna on TCP/IP- ja UDP-protokollat.



Kuva 5.3 Viestikanavan rakenne yleisellä tasolla kuvattuna

ChannelListener luo ChannelSessionThread-olion jokaiselle sisääntulevalle yhteyspyynnölle. ChannelSessionThread vastaanottaa saapuvan datan ja antaa sen edelleen luomalleen MessageHandler-oliolle. ChannelSessionThread periytyy TCPSessionThread-luokasta, joka vuorostaan on laajennus CommonCPP:n TCPSession-luokasta ([COMMON C++ library, 2006]).

MessageHandler käsittelee viestin ja käyttää viestin tai vastauksen lähettämiseen TCPSender-luokan instanssia. TransportSender on laajennus abstraktista TransportSender luokasta, joka määrittelee yhteisen rajapinnan viestien lähettämiselle protokollasta riippumatta.

Jokainen MessageHandler-olio käyttää omaa instanssia DirectoryFacilitator-luokasta, joka toimii rajapintana MessageHandler-olion ja jaetun SQL-tietokannan ([SQLite, 2006]) välissä. DirectoryFacilitator vastaa niistä operaatioista, jotka lisäävät, hakevat tai muuttavat tietokannan sisältöä.

Viestikanavalle kehitettiin projektin loppupuolella myös graafinen käyttöliittymä (kuva 5.4). Sen avulla käyttäjä pystyy tarkkailemaan järjestelmässä tapahtuvaa tiedonvälitystä ja kontrolloimaan agenttien elinkaaritoimintoja (käynnistys, lopetus, pysäytys ja pysäytyksestä jatkaminen).

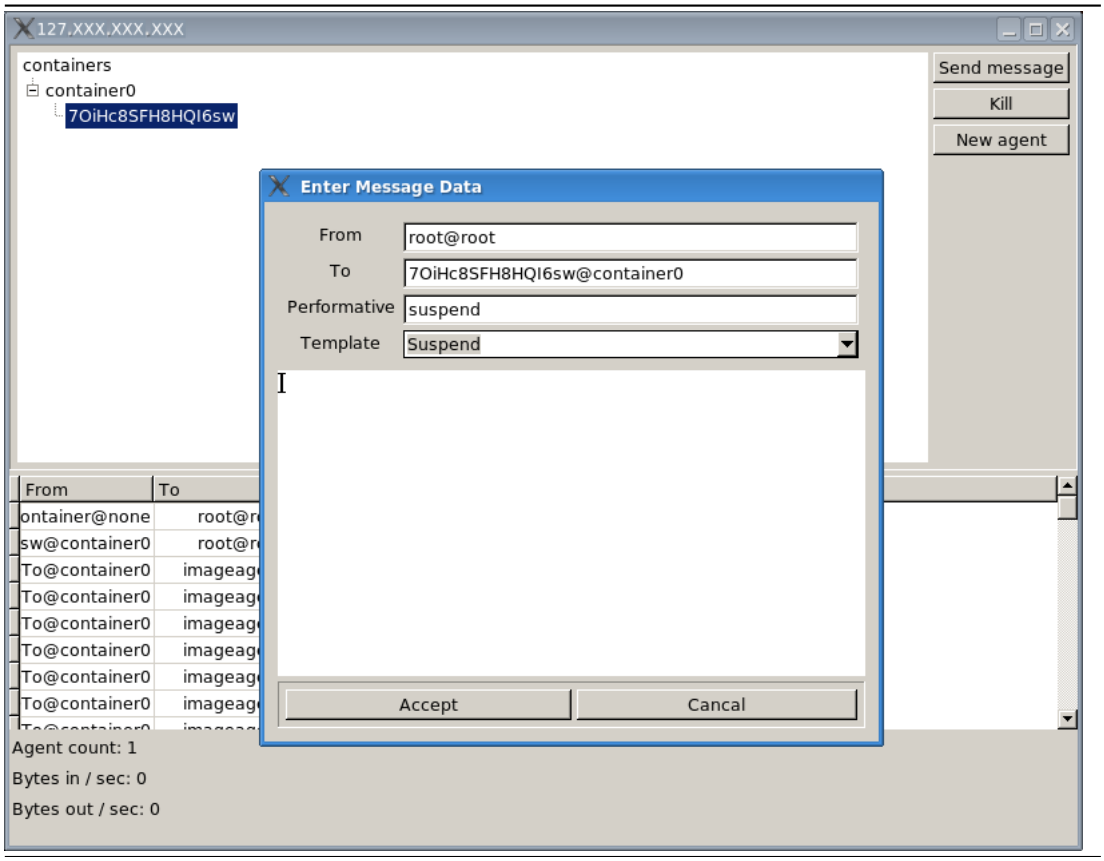
5.3.2 Agenttisäiliöiden liittäminen järjestelmään

Säiliöillä on yksinkertainen XML-konfigurointitiedosto, jonka muokkaaminen riittää niiden toimintakuntoon saattamiseksi. Tiedosto sisältää viestikanavan IP-osoitteen ja sen portin numeron, mitä viestikanava kuuntelee. Lisäksi tiedostossa määritellään agenttisäiliön kommunikointiin käyttämä portin numero.

Käynnistyessään agenttisäiliö kirjautuu viestikanavalle. Säiliö antaa kirjautuessaan listan niistä tietoliikenneprotokollista, mitä säiliö tukee ja kuuntelee. Tällä hetkellä vain TCP/IP-protokolla on tuettuna, mutta on helppo lisätä muitakin protokollia järjestelmään laajentamalla abstrakteja TransportListener- ja TransportSender-luokkia, jotka määrittelevät yhteiset rajapinnat kaikille protokollille.

Useiden tuettujen protokollien tarkoitus on mahdollistaa käytössä olevan protokollan vaihdon, mikäli se lakkaa toimimasta. HTTP:n kaltaisten protokollien käyttö mahdollistaisi myös liikenteen säiliöihin, jotka ovat palomuurien takana tai joiden liikennöintimahdollisuuksia on rajoitettu muista syistä.

Kirjautumisprosessin (kuva 5.5) lopuksi viestikanava antaa säiliölle ainutlaatuisen nimen, jota agenttisäiliö ja siinä olevat agentit käyttävät säiliön ollessa toiminnassa.



Kuva 5.4 Viestikanavan graafinen käyttöliittymä

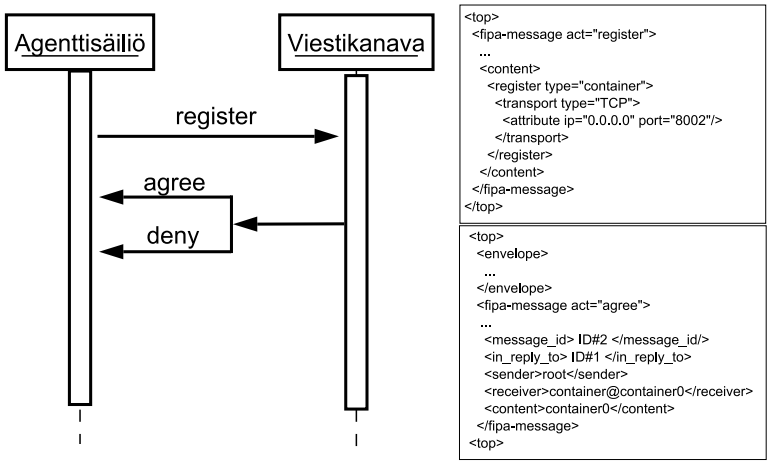
5.3.3 Agenttien ylläpito

Agentti lähettää viestikanavalle käynnistyessään tarjoamiensa palveluiden tiedot (kuva 5.6). Palvelu on merkkijono, jolla agentti voi mainostaa ominaisuuksiaan muille agenteille. Viestikanava pitää listaa kutakin palvelua tarjoavista agenteista. Agentit voivat kysyä viestikanavalta listaa tiettyä palvelua tarjoavista agenteista. Sekä agentit että säiliöt poiskirjautuvat viestikanavasta lopettaessaan.

5.3.4 Yleiset operaatiot ja perusprotokollat

Perusprotokollien joukko on pieni ja kattaa vain kaikkein oleellisimman toiminnallisuuden. Perusprotokolliin kuuluvat edellä mainitut protokollat agenttien ja Agenttisäiliöiden viestikanavaan kirjautumiseksi. Lisäksi perusprotokollien joukkoon kuuluvat myös protokollat uusien agenttien käynnistämiseen, agentin toiminnan pysäyttämiseeen, pysäytyksestä jatkamiseen, agentin toiminnan lopettamiseen ja agenttien ja säiliöiden viestikanavasta poiskirjautumiseen.

Viestikanavaan on myös rakennettu erilaisia palveluita, joista tietyn palve-



Kuva 5.5 Agenttisäiliön kirjautumisprotokolla

lun tarjoavien agenttien kysyminen esiteltiin aiemmin. Sen lisäksi, että agentti voi lähettää viestejä toisille agenteille, voi agentti myös lähettää viestin kaikille tietyn palvelun tarjoaville agenteille asettamalla palvelun nimen viestin vastaanottajaksi. Viesti voidaan myös osoittaa kaikille agenteille asettamalla viestin vastaanottajaksi 'broadcast'.

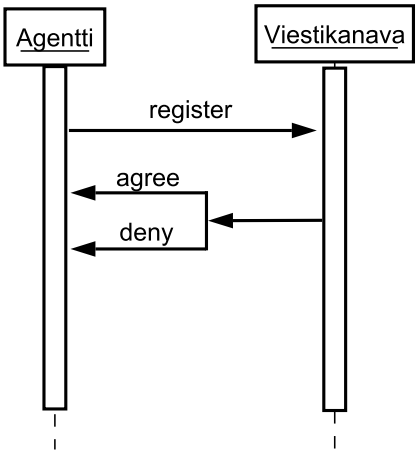
5.4 AgentContainer

Agenttisäiliö (AgentContainer) toimii agenttien ajoalustana ja tarjoaa yksinkertaiset postipalvelut agenteille. Postipalveluihin kuuluu mm. agentin omakohtainen postilaatikko ja yksinkertaiset viestin lähetysoperaatiot. Säiliö myös kontrolloi agentin elinkaarta mahdollistaen agenttien luonnin, tuhoamisen, pysäyttämisen ja pysäytyksestä jatkamisen. Agenttisäiliö antaa agentille ainutlaatuisen nimen agentin luontihetkellä, ellei agentille ole määritelty nimeä agentin konfiguraatiotiedostossa.

5.4.1 Agenttisäiliön rakenne

Kuten Viestikanavallakin, on Agenttisäiliöllä oma TCPListener-luokan instanssi, joka vastaanottaa kaikki säiliölle ja säiliön agenteille tulevat viestit (kuva 5.7). Säiliön TCPSender instanssi vuorostaan vastaa viestien lähettämisestä edelleen viestikanavaan. Yleisessä tapauksessa agentit käyttävät säiliön tarjoamia viestinvälityspalveluita.

Viitteet säiliössä oleviin agentteihin talletetaan ElementContainer-luokan ins-




```
<top>
 <fipa-message act="register">
  ...
  <conversation_id> ID#1 </conversation_id>
  <sender>agent@containerX</sender>
  <receiver>root@root</receiver>
  <content>
   <register type="agent">
    <service name="name1"/>
    ...
    <service name="nameN"/>
   </register>
  </content>
 </fipa-message>
</top>
```

```
<top>
 <envelope>
 ...
 </envelope>
 <fipa-message act="agree">
  ...
  <sender>root</sender>
  <receiver> agent@containerX </receiver>
  <conversation_id> ID#2 </conversation_id>
  <in_reply_to> ID#1 </in_reply_to>
  <content>
   <register type="agent">
    <service name="name1"/>
    ...
    <service name="nameN"/>
   </register>
  </content>
 </fipa-message>
<top>
```

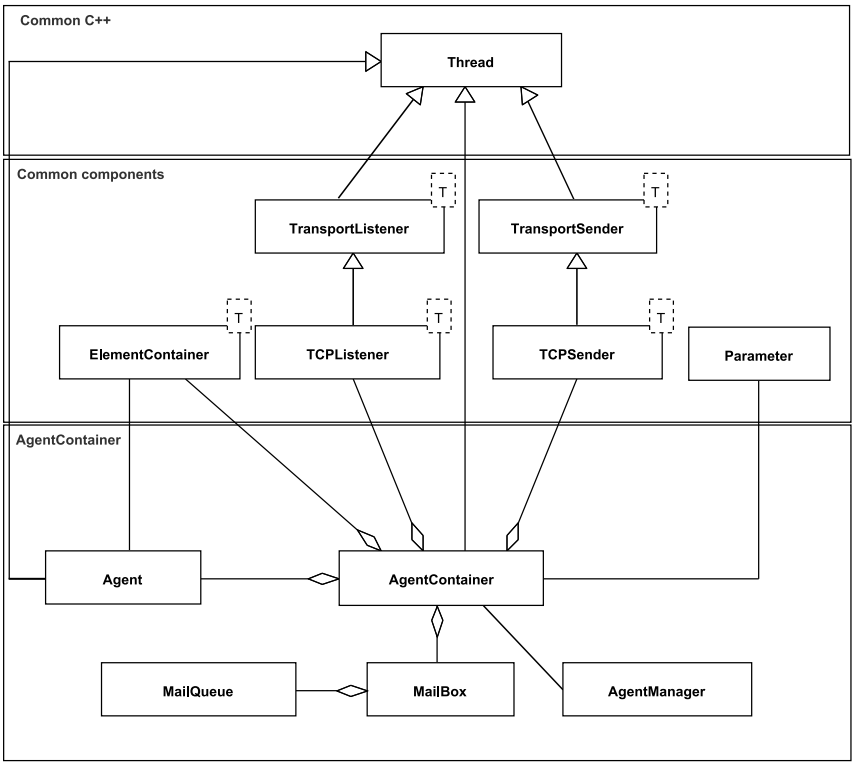
Kuva 5.6 Agentin kirjautumisprotokolla

tanssiin. ElementContainer on laajennettu säiliöluokka, joka tarjoaa muutamia alustaspesifejä operaatioita. ElementContainer mm. luo lisätylle elementille satunnaisen avaimen, jonka avulla elementin saa haettua säiliöstä. Tässä tapauksessa avainta käytetään myös agentin nimenä.

Agenttisäiliö luo jokaiselle agentille postilaatikon saapuvien viestien säilytykseen. Mailbox-luokka sisältää kaikkien agenttien postilaatikot (MailQueue). Postilaatikon viestit on lajiteltu saapumisjärjestykseen viestiin talletetun aikaleiman mukaan.

Agenttien lisääminen järjestelmään on pyritty tekemään mahdollisimman helpoksi ja yksinkertaiseksi. Alunperin uutta agenttia lisättäessä piti agenttisäiliön lähdekoodeja muokata agentin toimintaan saattamiseksi, mutta alustan myöhemmissä versioissa on jo käytettävissä dynaamisiin kirjastoihin perustuva liitännäisratkaisu, joka mahdollistaa agenttien kehittämisen täysin erillään agenttisäiliöstä. AgentManager on luokka, joka vastaa annetun nimen mukaisen agentin lataamisen kirjastosta ja uuden agentti-instanssin palauttamisesta agenttisäiliölle (kuva 5.8). AgentManager ja siihen liittyvät apuluokat ja tietorakenteet pohjautuvat C++ Users Journal lehdessä olleeseen artikkeliin [Musgrove, 2004].

Agenttisäiliöllä on konfiguraatiotiedosto, jossa määritellään viestikanavan yhteystiedot ja käynnistettävien agenttien konfiguraatiotiedostot (esimerkki 5.1).



Kuva 5.7 Agenttisäiliön rakenne yleisellä tasolla

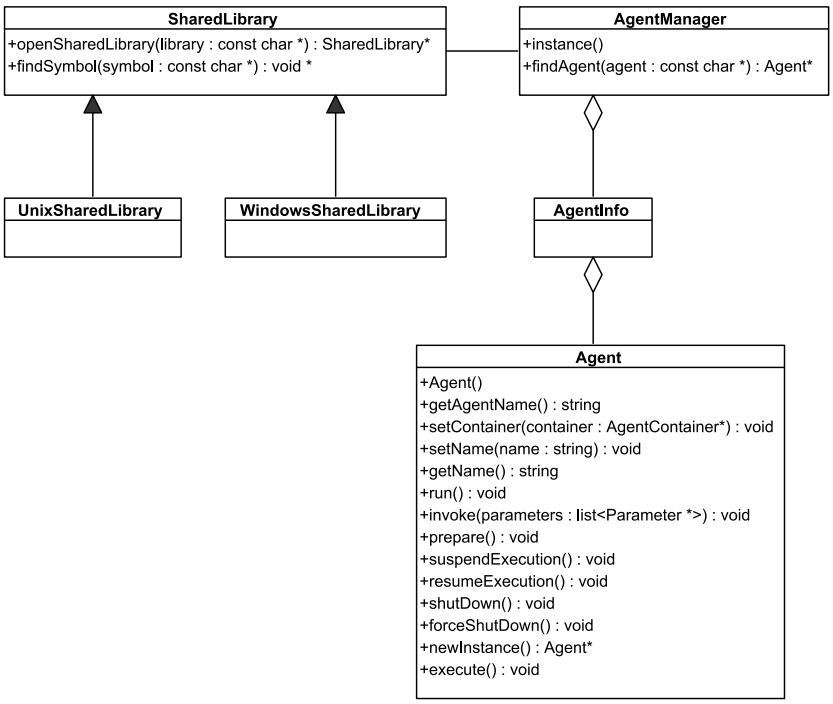
Säiliö lataa agenttien konfiguraatiotiedostot ja luo niiden mukaan uudet agentit. Agenttisäiliö voi luoda agentteja myös ajon aikana.

Esimerkki 5.1: Agenttisäiliön konfiguraatiotiedosto

```
<?xml version="1.0" ?>
<container>
  <server ip="0.0.0.0" port="8002"/>
  <port port="8006"/>
  <agents>
    <agent file="filterspec.xml"/>
  </agents>
</container>
```

5.5 Agentit

Käynnistyksen yhteydessä agentille voidaan antaa parametreja. Parametrit määritellään agentin konfiguraatiotiedostossa (esimerkki 5.2), josta löytyy myös tieto käynnistettävästä agentista. Agenttia ajetaan erillisessä säikeessä. Arkkitehtuuri



Kuva 5.8 Agenttiliitännäisten arkkitehtuuri

sallii agentin käynnistää uusia säikeitä. Agentin kuitenkin tulee itse tällöin vastata niiden tuhoamisesta.

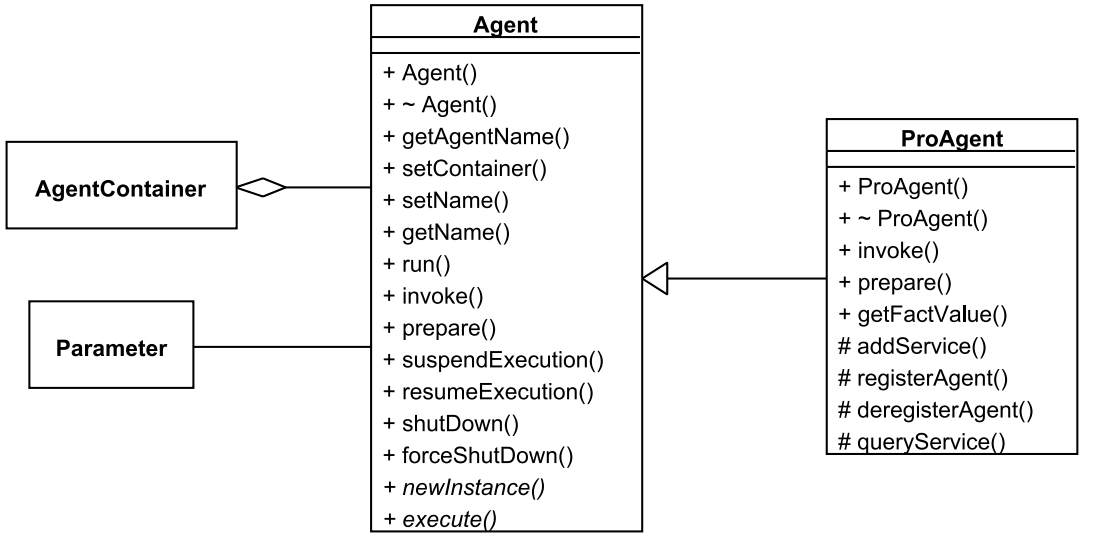
Esimerkki 5.2: Agentin konfiguraatiotiedosto

```
<?xml version="1.0" ?>
<agent class="ClassName">
  <parameter type="string" name="param" value="string"/>
</agent>
```

Järjestelmän tarjoama agenttien yläluokka tarjoaa muutamia valmiita toimintoja. Esimerkiksi tuhoutuessaan agentti automaattisesti kirjautuu pois viestikanavasta ja ilmoittaa tuhoutumisestaan säiliölleen. Tämän lisäksi agentilla on rajapinta, jonka avulla sille voidaan helposti määritellä agentin tarjoamat palvelut ja ne voidaan rekisteröidä viestikanavaan.

5.5.1 Agentin rakenne

Kuvassa 5.9 on yleisellä tasolla kuvattu agentin rakennetta. ProAgent, abstraktin Agent-yläluokan laajennus, toteuttaa joitakin PROAGENTS-projektissa tärkeitä toiminnallisuuksia, ja täten kaikki projektin agentit perivät ProAgent yläluokan.



Kuva 5.9 Agentin rakenne kuvattuna UML-kaaviolla

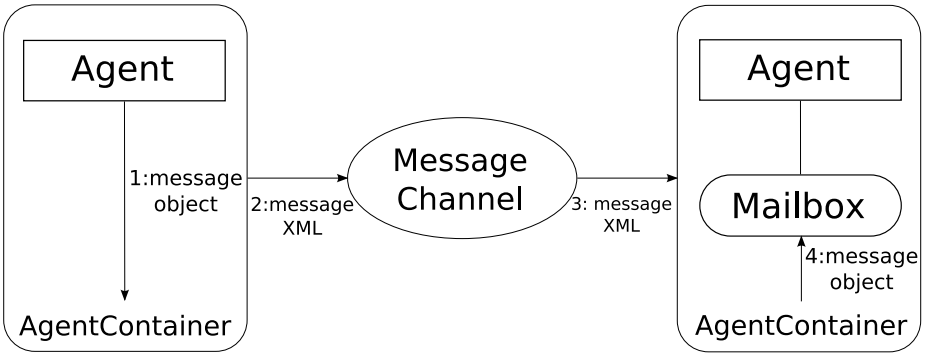
5.6 Esimerkki alustan toiminnasta

Kuten aiemmin mainittiin, käynnistyessään agenttialusta kirjautuu viestikanavalle ja saa siltä ainutlaatuisen nimen. Viestikanava tallettaa agenttialustan tiedot käyttääkseen niitä myöhemmin. Kirjautumisen jälkeen agenttialusta lataa sen käynnistystiedostossa määritellyt agenttien konfiguraatiotiedostot. Näiden tiedostojen avulla alusta luo ja käynnistää käynnistettäväksi määritellyt agentit. Luontihetkellä säiliö luo agentille ainutlaatuisen nimen. Jokaiselle agentille luodaan myös oma postilaatikko agentille tuleville viesteille. Agentti vuorostaan käynnistyessään kirjautuu viestikanavalle ja samalla ilmoittaa tarjoamansa palvelut.

Tässä esimerkissä agentti lähettää toisessa säiliössä olevalle agentille viestin. Kuvassa 5.10 on kuvattuna viestinvälitystapahtuma yleisellä tasolla ja kuvassa 5.11 on kuvattu operaation sekvenssikaavio yleisluontoisesti. Ennen viestin lähettämistä viestiä lähettävä agentti on hankkinut vastaanottaja-agentin nimen selville. Aluksi agentti luo viestiolion ja antaa sen säiliölle toimitettavaksi.

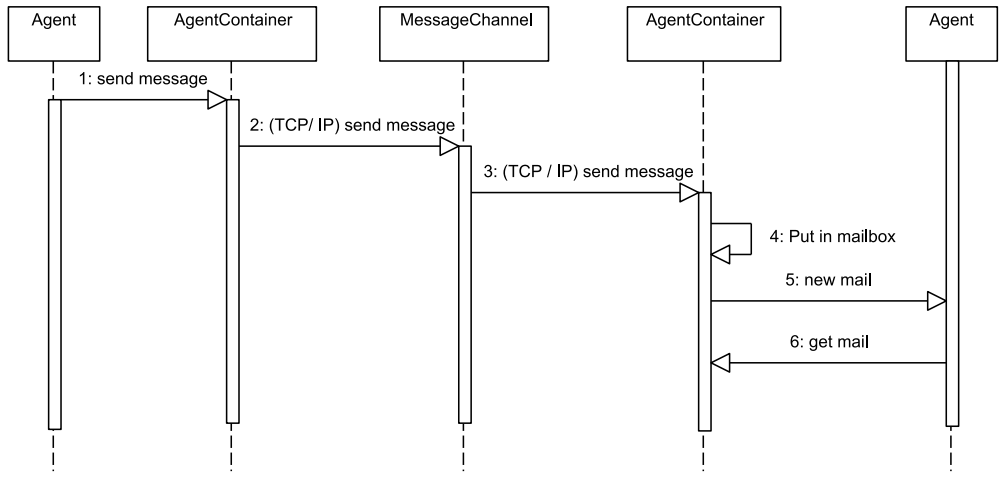
Agenttisäiliö, todettuaan että vastaanottaja sijaitsee toisessa säiliössä, muuntaa viestin XML-muotoon, kiinnittää siihen kirjekuoren, mikä sisältää viestiin liittyvää tietoa ja lähettää sen viestikanavalle. Viestikanava tutkii kirjekuorta ja käy läpi vastaanottajien listan etsien vastaanottajan säiliötietoja. Viestikanava

lähettää viestin edelleen sille säiliölle, jossa vastaanottaja on.



Kuva 5.10 Viestin kulku

Vastaanottava agenttisäiliö saa viestin ja kirjekuorta tutkimalla päättelee varsinaiset vastaanottaja-agentit. Viestistä luodaan viestiolio, joka kopioidaan vastaanottajan postilaatikkoon. Vastaanottava agentti tutkii postilaatikkoaan, saa viestin ja toimii sen mukaan.



Kuva 5.11 Agenttialustan toiminta sekvenssikaaviona

Agentilla on kaksi tapaa tarkkailla postilaatikkoaan. Ensinnäkin agentti voi katsoa, onko postilaatikossa viestejä. Toisekseen agentti voi tarkistaa postin tilanteen siten, että agentti jää odottamaan seuraavaa viestiä postilaatikon ollessa tyhjä.

Kuten esimerkistä huomataan, on agenttien välinen kommunikaatio tehty mahdollisimman helpoksi ja virtaviivaiseksi. Agentit käsittelevät vain viestiolioita ja täten niiden ei tarvitse välittää esimerkiksi viestin muodosta tai eri lähetysta-

voista. Viestin muuttaminen XML-muotoon lisää hieman viestien prosessointiaikaa, mutta se myös mahdollistaa alustan hyvän liitettävyyden.

6 Moniaistinen opetusohjelma

Edellä kuvattu arkkitehtuuri suunniteltiin PROAGENTS-projektin yhteydessä. Projektin päämääränä oli rakentaa opetusohjelma, joka olisi käyttökelpoinen niin sokeille kuin näkevillekin lapsille. Pidemmän ajan tähtäimenä oli ryhmätyön mahdollistaminen sokeiden ja näkevien lapsien välillä. Projektissa alustaa laajennettiin siten, että käyttäjän tietokoneella on suorituksessa erikoistettu Agenttisäiliö ja sen yhteydessä itse opetusohjelma ja PHANToM-tuntopalautelaite [SensAble Technologies Inc, 2006].

PHANToM-tuntopalautelaite on useilla moottoreilla varustettu nivelletty laite (kuva 6.1), jonka päähän on kiinnitetty kynän muotoinen esine. Kynänmuotoista esinettä käyttäen voidaan kosketella kolmiuloitteisessa virtuaalimaailmassa olevia esineitä. Kynän tilalle voidaan kiinnittää muitakin esineitä, kuten pallo tai sormen päähän kiinnitettävä hattumainen otin. Kosketeltavilla kappaleilla on erilaisia ominaisuuksia, kuten massa, pinnan kimmoisuus ja kitka. Kappale voi olla myös liikkeessä. Kappaleen liikkeeseen on mahdollista vaikuttaa kynän avulla.

Laite toimii vastavoimien periaatteella laskien käyttäjän kynään kohdistavien voimien suunnan ja voimakkuuden. Mikäli kynän liikeradalle on kuvattu kiinteitä kappaleita, lasketaan kynään kohdistuneelle voimalle vastavoima, joka ottaa huomioon kappaleen pinnan ominaisuudet. Voima ohjataan laitteeseen useiden pienten moottoreiden avulla. Kynään voidaan myös kohdistaa muunlaisia voimia, kuten kynää puoleensa vetäviä jousivoimia, magnetismin kaltaisia voimia ja tietyn suuntaisia ja voimakkuuksisia voimavektoreita.

6.1 PROAGENTS-Minimaailmat

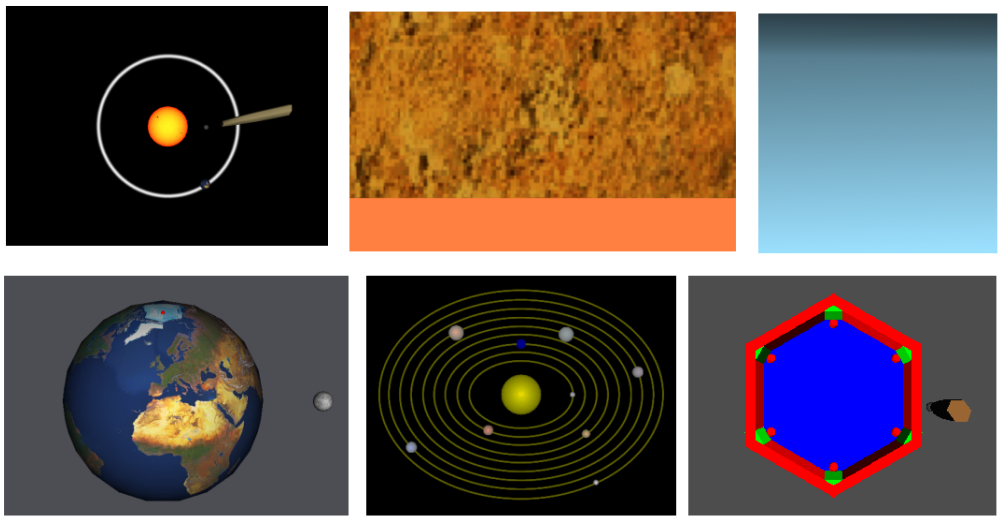
Projektissa rakennetun opetusohjelman teemoja oli Maan suhde aurinkoon ja vuodenajat, maan kerrokset, ilmakehä, Maa ja Aurinkokunta (kuva 6.2). Jokainen teema muodostaa oman virtuaalimaailmansa. Kutsumme yksittäistä virtuaalimaailmaa minimaailmaksi. Kuvassa on myös jälkeenpäin lisätty keskusasema-minimaailma.

Sovelluksen luonne ja kohderyhmä asettivat sen suunnittelulle selviä suunnittelukriteereitä. Esimerkiksi vapaa kolmiulotteinen navigointi suurissa virtuaalimaailmoissa ei sovellu kohderyhmän käytettäväksi. Myös kosketeltavien kappaleiden ominaisuudet oli harkittava tarkkaan, jotta ne olivat selkeästi löydettävissä ja tunnistettavissa ilman näköaistiakin. Toisaalta kappaleilla tuli olla myös visuaalisesti kiinnostava ulkomuoto, jotta täysin tai osittain näkevien käyttäjien mie-

Kuva 6.1 PHANToM-tuntopalautelaite

lenkiinto säilyisi. Visuaalinen ilme myös tukee osittain näkevän lapsen sovelluksen käyttöä. Puuttuvaa aistia korvataan tunto- ja äänipalauteen avulla. Minimaailmat on suunniteltu siten, että niissä on mahdollisimman helppo navigoida ilman näköaistia. Suurin osa tuntopalautteesta on kuvattu minimaailmojen määrittelyn yhteydessä, kun taas äänipalaute on osittain määritelty minimaailmojen määrittelyn yhteydessä ja on luonteeltaan dynaamista ja käyttäjän toimiin liittyvää.

Minimaailmojen suunnittelu pohjaa mm. Patomäen ja kumppaneiden [Patomäki *et al.*, 2004] ja Sjöströmin [Sjöström, 2001] tuloksiin. Pyrimme esimerkiksi pitämään virtuaalimaailmojen objektit mahdollisimman yksinkertaisina niin kuin Patomäki *et al.* ehdottavat. Sjöström vuorostaan ehdottaa, että sokeille tehdyssä virtuaalimaailmassa tulisi olla kiintopisteitä, mitkä helpottavat käyttäjän navigointia. Lisäsimmekin sovellukseen kuudennen minimaailman, keskusaseman, jonka kautta käyttäjällä on pääsy kaikkiin muihin minimaailmoihin. Minimaailmat järjestettiin niin, että käyttäjä joutuu kulkemaan keskusaseman kautta siirtyes-



Kuva 6.2 Opetusohjelman minimaailmoja. Ylärivi: Maa ja aurinko, maan rakenne ja ilmakehä. Alarivi: Maa, Aurinkokunta ja tutkimusasema.

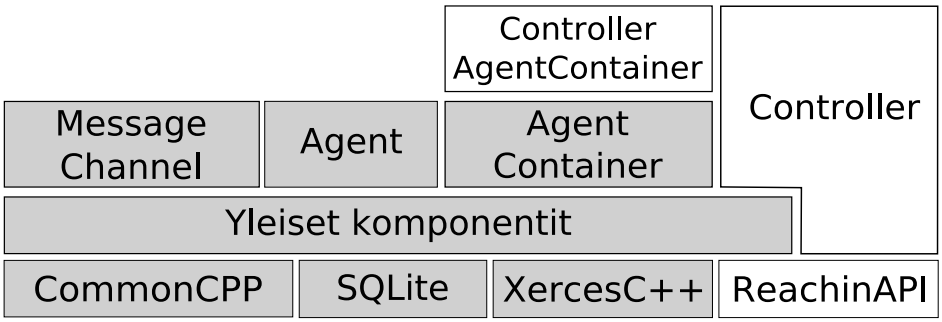
sään minimaailmasta toiseen. Näin käyttäjä on aina korkeintaan yhden askeleen päässä keskusasemasta ja eksymisen vaara on pieni. Käyttäjä pääsee suoraan siirtymään keskusasemalle painonapin avulla. Navigointia tuettiin myös siirtämällä PHANToM-laitteen kynäosa aina samaan kohtaan käyttäjän siirtyessä minimaailmasta toiseen. Tämä antaa käyttäjälle aina saman lähtöpisteen minimaailman tutkimiseen ja estää ikävien tuntopalautepiikkien esiintymisen, joita saattaa esiintyä esimerkiksi silloin kun kynän kohdalle luodaan kiinteä kappale. Sovelluksen käyttö aloitetaan aina keskusasemalta.

Minimaailmoissa on tuntopalautetta käytetty asioiden havainnoillistamisen lisäksi myös käytön helpottamiseen. Esimerkiksi Maa-minimaailmassa tuntopalautelaitteeseen on liitetty jousivoima, joka vetää kynää kevyesti kohti Maan keskipistettä. Tämä havainnoillistaa painovoimaa ja auttaa käyttäjää löytämään mielenkiintoiset asiat minimaailmasta. Aurinkokunta-minimaailmassa vuorostaan käyttöalueen syvyyttä on rajattu tasolla, jonka pinnalla tutkittavat planeetat ja kiertoradat ovat. Kiertoradat ovat uurteita tasolla ja planeetat pallomaisia, lievästi magneettisia, elementtejä.

6.2 Kontrolli ja PHANToM

Pääohjelma, Kontrolli (Control), on rakennettu niin, että sillä on yhteys sekä käytettävään tuntopalautelaitteeseen että erikoistetun agenttialustan avulla myös

järjestelmän muihin agentteihin. Vastavuoroisesti järjestelmän muut agentit voivat olla yhteydessä Kontrolliin sen agenttisäiliössä olevien edustaja-agenttien välityksellä. Kontrolli rakennettiin osittain agenttialustan ja osittain tuntopalautelaitteen kaupallisen ohjelmointiympäristön, Reachin API:n [Reachin API, 2006], päälle (kuva 6.3).



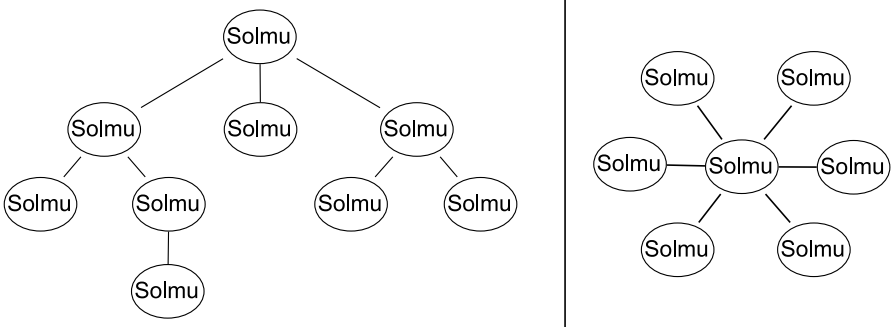
Kuva 6.3 Agenttialustan laajennukset PROAGENTS-projektissa

Kuten edellä mainittiin, jaettiin jokainen opetusohjelman teema omaksi erilliseksi minimaailmaksi. Minimaailmojen käyttö sovelluksessa suunniteltiin suhteellisen vapaaksi. Ajatuksena oli, että minimaailmoja voitaisiin käyttää ja yhdistää mahdollisimman vähin muutoksin itse minimaailman kuvaukseen. Tässä ympäristössä sovellusohjelma voidaankin nähdä joukkona minimaailmoja, joista on pääsy toisiin minimaailmoihin. Kukin maailma vastaa yhtä solmua maailmojen verkko- tai puumaisessa rakenteessa ja yhteys maailmasta toiseen on siirtymä verkon solmusta toiseen.

Jokainen virtuaalimaailma määritellään erillisessä VRML -määrittelytiedostossa (Virtual Reality Markup Language) [The VRML Consortium Incorporated, 2006]. Minimaailmojen välisiä suhteita kuvataan XML-tiedostossa, jossa voidaan määritellä kahdenlaisia suhteita maailmojen välillä. Ensinnäkin virtuaalimaailmat voidaan järjestää puumaiseen rakenteeseen, jolloin minimaailmasta on suora pääsy sen lapsimaailmoihin ja hierarkiassa ylempänä olevaan isämaailmaan. Hierarkisilla suhteilla voidaan esimerkiksi jakaa minimaailmat kontekstuaalisesti samankaltaisiin ryppäisiin. Toisekseen maailmojen välillä voi olla suora yksisuuntainen reitti hierarkiasta riippumatta.

Hierarkkisille siirtymille on sovellukseen rakennettu navigointipaneeli ja suorille reiteille pitää määritellä virtuaalimaailmaan painonapin kaltainen pinta, jota painamalla siirtyminen toiseen virtuaalimaailmaan tapahtuu. Kuvassa 6.4 näkyy sekä hierarkkinen että verkkomainen rakenne. Näiden edeltäkäsin suunniteltujen

siirtymien lisäksi agenteilla on mahdollisuus siirtää käyttäjä suoraan minimaailmasta toiseen.



Kuva 6.4 Hierarkkinen ja verkkomainen minimaailmojen rakenne

Kontrolli lukee käynnistyessään virtuaalimaailmojen suhteita kuvaavan XML-tiedoston, jossa on myös jokaisen virtuaalimaailman määrittelevän VRML-tiedoston nimi. Tiedoston pohjalta kontrolli luo sisäisen navigointirakenteen ja lataa keskusaseman virtuaalimaailman näytettäväksi. Kontrolli näyttää yhden virtuaalimaailman kerrallaan. Maailma ladataan VRML-tiedostosta (esimerkki 6.1), mikäli sitä ei löydy kontrollin ylläpitämästä minimaailmojen puskurista. PRO-AGENTS-projektissa navigointipaneeli oli kytkettynä pois päältä ja sovellus rakennettiin kokonaan reittien varaan niin, että keskusasemalta oli reitti kaikkiin muihin minimaailmoihin. Muissa minimaailmoissa ei ollut suoraa pääsyä muihin minimaailmoihin kuin takaisin keskusasemalle. Muutamissa erikoistapauksissa agentit käynnistivät suoran siirtymisen minimaailmasta toiseen.

Virtuaalimaailman VRML-määrittelytiedosto voi sisältää myös Python-skriptikieltä. Esimerkissä 6.1 luodaan yksinkertainen painonapin kaltainen pallo. Virtuaalimaailman yhteydessä ladataan myös Python-skripti.

Esimerkki 6.1: Yksinkertainen VRML-määrittelytiedosto

```
#VRML V2.0 utf8
Transform{
  scale 0.75 0.75 0.75
  children [
    Transform {
      children [
```

```
        Shape {
          appearance Appearance {
            material Material {
              diffuseColor 1 0 1
            }
           surface DEF RBUTTON_1 ButtonSimpleSurface {
           }
          }
          geometry Sphere {
            radius 0.08
            depth 3
          }
        }
      ]
    }
  ]
}
DEF SCRIPT PythonScript {
  url "esim1.py"
}
```

Virtuaalimaailma ladataan osaksi jo olemassaolevaa näkymägraafia, jossa on valmiina jaettuja elementtejä, jotka mahdollistavat lähinnä kahdensuuntaisen liikenteen kontrollin ja Python-skriptikielen välillä. On siis mahdollista vaikuttaa virtuaalimaailman skriptattuun toiminnallisuuteen ja skripteistä voidaan lähettää esimerkiksi viestejä agenteille.

Kontrolli tarjoaa erikoistetun agenttisäiliön ja siinä olevien edustaja-agenttien avulla rajapinnan muille järjestelmän agenteille. Edustaja-agenttien avulla muut agentit voivat olla esimerkiksi vuorovaikutuksessa käyttäjän kanssa. Rajapinta mahdollistaa käyttäjän informoinnin jostain asiasta ja yksinkertaisen kyllä/ei-kysymyksen esittämisen käyttäjälle. Kummassakin tapauksessa PHANToM-laitteen kynää täristetään ja pieni merkkiääni soitetaan interaktioyrityksen merkiksi. Käyttäjä voi joko hyväksyä tai evätä tulevan viestin. Jos käyttäjä ei reagoi viestiin, tulkitaan se tietyn ajan kuluessa kieltäytymiseksi. PROAGENTS-projektin viesteissä käytetään vain äänipalautetta, eli viestit soitetaan käyttäjälle hänen niin halutessaan.

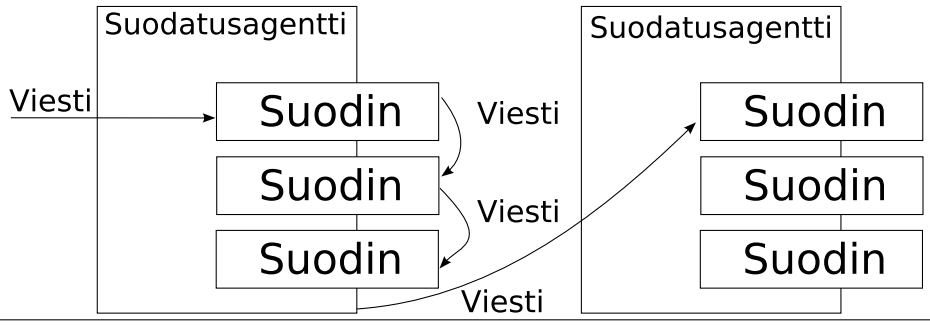
Navigointia helpottavaa äänipalautetta ei ole toteutettu projektissa, mutta

rajapinta mahdollistaa äänien suoran soiton eri äänikanavilla. Esimerkiksi opasteäänet ja huomioäänet soitetaan tämän rajapinnan kautta. Kontrollin rajapinnan avulla agentit voivat myös antaa haptista palautetta käyttäjälle. Projektissa toteutettiin vain kynän liikuttaminen tiettyyn koordinaattiin ja kynän täristäminen, mutta olisi myös mahdollista vaikuttaa virtuaalimaailmaan suoraan tai esimerkiksi toteuttaa dynaamista haptista palautetta.

6.3 PROAGENTS-agentit

Minimaailmat ovat vain osa PROAGENTS-sovellusta. Suurin osa sovelluksen toiminnallisuudesta sijaitsee agenteissa, jotka ovat yhteydessä kontrolliin viestikanavan kautta. Tärkeimmät PROAGENTS-sovelluksessa rakennetut agentit ovat suodatusagentit, tietokanta-agentti, sääntökone-agentti ja pedagoginen agentti.

Suodatusagentit ovat agentteja, jotka sisältävät yhden tai useamman suotimen. Suodin on toiminnaltaan hyvin rajattu ja pieni komponentti, joka reagoi vain tiettyihin asioihin. Suodin voi muuttaa saamaansa dataa ja lähettää sitä eteenpäin toisille (suodin)agenteille (kuva 6.5). Yhden agentin sisällä olevat suotimet voidaan ketjuttaa siten että kun yksi suodin reagoi tulevaan dataan, sitä ei enää anneta jäljellä oleville suotimille.



Kuva 6.5 Suodinagenttien toiminta

Suodinten tarkoitus on mahdollistaa tietovirta-arkkitehtuurin (pipes and filters) toiminnallisuus [Buschmann *et al.*, 1996]. Tietovirta-arkkitehtuuri koostuu suodinten ketjuista, joissa ne suodattavat tietoa ja lähettävät muutetun tiedon toisille suotimille. Suotimia voidaan esimerkiksi käyttää kerätyn tiedon muuttamiseen korkeamman tason operaatioiksi. Suodinten verkko voidaan luoda niin, että alimmalla, ensimmäisellä, tasolla on atomisia tapahtumia tarkkailevia agentteja. Ne lähettävät tiedon tapahtumista seuraavan tason suodatusagenteille, jotka koostavat alemman tason agenteilta saatavaa tietoa ja lähettävät sitä eteenpäin

seuraavalle tasolle. Tämän kaltaisella suodinverkolla voisi esimerkiksi yrittää selvittää sitä, mitä käyttäjä yrittää tehdä.

PROAGENTS-projektissa suotimia käytetään esimerkiksi järjestelmässä kulkevien lokitietojen tarkkailuun ja tietokanta-agentin tietojen päivittämiseen. Järjestelmässä on esimerkiksi suodin, joka käynnistää minimaailmakohtaisen taustamusiikin maailmasta toiseen siirryttäessä.

Tietokanta-agentti pitää yllä sovelluksen jaettua tietoa. Jaettua tietoa on mm. se, missä minimaailmoissa ja kuinka monta kertaa käyttäjä on käynyt. Tämän lisäksi tietokantaan on tallennettuna minimaailmakohtaista tietoa. Agentti on rakennettu SQLite tietokannan [SQLite, 2006] päälle. Tietokanta on pysyvä, joten sovelluksen tila välittyy seuraavalle käyttökerralle.

Agentit voivat hakea ja muuttaa tietokannan tietoalkioita helposti arkkitehtuurin tarjoamien apuluokkien avulla. Apuluokat käytännössä käärivät yksittäisen tietokannan tietoalkion SqlFact-luokan instanssiin. SqlFact-luokalla on metodit tietoalkion hakemiseen, muuttamiseen ja muutoksen lähettämiseen tietokanta-agentille. Agentti voi myös ilmoittaa olevansa kiinnostunut tietystä tietoalkiosta ja sille lähetetään välittömästi tieto sen muuttumisesta.

Sääntökone-agentti vastaa sovelluksen dynaamisesta toiminnallisuudesta ja lähes kaikesta interaktiosta käyttäjän kanssa. Sääntökone-agentti on tiukasti yhteydessä tietokanta-agenttiin, eli se on kirjautunut kuuntelemaan tietokanta-agentin kaikkia tietoalkioita.

Sääntökone-agentti sisältää CLIPS-sääntökoneen [CLIPS, 2006]. Sääntökoneessa on työmuisti, jossa kaikki sääntökoneen tietämys sijaitsee. Työmuistin lisäksi sääntökoneessa on joukko työmuistissa olevia tietoalkioita koskevia sääntöjä.

Sääntö koostuu ehto-osasta ja toimintaosasta. Sääntökonetta ajettaessa se koostaa agendan niistä säännöistä, joiden ehto-osa toteutuu. Tämän jälkeen sääntökone soveltaa ennalta määrättyä valintastrategiaa laukaistavan säännön valitsemiseksi. Valintastrategioita on useita, joista oletuksena käytetään strategiaa, joka valitsee säännön, jonka sääntöosan tyydyttämisen aikaansaavan faktan muutos on tuorein. Sääntöihin voidaan myös kiinnittää painoarvoja, joiden avulla voidaan korostaa joidenkin sääntöjen tärkeyttä laukaistavaa sääntöä valittaessa.

Säännöt kuvaavat lähes kaiken käyttäjälle näkyvän agenttitoiminnallisuuden, eli laukeavat säännöt voivat saada aikaiseksi yhteydenoton käyttäjään ja informaation tarjoamisen käyttäjälle. Esimerkissä 6.2 on yksinkertainen sääntökoneen sääntö, joka Suomen löydyttyä käynnistää Maa-minimaailmassa pienen pelin etelä- ja pohjoisnavan löytämiseksi. Suurin osa esimerkin tarkistettavista faktois-

ta on tietokannassa olevien tietoalkioiden sääntökoneen työmuistissa sijaitsevia vastineita.

Esimerkki 6.2: Yksinkertainen sääntökoneen sääntö

```
(defrule earth_finrule
        (mode run)
        (active_scene value Earth)
        (Earth found finland times ?x&:(> ?x 0))
        (Earth found north_pole times 0)
        (Earth found south_pole times 0)
        (Earth mode 1)
        (not (earth earth_navrule fired))
        ?evr <- (earth earth_modechange variable ?ev)
        (times_visited room Earth times ?tie)
        (test (>= ?tie ?ev))
=>
        (assert
                (earth earth_navrule fired)
                (nav-mode fin-first 0)
        )
)
```

Kaikki agenttien käyttäjälle lähettämät interaktioviestit kulkevat pedagogisen agentin kautta. Pedagoginen agentti vastaa interaktion mielekkyyden tarkistamisesta ja kontekstin säilymisestä. Projektissa käytetty pedagoginen agentti on pieni ja periaatteessa lähettää vain kaikki viestit edelleen kontrollille. Pedagogisella agentilla voisi olla esimerkiksi kontekstuaalista tietoa tutkinnan alla olevasta aiheesta ja vaikkapa erilaisia opetussuunnitelmia. Jos esimerkiksi käyttäjän katsotaan juuri oppineen asian A, niin pedagoginen agentti voisi tutkia, mitkä asiat olisi hyvä tietää ennen asian A oppimista ja tarjota niitä käyttäjän tutkittavaksi. Agentti voisi myös tarjota tutkittavaksi sellaisia asioita, jotka ovat riippuvaisia juuri opitusta asiasta. Näin olisi mahdollista säilyttää opittavien asioiden loogiset yhteydet ja konteksti.

7 Arkkitehtuurin arviointia

Kuten aiemmin todettiin, alusta rakennettiin PROAGENTS-projektin yhteydessä vastaamaan projektissa syntyneitä tarpeita. Alusta suunniteltiin yleisluontoiseksi ja sitä käytettiinkin myöhemmin sekä Tekes-rahoitteisessa MUKLA-projektissa että EU-rahoitteisessa MICOLE-projektissa. Tässä luvussa arvioin sekä alustan että PROAGENTS-projektissa rakennettujen komponettien toimivuutta.

7.1 Agenttialusta

Agenttialustan rakenne on hyvin avoin ja yleinen ja mahdollistaa uuden toiminnallisuuden lisäämisen alustaan. Arkkitehtuurin voima on siinä, että sen piiriin pystyy ulottamaan jo valmiina olevia komponentteja ja kirjastoja. Järjestelmään on myöhemmin lisätty onnistuneesti uusia komponentteja ja agentteja, kuten Prolog- ja SDL-agentit. Prolog-agentti käyttää SWI-Prolog-kirjastoa [SWI-Prolog, 2006] ja mahdollistaa Prolog-ohjelmien liittämisen agenttien toiminnallisuuteen. SDL-agentit vuorostaan käyttävät Simple Directmedia Layer -kirjastoa mm. äänen ja kuvan tulostamiseen. SDL [Simple Directmedia Layer, 2006] on avointa lähdekoodia oleva käyttöjärjestelmäriippumaton multimediakirjasto, joka lähinnä vastaa Windows:n DirectX-multimediakirjastoa. MUKLA-projektissa alustaan integroitiin myös erilaisia peliohjaimia, kuten ratti ja tärinällä varustettu pad-ohjain.

Toinen alustan vahvuus, helpon laajennettavuuden lisäksi, on standardinomainen tiedonsiirtokanava ja käyttöjärjestelmä- ja ohjelmointikieliriippumaton kommunikointiformaatti. Tiedonsiirtoon käytetään yleensä TCP/IP-protokollaa, vaikkakin muiden protokollien käyttö on mahdollista. Kommunikaatioon käytetään XML-koodattuja tekstipohjaisia viestejä. Sekä TCP/IP-pohjainen tietoliikennöinti että XML-dokumenttien käsittely on toteutettu lähes kaikissa nykyaikaisissa ohjelmointikielissä, joten alustoja ja agentteja voidaan rakentaa lähes millä kielellä tahansa. Käyttöjärjestelmä- ja ohjelmointikieliriippumaton kommunikointiformaatti mahdollistaa eri ohjelmointikielillä toteutettujen agenttien saumattoman keskustelun rajoittamatta viestin sisältöä. Viestin sisältönä saa siis olla mitä vain binääridatasta XML:ään, kunhan viestin *language*-kentästä selviää sisällön tyyppi. Käyttöjärjestelmä- ja ohjelmointikieliriippumattomasta kommunikointiformaatista seuraa, että agentit eivät voi olla liikkuvia, eli agentti ei voi siirtyä alustalta toiselle.

Alusta ei ominaisuuksiltaan ole vielä täysin valmis. Alustaan ei esimerkiksi ole

tällä hetkellä rakennettu minkäänlaisia käyttöoikeusluokituksia. Alustat ja agentit voivat siis kirjautua viestikanavaan ilman minkäänlaisia oikeuksien tarkastelua. Tämän lisäksi agentti voi, niin halutessaan, väärentää viestin toisen agentin nimiin. Eikä agentin ole mahdotonta urkkia toisen samassa säiliössä olevan agentin postilaatikon sisältöäkään. Tietoturvallisuutta ei todettu projektin puitteissa prioriteetiltaan tärkeäksi toteutettavaksi. Käyttöoikeuksien tarkistusten lisääminen viestikanavan kirjautumisprotokolliin ei ole monimutkainen operaatio.

Agenttialustan keskitetystä rakenteesta on sekä hyötyä että haittaa. Keskitetty rakenne helpottaa suuresti alustojen ja agenttien käynnistämistä ja hakupalvelut mahdollistavat joustavan ja helpon tavan agenttien etsimiseksi. Toisaalta keskitetty järjestelmä voi hyvin nopeasti muodostua järjestelmän pullonkaulaksi, varsinkin jos liikennettä on paljon ja suuri osa viesteistä on luonteeltaan sellaisia, että ne vaativat viestikanavan prosessointia. Alustan arkkitehtuurista löytyy yleisten komponenttien joukosta valmiit luokat sekä TCP- että UDP-yhteyksien luomiseen, joita agentit voivat vapaasti käyttää. Niiden käyttäminen kuitenkin lisää agentin monimutkaisuutta jonkin verran. Keskitetty rakenne ei muodostunut pullonkaulaksi niissä järjestelmissä, joissa agenttialustaa käytettiin.

Agentin rakenteen avoimuudesta on myös sekä hyötyä että haittaa. Agentin suoritus tapahtuu silmukassa, joka suorittaa samaa rutiinia yhä uudelleen. Jos agentin toiminnallisuutta ei osata purkaa osiin, muodostuu rutiinista suunnattoman iso ja erittäin vaikeasti hallittava. Yksinkertaisen agentin saa erittäin helposti luotua, mutta arkkitehtuurista puuttuu tuki rakentaa agentteja, jotka pystyvät käymään ja hallinnoimaan useita dialogeja samanaikaisesti.

Esimerkiksi JADE-agenttialusta jakoi agentin toiminnallisuuden käyttäytymismalleihin, joiden avulla agentin toiminnallisuus saatiin jaettua pienempiin loogisiin kokonaisuuksiin. Myös tämän alustan tapauksessa agenttien välisen vuorovaikutuksen kapseloiminen omiksi loogisiksi kokonaisuuksiksi olisi kannatettavaa. Jos agenttien välisiä dialogeja pidetään tarkkaan määritteltyinä protokollina, voisi ratkaisuna olla mukautettujen tilakoneiden käyttö. Jokainen protokolla olisi oma tilakone. Tilakoneeseen voisi liittyä useampia dialogeja toisten agenttien kanssa. Tilakoneen lisäksi tarvitaan tietorakenne pitämään yllä kunkin dialogin tunnistin ja tieto siitä, missä tilassa kyseinen dialogi on tällä hetkellä. Tarvitsee siis ylläpitää tietoa, siitä missä tilassa kyseinen dialogi toisen agentin kanssa on ja minkälaisella tunnisteella varustetut viestit kuuluvat juuri tähän kyseiseen dialogiin. Esimerkiksi tuleva viesti tulkitaan siinä tilassa, missä dialogi sillä hetkellä on. Viestin tulkinnasta saattaa seurata siirtymä seuraavaan tilakoneen tilaan.

Arkkitehtuurin muutos siihen, että agentteja voidaan luoda linkitetyiksi kir-

jastoiksi, on erittäin toimiva ja joustava ratkaisu. Se mahdollistaa agenttien kehittämisen täysin erillään itse agenttialustasta. Ratkaisu oli helppo mukauttaa agenttialustan tarpeisiin ja se toimii hyvin eri käyttöjärjestelmissä.

7.2 PROAGENTS-laajennukset

Agenttialustaa laajennettiin PROAGENTS-projektiin rakentamalla siihen erikoistettu agenttialusta, joka toimi yhdyskäytävänä itse opetusohjelman ja agenttien välillä. Tämän lisäksi rakennettiin itse opetusohjelma ja siihen liittyvät agentit. Agenttien välisiin dialogeihin suunniteltiin yksinkertaiset protokollat.

Opetusohjelmaa testattiin useaan otteeseen sekä näkevillä että sokeilla lapsilla. Järjestelmän testaus aloitettiin tammikuussa 2003 ja se jatkui läpi projektin [Saarinen *et al.*, 2006]. Suurin testi suoritettiin keväällä 2004, jolloin seitsemän 12-vuotiasta sokeaa ja osittain näkevää lasta osallistuivat viikon kestäviin testeihin näkövammaisten koululla. Lapsia kannustettiin arvioimaan opetusohjelman toiminnallisuutta ja näitä tuloksia käytettiin hyväksi opetusohjelman jatkokehityksessä. Viimeinen testi suoritettiin syksyllä 2004, jolloin kolme sokeaa lasta testasivat järjestelmän viimeisintä versiota. Lasten kommenttien perusteella pystyttiin rakentamaan ohjelma, joka suurimmilta osin oli onnistunut. Erityistä kiitosta sai aurinkokuntaa mallintava minimaailma. Kynän täristys viestin saapumisen merkiksi koettiin luontevaksi ja lapset oppivatkin käyttämään viestien kuuntelujärjestelmää hyvin nopeasti.

Tällä hetkellä käyttäjän reagoimista saapuvaan viestiin odotetaan tietyn aikaa kunnes viesti katsotaan hylätyksi. Agentit käyttävät samaa viestien rajapintaa sekä asioista informoimiseen että kysymiseen. Mekanismi ei osaa vielä katsoa kysymyksen ajallista pituutta, joten pitkissä kysymyksissä voi käydä niin, että viesti hylätään ennen kuin se on kokonaisuudessaan esitetty käyttäjälle.

Erikoistettu agenttialusta ja agentit toimivat odotetulla tavalla. Jotkut agenteista on hyvin laajoja, esimerkiksi sääntökoneagentti koostuu yli yhdeksästäkymmenestä säännöstä. Sääntökoneen lisääminen järjestelmään helpotti monimutkaisten asiayhteyksien luomista ja käsittelyä. Erityisesti SQL-agentti oli ominaisuuksiltaan ja toiminnaltaan hyvä. Muiden agenttien mahdollisuus tarkkailla tietojen muuttumista helppokäyttöisten SqlFact-olioiden avulla teki agenttien välisen kommunikaation ja tiedonvälityksen helpoksi.

Projektissa luotujen agenttien ehkä heikoin piirre oli se, että suurin osa käyttäjän suuntaan tapahtuvasta liikenteestä luotiin sääntökoneagentissa ja pedagoginen agentti jäi hyvin kevyeksi. Parempi olisi ollut, jos sääntökone, niin kuin

muutkin agentit, olisivat vain ehdottaneet jonkin tapahtuman tapahtumista pedagogiselle agentille, ja se olisi itse päätellyt miten ja koska tapahtuma tapahtuu.

Agenttien toiminnan kannalta olisi ollut hyvä, jos opetettavia asioita olisi saatu ryhmiteltyä ja niiden välisiä riippuvuussuhteita määriteltyä. Tällöin agenttien toimintaan olisi voitu lisätä ennustavaa ja neuvovaa toimintaa. Agenttien ehdotukset olisivat esimerkiksi voineet olla temaattisesti yhteensopivia. Opetusohjelmassa kuitenkin haluttiin tukea tutkivaa oppimista ja opetettavien asioiden välisiä hierarkioita ei haluttu luoda.

Pedagogiselle agentille olisi voinut rakentaa persoonallisuutta ja erilaisia opetus-agendoja, mutta kuten edellä mainittiin haluttiin järjestelmä pitää mahdollisimman vapaasti käyttäjän valinnan mukaan tutkittavana.

Kontrolli, joka sekä toteuttaa itse opetusohjelman että virtuaalimaailman ja tuntopalautelaitteen manipuloinnin rajapinnat, on erittäin laaja ja monimutkainen. Se rakennettiin siten, että järjestelmään on erittäin helppo lisätä uusia minimaailmoja ja niiden välisiä yhteyksiä. Kontrolli toimiikin erittäin hyvin projektin tarpeisiin nähden. Kontrollin tehokkuutta lisättiin rakentamalla siihen minimaailmojen välimuisti, joka pitää viittä viimeisintä minimaailmaa ladattuna muistissa. Näin nopeat edestakaiset siirtymiset eivät aiheuta viiveitä, jotka syntyvät kun minimaailma luetaan ja luodaan tiedostosta. Kontrolli kärsii rakenteen rapautumisesta. Se olisi tarvinnut asteittaista uudelleenjärjestämistä.

Opetusohjelmassa käytetään syötteenä tuntopalautelaitetta ja tulosteena tuntoaistia, kuvaa ja ääntä. Toisin kuin palautteiltaan, on sovellus syötemodaliteeltaan heikosti multimodaalinen. Toinen ohjelman heikkous on eri tulosteiden synkronoinnin puute. Sekä PROAGENTS- että MUKLA- projekteissa median toistamisen hoiti sama laite, joten samanaikaisesti toistettavat mediat toistettiin peräkkäin. Tämä järjestely ei ole hyvä erityisesti, jos työympäristö on jaettu useamman käyttäjän kesken. Seuraavassa kappaleessa esitänkin kuinka alustaa voidaan järjestää ja kehittää niin, että nämä ongelmat saadaan ratkaistua.

7.3 Lopuksi

Vaikka arkkitehtuurissa on osittain vielä parantamisen varaa, suoriutui se kaikista niistä tehtävistä, mihin se valjastettiinkin. Arkkitehtuurin avoimuus ja yleisluonteisuus mahdollisti sen käytön useammassa projektissa. Alustan vahvuus on mielestäni juuri käyttöjärjestelmäriippumattomuus, käyttöjärjestelmä- ja ohjelmointikieliriippumaton kommunikaatioformaatti ja alustan mukautettavuus eri tilanteisiin. Katson, että arkkitehtuuri ja projekti mihin se rakennettiin onnistui

hyvin ja toivon että arkkitehtuurin kehitystä jatkettaisiin edelleen. Tulokset on arvioitu myös tieteellisesti päteviksi arkkitehtuurin kuvaavassa ICMI-konferenssin paperissa [Saarinen *et al.*, 2005] ja PROAGENTS-hankkeen järjestelmästä kertovassa lehtiartikkelissa [Saarinen *et al.*, 2006], joiden pääkirjoittajana toimin.

8 Jatkotutkimusaiheita

Vaikka PROAGENTS-projektissa agenttialustan päälle rakennettu multimodaalinen sovellus täytti projektin vaatimukset ja toimi hyvin, jätti se monia multimodaalisten ohjelmien ongelmia ja ominaisuuksia ratkaisematta. Projektissa käytettiin vain yhtä syötelaitetta ja palaute oli tiukkaan määritelty ja sidottu tiettyihin laitteisiin. Multimodaalisissa sovelluksissa on kuitenkin tarkoitus mahdollistaa sekä syötteiden yhdistäminen ja tulkitseminen että palautteiden antaminen useampaa syöte- ja palautekanavaa apuna käyttäen.

Toinen ongelma, mihin projektissa ei tarvinnut perehtyä, oli tulosteiden synkronointi. Opetusohjelma on rakenteeltaan ja toiminnaltaan sellainen, että tulosteita ei tarvitse erityisemmin synkronoida. Tapahtumien samanaikaisuus saavutetaan laukaisemalla ne peräkkäin, sillä ohjelma käyttää samaan koneeseen liitettyjä multimedialaitteita. Palautteiden sidonta samaan laitteeseen ei kuitenkaan ole järkevää agenttialustaa käytettäessä, varsinkaan silloin kun työympäristö on jaettu useamman käyttäjän kesken. Tässä luvussa pohditaan, miten agenttialustaa voisi laajentaa siten, että medioiden fuusio ja median toiston hajauttaminen olisi mahdollista.

8.1 Fuusio PAC-Amodeus-mallin avulla

Kuten luvussa 3 esitettiin, PAC-Amodeus on malli interaktiivisten multimodaalisten sovellusten kehittämiseen. Malli mahdollistaa erityisesti medioiden fuusion eli yhdistämisen. Seuraavaksi esitetään muutamia ratkaisuja siihen, miten agenttialustan avulla voisi rakentaa PAC-Amodeusta vastaavan järjestelmän.

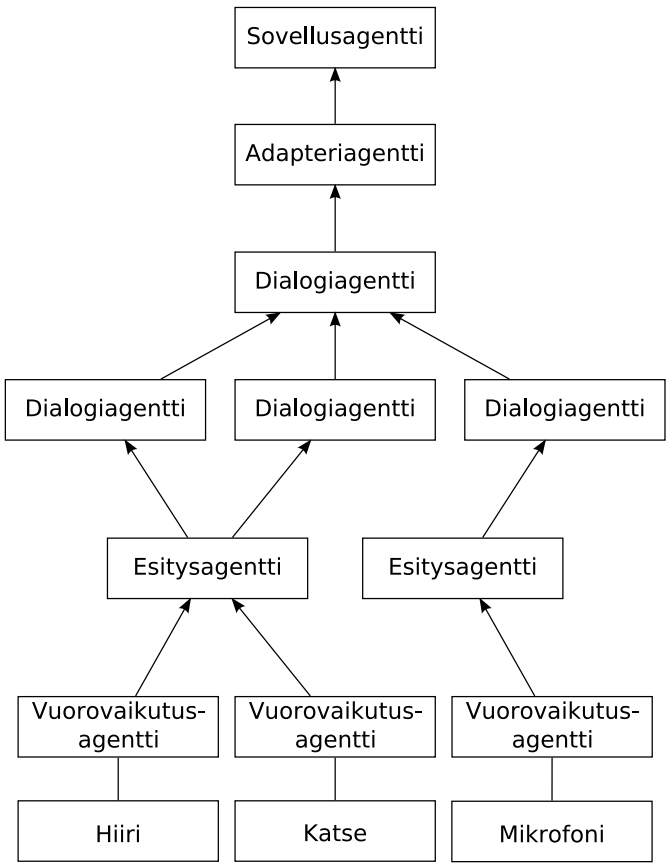
PAC-Amodeus koostuu Arch-mallista ja PAC-agenteista. PAC-Amodeus-mallissa sovellus jaetaan viiteen eri toiminnalliseen osaan: sovellusspesifi, sovellusmuunnin-, dialogi-, vuorovaikutus- ja esityskomponentti (kuva 3.6, s. 29). Data liikkuu komponentista toiseen ja sitä muokataan sovelluskohtaisesta muodosta esitysmuotoon ja toisin päin. Dialogikomponentissa tapahtuu syötteiden fuusio PAC-agenttien avulla.

Vuorovaikutuskomponentti voidaan nähdä erillisinä agentteina, jotka on liitetty tarkkailemaan tiettyjä syötelaitteita ja mahdollisesti myös käyttämään niitä palautteeseen. Jokainen vuorovaikutusagentti lähettää tiedon syötteestä esitysagentille tai -agenteille, jotka vastaavat esityskomponenttia PAC-Amodeus-mallissa.

Esitysagentti tietää siihen yhteydessä olevat vuorovaikutusagentit ja näiden

ominaisuudet. Tätä tietoa apuna käyttäen esitysagentti voi valita mieleisensä vuorovaikutusagentit palautteen antamiseen. Sen lisäksi, että esitysagentti jakaa palautteen vuorovaikutusagenteille, saa se syötettä vuorovaikutusagenteilta, jotka se yleistää ja lähettää ne edelleen dialogikomponentin toiminnallisuudesta vastaaville agenteille.

Dialogikomponentin toiminnallisuus voidaan toteuttaa agenteilla, jotka on järjestetty tasomaisesti. Alimman tason agentti kuuntelee esitysagenteilta tulevia sulatusuunien vastineita ja osaa lähettää käsittelemiään viestejä eteenpäin ylemmällä tasolla olevalle agentille. Näin aikaansaadut agenttien ketjut vastaavat PAC-hierarkioita. Ketjun viimeisin agentti toimittaa valmiin komennon adapteriagentille (sovellusmuunninkomponentille), joka muuntaa viestin sovelluskohtaiseksi komennoksi ja lähettää sen sovelluksen toiminnallisesta ytimestä vastaavalle agentille tai agenteille. Kuvassa 8.1 on kuvattu, kuinka viestit etenevät syötelaitteilta aina sovelluksen ytimeen saakka.



Kuva 8.1 Viestien eteneminen syötelaitteilta sovellukselle

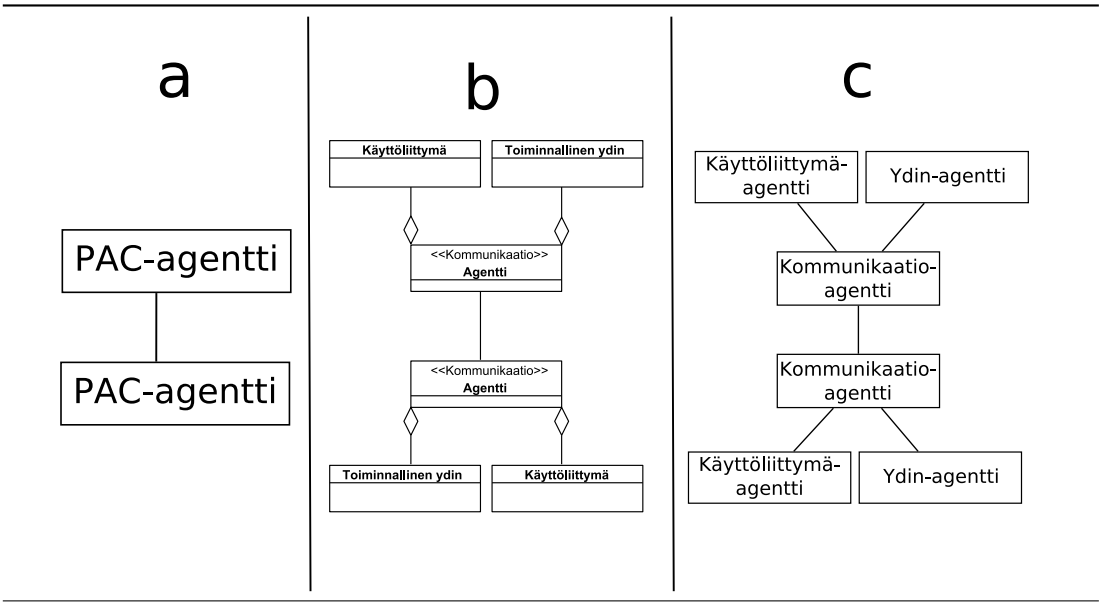
Järjestelmää voisi laajentaa lisäämällä siihen käyttäjäprofiiliagentin, joka pi-

täisi kirjaa esimerkiksi käyttäjän ominaisuuksista, taitotasosta ja mieltymyksistä. Profiili voisi olla hyvin hyödyllinen monen agentin toiminnassa. Esimerkiksi vuorovaikutusagentti voisi käyttää tätä tietoa hyväkseen valitessaan palautemodaliteetteja.

8.1.1 PAC-agentin toteutus

Yksittäisen PAC-agentin toteuttamiseen on myös useita erilaisia mahdollisuuksia:

- PAC-jako voidaan yhdistää samaan agentin toteutukseen (kuva 8.2 a).
- Agentti voi myös sisältää oman olion toiminnalliselle ytimelle ja käyttöliittymäosalle. Kommunikaatio-osaa vastaa periaatteessa agentin (säikeen) pääsuoritussilmukka, joka vastaanottaa ja lähettää viestejä ja koordinoi muiden osien toimintaa. Käyttöliittymäosaa ja toiminnallista ydintä vastaavat oliot voidaan myös tehdä säikeiksi, jolloin ne ovat jatkuvasti suorituksessa (kuva 8.2 b).
- Jokainen osa voidaan myös toteuttaa erillisinä agentteina, jolloin kommunikaatioagentti koordinoi käyttöliittymäagenttia ja ydinagenttia. Kolmikko olisi muihin vastaaviin agentteihin yhteydessä kommunikaatioagentin kautta (kuva 8.2 c).



Kuva 8.2 Erilaisia tapoja rakentaa PAC-agentti

Ensimmäisen toteutusvaihtoehdon ongelma on se, että agentista tulee monoliittinen ja monimutkainen. Tämän lisäksi agentti on vain osittain toiminnallinen toiminnan tapahtuessa samassa säikeessä, josta saattaa seurata, että kommunikaatio käyttöliittymäosan ja toiminnallisen ytimen välillä saattaa olla vajavaista. Toinen vaihtoehto on huomattavasti toteutuskelposempi kuin ensimmäinen, eritoten jos käyttöliittymäosa ja toiminnallinen ydin ovat myös säikeitä. Kolmas vaihtoehto on työläs saada toimivaksi, sillä agenttien välille pitää laatia tehokas kommunikointiprotokolla. Kun protokolla on valmis, voidaan agentteja vaihtaa suorituksenkin aikana. Kolmas vaihtoehto soveltuukin hyvin erittäin laajoihin ja monimutkaisiin sovelluksiin.

8.2 Synkronointi agenttialustassa

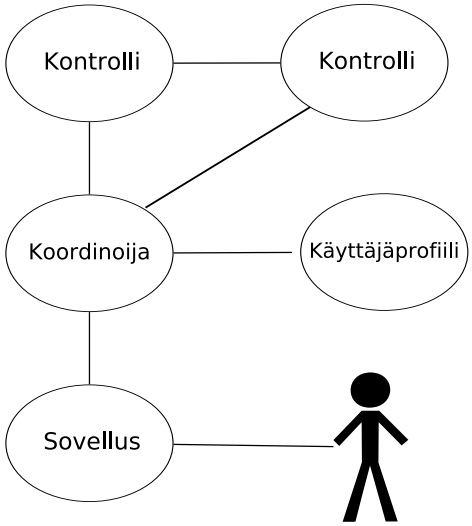
Kuten aiemmin todettiin, synkronointi ongelmana on normaalia vaativampi agenttiarkkitehtuureissa. Enää ei voida tehdä oletuksia agenttien sijainnista, sillä ne saattavat olla liikkuvia. Myös eri mediantoistolaitteet voivat olla hajallaan ja niitä ei yhdistä mikään muu, kuin yhteys lähiverkkoon. Näiden ongelmien ratkaisemiseen rakentaisin synkronointijärjestelmän keskitetyn koordinoija-agentin varaan. Koordinoija-agentin tehtävänä on tietää käyttäjän sijainti, käyttäjää lähellä olevien medialaitteiden kontrolliagentit, käyttäjän mahdolliset mieltymykset medialaitteiden käytöstä ja kontrolliagenttien tämänhetkinen kuormitus.

Tässä osiossa esittelemäni agenttijärjestelmään mukautettu ratkaisu on yksinkertainen pistesynkronointimalli. Ratkaisumalli on yksinkertainen, eikä ota kantaa moniin tosielämän ongelmatilanteisiin. Ratkaisussa oletetaan että tarvittavat mediaelementit ovat jo median toistavan agentin ulottuvilla, joten dataa ei lähetetä lähiverkon yli. Datan lähettäminen lähiverkon yli loisi uusia varteenotettavia ongelmia, kuten datan puskurointi ja synkronointi toiston aikana. Pohdin myöhemmin tämän ratkaisun laajennettavuutta vastaamaan myös datan lähettämisen tuomiin ongelmiin. Asian kattava tutkiminen tosin olisi itsessään erillisen tutkimuksen arvoinen ja täten se ei mahdu tähän tutkielmaan.

8.2.1 Matalan tason synkronointi

Koordinoija päättää aluksi, mitä laitteita käytetään minkäkin mediatyypin toistoon. Valinta tapahtuu käyttäjän mieltymysten ja käyttäjän sijainnin mukaan. Pistesynkronoinnissa koordinoija laskee valitsemiensa kontrolliagenttien ja koordinoijan välisen verkkoliikenteen viiveet ja muodostaa normalisoidun kellonajan

kullekin kontrolliagentille. Viive lasketaan käyttäen tyypillisintä mallia, eli koordinoija lähettää kontrolliagentille aikaleimalla varustetun viestin ja kontrolliagentti palauttaa samaisen viestin takaisin koordinoijalle liittäen samalla siihen oman aikaleimansa. Kun viesti saapuu takaisin koordinoijalle, voi se laskea viestin edestakaiseen kulkuun kuluneen ajan. Aikaleimoja vertailemalla koordinoija saa selville myös mahdolliset eroavaisuudet tietokoneiden kelloissa. Mittaus toistetaan useamman kerran, jotta viiveestä saadaan laskettua sopiva keskiarvo. Kellojen poikkeavuus tallennetaan koordinoijan ylläpitämään kontrolliagenttikohtaiseen synkronointiprofiiliin.



Kuva 8.3 Synkronointiagentit

Synkronoinnissa on kyse pohjimmiltaan siitä, milloin median toisto pitää aloittaa. Kun viiveet ja kelloon liittyvät erot on selvitetty, voidaan laskea kullekin mediavirralle sopivat ajankohdat toiston aloittamiseksi. Voidaan esimerkiksi laskea tämänhetkiseen kellonaikaan suurin verkkoviive. Summaan voidaan lisätä vielä mediaelementin käsittelyyn kuluvan ajan arvio, viestin käsittelyn kesto itsessään sisältyy jo verkkoviiveeseen. Tämä aika muutetaan vastaanottajan kellonajan mukaiseksi käyttäen kontrolliagentin profiiliin tallennettua kellojen eroa. Pyyntö median toiston aloittamisesta sovittuna ajankohtana lähetetään lopuksi kontrolliagentille.

Mallia voidaan vielä laajentaa siten, että kontrolliagentit välittävät tietoa median toiston etenemisestä toisille samassa synkronointitehtävässä oleville kontrolliagenteille. Tällöin muut agentit voivat säätää omaa toimintaansa muiden agenttien toiminnan mukaan. Mallin laajennus vaatii synkronointitehtävään osallistu-

vien agenttien listaamisen synkronointitapahtumassa.

Mikäli kyseessä ei ole pistesynkronointi, vaan mediaa suoratoistetaan verkon yli, on tehokkampaa sallia lähettäjän ja kontrolliagentin keskustella suoraan keskenään siten, että koordinaattori antaa kontrolliagentille lähettäjän tiedot ja kontrolliagentti laskee kellojen eroavaisuuden ja verkon viiveen kontrolliagentin ja median lähettäjän välillä. Kontrolliagentin tulee ilmoittaa verkkoviive ja kellojen eroavaisuus myös koordinaattorille, sillä koordinaattorin vastuulle jää vielä synkronian aikaansaaminen muiden medioiden kanssa. Dataa lähetettäessä ja vastaanottaessa tulee kontrolliagenttien välinen kommunikaatio tärkeäksi. On nimittäin todennäköistä että mediavirrat menevät epäsynkroniaan esimerkiksi verkkoviiveen takia, jolloin joudutaan sovittelemaan tilannetta jollakin tavalla. Mikäli mediaelementit ovat auttamatta myöhässä, yleensä joko toistetaan viimeistä mediaelementtiä (kuva) tai ei toisteta mitään (ääni). Jos taas mediaelementtien puskuri täyttyy tai muiden säikeiden kulku pakottaa siirtymään toistossa eteenpäin, jätetään yleensä mediaelementtejä toistamatta.

8.2.2 Synkronointiprotokolla

Matalan tason synkronoinnin lisäksi tarvitaan agenteille formaali tapa sopia synkronisaatiosta. Synkronointiprosessi voidaan eriyttää muista agenteista erillisen media-agentin kautta, joka vastaanottaa tapahtumapyyntöjä agenteilta, muuttaa pyynnöt synkronointiprotokollaksi ja aloittaa neuvottelun koordinaattorin kanssa. Tällöin muiden agenttien ei tarvitse huolehtia yksityiskohdista, vaan ne voivat helposti tehdä yksinkertaisia pyyntöjä.

Kuten agenttien väliseen kommunikointiin, on medioiden synkronointiin suunniteltu useita standardeja. Ylläpidon ja laajennettavuuden vuoksi on järkevää valita yksi niistä synkronointiprotokollan viitteelliseksi malliksi. Synkronointistandardien vertailun jälkeen päätin ottaa tämän esimerkin pohjaksi W3C:n SMIL 2.1 -standardista (Synchronized Multimedia Integration Language) ajoitus- ja synkronointimodulin [SMIL 2.1, 2006b]. Moduli kuvaa korkealla tasolla erilaisten multimediaelementtien synkronointiprotokollan XML-kielellä. Tässä käsittelemme standardista ne osat, joita arkkitehtuuri tukisi. Koko synkronointimoduli on hyvin laaja ja täten sen kuvaaminen tässä on mahdotonta.

SMIL 2.1 määrittelee kolme synkronointielementtiä. Ensinnäkin synkronointielementti <seq> määrää että kaikki elementin lapsiksi liitetyt mediaelementit toistetaan peräkkäin siinä järjestyksessä, missä ne on listattu elementin lapsiksi. Elementti <excl> määrää, että elementin lapset toistetaan peräkkäin, mutta ei

määrää elementtien järjestystä. Kolmantena elementti <par> määrää, että elementin lapset toistetaan samanaikaisesti. Synkronointielementit voivat sisältää toisia synkronointielementtejä. Taulukossa 4.1 esitetyt synkronointitapahtumat on mahdollisia synkronointielementtejä <seq> ja <par> käyttäen.

SMIL-spesifikaation mediamoduulissa määritellään synkronoitavat mediaelementit [SMIL 2.1, 2006a], joista protokolla tukisi seuraavia:

- <animation>, animaatio.
- <audio>, ääni.
- <img>, kuva.
- <text>, teksti.
- <video>, video.

Jos esimerkiksi halutaan näyttää kuva ja soittaa musiikkia samanaikaisesti, muodostetaan seuraava SMIL-skripti (esimerkki 8.1). Skriptistä on jätetty pois osa SMIL-määritelmästä, jotta esimerkki pysyy yksinkertaisena.

Esimerkki 8.1: Yksinkertainen synkronointitapahtuma

```
<par>
 <audio id="song1" src="song1.au" />
 <img src="img1.jpg" />
</par>
```

Jokaiseen media- ja synkronointielementtiin voidaan liittää attribuutteja, joilla voidaan kontrolloida mediaelementin toimintaa tai ajoitusta. Mediaelementillä on yksinkertainen kesto, mikä määrittää elementin suorituksen ajan. Elementille voi myös määritellä toistoattribuutin, jonka avulla voidaan toistaa kyseistä elementtiä haluttu määrä. Yksinkertainen kesto ja mahdolliset toistot huomioon ottaen saadaan elementille laskettua aktiivinen aika, mikä määrittää kyseisen elementin elinkaaren synkronointitehtävässä.

Elementille voidaan määritellä toiston aloitushetki. Protokollamme tukisi kolmenlaista aloitushetken määrittelyä. Ensinnäkin voidaan määritellä kellonaika, milloin suoritus aloitetaan. Toiseksi voidaan antaa positiivinen ajan määre, jolloin suoritus aloitetaan: esimerkiksi kolmen minuutin kuluttua. Ja viimeiseksi voidaan määritellä aloitusajaksi toisen elementin aloitus- tai lopetusaika, jolloin

suoritus aloitetaan toisen elementin alkaessa tai loppuessa. Aloitus- ja lopetusaikaan voidaan myös liittää ajanmääre, jolloin voidaan esimerkiksi määritellä aloitushetkeksi 50 sekuntia ennen toisen elementin loppua.

Jos halutaan että median toisto alkaa toisen median toiston aloituksen jälkeen, voidaan se määritellä esimerkissä 8.2 olevalla tavalla. Siinä kuva näytetään kaksi sekuntia musiikin soiton alun jälkeen.

Esimerkki 8.2: Kuva alkaa kaksi sekuntia laulun jälkeen

```
<par>
  <audio id="song1" src="song1.au" />
  <img src="img1.jpg" begin="song1.begin+2s" />
</par>
```

Aloitushetken lisäksi elementille voidaan määritellä lopetushetki. Kuten aloitusajassakin, protokollamme tukisi kolmenlaista lopetushetken määrittelyä. Ensinnäkin voidaan määritellä kellonaika, milloin suoritus lopetetaan. Voidaan myös antaa positiivinen ajan määre, jolloin elementin suoritus lopetetaan: esimerkiksi kolmen minuutin kuluttua. Ja viimeiseksi voidaan määritellä lopetusajankohdaksi toisen elementin aloitus- tai lopetusaika, jolloin suoritus lopetetaan toisen elementin alkaessa tai loppuessa. Aloitus- ja lopetusaikaan voidaan edelleen liittää ajanmääre, jolloin voidaan esimerkiksi määritellä lopetushetkeksi 12 sekuntia ennen toisen elementin alkua.

Mediaelementtiä voidaan toistaa kahdella tavalla, joista protokollamme tukisi molempia. *RepeatCount* toistaa elementtiä n kertaa tai loputtomasti, mikäli sille on annettu arvoksi "indefinite". *RepeatDur* toistaa elementtiä tietyn ajan tai loputtomasti, mikäli sille on määritelty kestoksi "indefinite".

Toistoattribuutin, yksinkertaisen keston, aloitushetken ja lopetushetken voi myös määritellä synkronointielementille. Mikäli yksinkertaista kestoa ei ole määritelty, määrittelee järjestelmä sen. Esimerkiksi synkronointielementin yksinkertainen kesto on sen lapsielementtien kestojen summa. Tarkempi attribuuttien määrittely löytyy SMIL 2.1 -spesifikaatiosta [SMIL 2.1, 2006b].

Esimerkissä 8.3 näytetään kolme kuvaa peräkkäin siten, että ensimmäinen kuva näkyy viisi sekuntia ja muut kuvat neljä sekuntia. Tämä kolmen kuvan sarja toistetaan kolme kertaa. Synkronointitapahtuman kokonaispituus, eli aktiivinen aika, on 39 sekuntia.

Esimerkki 8.3: Kolmesti toistettava kuvasarja

```
<seq begin="5s" repeatCount="3" >
```

```
    <img src="img1.jpg" dur="5s" />
    <img src="img2.jpg" dur="4s" />
    <img src="img3.jpg" dur="4s" />
</seq>
```

Kun elementin aktiivinen aika päättyy synkronointielementin keston jatkuessa, pitää ratkaista mitä jo päättyneelle elemetille tehdään. Protokollaan voidaan määritellä attribuutti, millä elementin päättymisen käyttäytymisen voi määritellä. Määrittely tapahtuu attribuutilla *fill*, joka mahdollisista arvoista protokolla tukisi *remove* ja *freeze* määrittelyjä. *Remove* määrittää, että elementin toistoa ei jatketa päättymisajan jälkeen ja *freeze* vuorostaan määrittää, että elementin lopputilaa toistetaan päättymisajan jälkeen.

Seuraavassa esimerkissä (esimerkki 8.4) videon toisto aletaan viisi sekuntia ja äänen toisto kaksi sekuntia synkronointitapahtuman alusta. Videolle on myös määritelty tyhjän tilan täyttötoiminta, eli jos video päättyy ennen synkronointitapahtuman päättymistä, loppuajan näytetään videon viimeistä ruutua.

Esimerkki 8.4: Viiveellä alkavat medioiden toistot, niiden pituus ja loppuajan täyttö

```
<par endsync="last">
  <video src="vid.mpg" begin="5s" dur="30s" fill="freeze"/>
  <audio src="intro.au"  begin= "2s" dur="40s"/>
</par>
```

Lopuksi esimerkkinä ote monimutkaisemmasta synkronointiskriptistä (esimerkki 8.5), jossa näytetään peräkkäin rinnakkain tapahtuvia kuvan ja äänen toistoja. Eli aluksi näytetään image1 ja samanaikaisesti soitetaan audio1. Tämän jälkeen tehdään sama image2:lle ja audio2:lle jne.

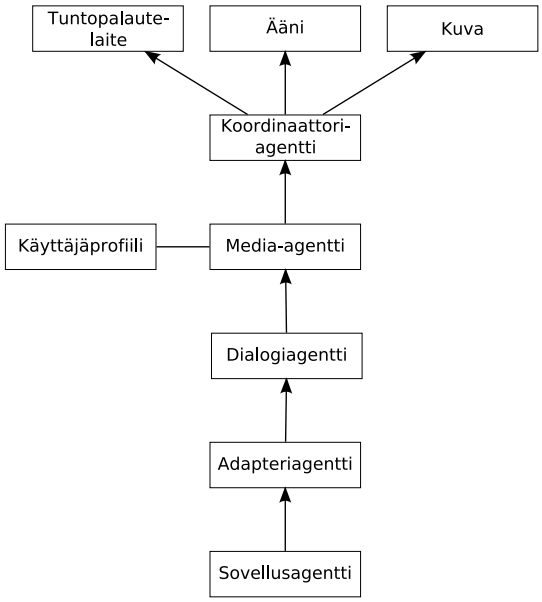
Esimerkki 8.5: Monimutkaisempi synkronointiskripti

```
<par>
  <seq>
    <par>
      <img src="image1.jpg" fill="freeze"/>
      <audio src="audio1.au"/>
    </par>
    ...
    <par>
```

```
      <img src="image2.jpg" fill="freeze"/>
      <audio src="audio2.au"/>
    </par>
    ...
    <par>
      <img src="imageN.jpg" fill="freeze"/>
      <audio src="audioN.au"/>
    </par>
  </seq>
</par>
```

8.2.3 Fuusion ja synkronoinnin yhdistäminen

Edellä kuvaillussa PAC-Amodeus-mallissa on hyvin joustava rakenne ja synkronoinnin saa lisättyä siihen varsin helposti. Synkronointiagentit tulevat vuorovaikutuskomponenttien yhteyteen siten, että jokainen mediaa toistava laite saa oman kontrolliagentin. Jos laite osallistuu sekä syötteen että palautteen antamiseen, voidaan kontrolliagentti integroida vuorovaikutusagenttiin. Kuvassa 8.4 järjestelmään on lisätty myös tarvittava koordinaattoriagentti, media-agentti ja käyttäjäprofiili. Media-agentti korvaa esitysagentin palautteiden antamiseen.



Kuva 8.4 PAC-Amodeukseen pohjaava agenttijärjestelmä, johon on lisätty synkronointiagentit

Media-agentti saa dialogiagenteilta palautepyyntöjä ja muodostaa niistä synkronointitapahtumia. Palautepyyntö on korkean tason kuvaus halutusta palautteesta. Jos se ei ota kantaa palautteeseen liitettäviin modaliteetteihin, saadaan yhdistettyä järjestelmään myös modaliteettien fissio, sillä Media-agentille jää silloin vastuu tapahtuman esittämistä sopivien modaliteettien avulla. Media-agentti antaa synkronointitapahtuman edelleen koordinaattoriagentille, joka toimii edellä mainitun mukaisesti. Media-agentti voi olla yhteydessä käyttäjän profiiliin, jolloin käyttäjälle voidaan antaa parempaa palautetta. Media-agentti voi esimerkiksi profiilin mukaan päättää mitä modaliteetteja palautteen antamiseen käytetään.

Agenttialustan joustava rakenne mahdollistaa näiden toiminnallisuuksien lisäämisen ilman suurempia muutoksia itse agenttialustan rakenteeseen, sillä suurin osa tarvittavasta uudesta toiminnallisuudesta on agenttien muodossa. Sen jälkeen, kun edellä mainitut muutokset on tehty, on alusta ominaisuuksiltaan monipuolinen ja sitä on helppo käyttää myös laajempien ja vaativien multimodaalisten sovellusten tekemiseen.

9 Yhteenveto

Agenttiteknologiat olivat kiivaan tutkimuksen alaisena 1990-luvulla, mutta myöhemmin kiinnostus niitä kohtaan on laantunut hieman. Tuleva kodin elektroniikka ja jatkuva laitteiden ja koneiden verkottuminen kuitenkin saattaa herättää kiinnostuksen agenttiteknologioita kohtaa uudelleen, sillä ne sopivat erittäin hyvin nykyajan ongelmien ratkaisujen pohjaksi. Agenttiteknologioille hajautettavuus ja kommunikoivien osapuolten heterogeenisuus on luontevaa.

PROAGENTS-projektissa valittiin agenttipohjainen lähestymistapa kahdesta syystä. Ensinnäkin toiminta haluttiin hajauttaa useammalle koneelle mm. tuntopalautelaitteen kovien resurssivaatimusten takia. Haluttiin myös toteuttaa valmis pohja, jonka varaan voidaan rakentaa tulevissa projekteissa.

Vuosien varrella on rakennettu useita erilaisia agenttialustoja, joista suurin osa on Javalla toteutettuja. Projekti vuorostaan on suunniteltu tuntopalautelaitteen varaan, jonka ohjelmointirajapinta on toteutettu C++:lla. Projektin puitteisiin sopivaa alustaa ei ollut saatavilla, joten lopulta rakennettiin oma C++:lla toteutettu agenttialusta.

Tässä tutkielmassa kuvattiin sekä agenttialusta että sen päälle rakennettu multimodaalinen opetusohjelma. Esitelty agenttialusta on rakenteeltaan kevyt ja helposti käyttöönotettava. Esimerkiksi alustan C++-toteutuksessa on helposti pystytty hyödyntämään muita valmiina olevia kirjastoja. Yleisluontoinen kommunikaatioformaatti mahdollistaa heterogeenisten agenttiympäristöjen välisen saumattoman keskustelun. Alustassa on vielä monia heikkouksia, mutta se soveltui käyttökohteisiinsa hyvin.

Itse opetusohjelma on ensimmäinen laajempi ohjelmisto, joka on rakennettu Tampereen Yliopistossa PHANToM-tuntopalautelaitetta käyttäen. Sovellus yhdistää useita erilaisia tekniikoita yrittäen samalla pysyä mahdollisimman joustavana ja helposti laajennettavana. Sovellukseen voidaankin lisätä uutta toiminnallisuutta helposti lisäämällä uusia virtuaalimaailmoja. Sovellusta kehitettäessä jouduttiin ratkaisemaan monia eteen tulleita yllättäviäkin ongelmia, kuten rikkinäisen äänentoistojärjestelmän korvaaminen omalla ratkaisulla. Sovellus toimi odotetulla tavalla ja tarjosi hyvät mahdollisuudet erilaisten ratkaisujen kokeilemiseen. Ajallisten rajoitusten takia sovelluksen rakenne rapautui hieman ja siitä tuli osittain monimutkainen.

Projektin sovellus käyttää useampaa palautekanavaa hyväksi, mutta multimodaalisen syötteen vahvuuksia ei sovelluksessa hyödynnetä. Esimerkiksi fuusiota ei esiinny ollenkaan, koska sovelluksessa käytettin vain yhtä syötelaitetta. Toinen

sekä agenttialustan että opetusohjelman puute oli medioiden synkronointitapojen puuttuminen. Esitin tässä tutkielmassa yhden mahdollisen ratkaisutavan sekä fuusion että medioiden toiston synkronoinnin lisäämiseksi agenttialustaan. Esitetty ratkaisutapa vahvistaisi alustan rakennetta ja antaisi hyvän rakennuspohjan monenlaisille multimodaalisille sovelluksille. Tarkoitukseni on jatkaa tutkimustyötä luvussa 8 esittämäni suunnitelman mukaisesti.

VIITELUETTELO

[Agarwal & Son, 1996] Nipun Agarwal & Sang H. Son. A model for specification and synchronization of data for distributed multimedia applications. *Multimedia Tools and Applications*, 3(2):79–104, 1996.

[Bass *et al.*, 1992] Len Bass, Ross Faneuf, Reed Little, Niels Mayer, Scott Reed, Robert Seacord, & Martha R. Szczur. A metamodel for the runtime architecture of an interactive system. Technical report, 1992.

[Bellifemine *et al.*, 1999] Fabio Bellifemine, Agostino Poggi, & Giovanni Rimassa. JADE - a FIPA-compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents*, 1999.

[Bellifemine *et al.*, 2001] Fabio Bellifemine, Agostino Poggi, & Giovanni Rimassa. Jade: a fipa2000 compliant agent development environment. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 216–217, New York, NY, USA, 2001. ACM Press.

[Biersack *et al.*, 1996] Ernst Biersack, Christoph Bernhardt, & Werner Geyer. Intra- and inter-stream synchronization of stored multimedia streams. In *International Conference on Multimedia Computing and Systems*, pages 372–381, 1996.

[Bigus *et al.*, 2002] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, & Y. Diao. Able: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3), 2002.

[Bradshaw, 1997] Jeffrey M. Bradshaw. *An introduction to software agents.* AAAI Press / MIT Press, 1997.

[Buschmann *et al.*, 1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, & Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

[Buxton, 1986] W. Buxton. There's more to interaction than meets the eye: Some issues in manual input. In D. A. Norman & S. W. Draper, editors, *User Centered System Design: New Perspectives on Human-Computer Interaction*, pages 319–337. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

[Calvary *et al.*, 1997] Gaëlle Calvary, Joëlle Coutaz, & Laurence Nigay. From single-user architectural design to pac*: A generic software architecture model for cscw. In *Proceedings of the SIGCHI Conference on HumanFactors in Computing Systems*, pages 242–249. ACM Press, 1997.

[CLIPS, 2006] CLIPS. A tool for building expert systems, 2006. http://www.ghg.net/clips/CLIPS.html.

[COMMON C++ library, 2006] Gnu common c++, a portable and highly optimized class framework for writing c++ applications, 2006. http://www.gnu.org/software/commoncpp/.

[Coutaz, 1987] Joëlle Coutaz. Pac: An object oriented model for implementing user interfaces. *SIGCHI Bull.*, 19(2):37–41, 1987.

[Falck, 2004] Kenneth Falck. Viihteen ytimenä mediakeskus, Joulukuu 2004. Tietokone.

[Finin *et al.*, 1992] Tim Finin, Richard Fritzson, & Don McKay. A language and protocol to support intelligent agent interoperability. In *Proceedings of the CE and CALS Washington 92 Conference*. ACM Press, 1992.

[Finin *et al.*, 1994] Tim Finin, Richard Fritzson, Don McKay, & Robin McEntire. Kqml as an agent communication language. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, pages 456–463, New York, NY, USA, 1994. ACM Press.

[FIPA, 2006a] The foundation of intelligent physical agents (fipa) 97 part 1 version 2.0: Agent management specification, 2006. http://www.fipa.org/specs/fipa00019/.

[FIPA, 2006b] The foundation of intelligent physical agents (fipa) 97 part 2 version 2.0: Agent communication language, 2006. http://www.fipa.org/specs/fipa00003/.

[FIPA, 2006c] The foundation of intelligent physical agents (fipa) 97 part 3 version 1.0: Agent software integration specification, 2006. http://www.fipa.org/specs/fipa00012/OC00012A.html.

[FIPA, 2006d] The foundation of intelligent physical agents (fipa) abstract architecture specifications, 2006. http://www.fipa.org/specs/fipa00001/SC00001L.html.

[Java Reflection API, 2006] Java reflection api, 2006. http://java.sun.com/docs/books/tutorial/reflect/index.html.

[Java RMI, 2006] Java remote method invocation, 2006. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

[Lamont *et al.*, 1996] Louise Lamont, Lian Li, Renaud Brimont, & Nicolas D. Georganas. Synchronization of multimedia data for a multimedia news-on-demand application. *IEEE Journal of Selected Areas in Communications*, 14(1):264–278, 1996.

[Manvi & Venkataram, 2005] S.S. Manvi & P. Venkataram. An agent based synchronization scheme for multimedia applications. *Journal of Systems and Software*, 79(5):701–713, 2005.

[Martin *et al.*, 1998] David L. Martin, Adam J. Cheyer, & Douglas B. Moran. Building distributed software systems with the open agent architecture. In *Proc. of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, Blackpool, Lancashire, UK, Mar 1998. The Practical Application Company Ltd. OAA.

[Martin *et al.*, 1999] D. Martin, A. Cheyer, & D. Moran. The open agent architecture: a framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1/2):91–128, 1999.

[Meyer, 2006] Mark Meyer. The features and facets of the agent building and learning environment (able), 2006. http://www-128.ibm.com/developerworks/autonomic/library/ac-able1/.

[Musgrove, 2004] Arthur J. Musgrove. Dynamic plug-ins for c++: Building flexibility into your programs, June 2004. http://www.ddj.com/184401819.

[Nigay & Coutaz, 1991] Laurence Nigay & Joëlle Coutaz. Building user interfaces: Organizing software agents. In *In Proceedings of Esprit'91*, pages 707–719. ACM Press, 1991.

[Nigay & Coutaz, 1993] Laurence Nigay & Joëlle Coutaz. A design space for multimodal systems: Concurrent processing and data fusion. In *In Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 172–178. ACM Press, 1993.

[Nigay & Coutaz, 1995] Laurence Nigay & Joëlle Coutaz. A generic platform for adressing the multimodal challenge. In *In Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 98–105. ACM Press/Addison-Wesley Publishing Co, 1995.

[OMG CORBA, 2006] Common object request broker: Core specification., 2006. http://www.omg.org/cgi-bin/apps/doc?formal/02-12-02.pdf.

[Patil *et al.*, 1992] Ramesh Patil, Richard F. Fikes, Peter F. Patel-Schneider, Don McKay, Tim Finin, Thomas Gruber, & Robert Neches. The DARPA knowledge sharing effort: Progress report. In Bernhard Nebel, Charles Rich, & William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 777–788. Morgan Kaufmann, San Mateo, California, 1992.

[Patomäki *et al.*, 2004] Saija Patomäki, Roope Raisamo, Jouni Salo, Virpi Pasto, & Arto Hippula. Experiences on haptic interfaces for visually impaired young children. In *Proceedings of the 6th International Conference on Multimodal Interfaces (ICMI'04)*, pages 281–288. ACM Press, 2004.

[Rao & Georgeff, 1995] A. S. Rao & M. P. Georgeff. Bdi-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[Reachin API, 2006] Reachin api, a development platform for haptic 3d applications, 2006. http://www.reachin.se/products/reachinapi/.

[Saarinen *et al.*, 2005] Rami Saarinen, Janne Järvi, Roope Raisamo, & Jouni Salo. Agent-based architecture for implementing multimodal learning environments for visually impaired children. In *Proceedings of the 7th International Conference on Multimodal Interfaces (ICMI'05)*, pages 309–316. ACM Press, 2005.

[Saarinen *et al.*, 2006] Rami Saarinen, Janne Järvi, Roope Raisamo, Eva Tuominen, Marjatta Kangassalo, Kari Peltola, & Jouni Salo. Supporting visually impaired children with software agents in a multimodal learning environment. *Virtual Reality*, 9(2-3):Springer–Verlag, London Ltd. 108–117, 2006.

[Schomaker *et al.*, 1995] L. Schomaker, J. Nijtmans, A. Camurri, P. Morasso, C. Benoit, T. Guiard-Marigny, B. Le Gof, J. Robert-Ribes, A. Adjoudani, I. Defee, S. Munch, K. Hartung, & J. Blauert. A taxonomy of multimodal interaction in the human information processing system: Report of the esprit project 8579 miami. Technical report, Nijmegen University, NICI, 1995.

[SensAble Technologies Inc, 2006] Phantom desktop haptic device, 2006. http://www.sensable.com/products/phantom_ghost/phantom-desktop.asp.

[Shoham, 1993] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[Shoham, 1997] Yoav Shoham. An overview of agent-oriented programming. In Jeffrey M. Bradshaw, editor, *Software Agents*, pages 271–290. AAAI Press / MIT Press, 1997.

[Simple Directmedia Layer, 2006] A cross-platform multimedia library, 2006. http://www.libsdl.org/index.php.

[Sjöström, 2001] Calle Sjöström. Designing haptic computer interfaces for blind people. In *In Proceedings of Sixth International Symposium on Signal Processing and its Applications (ISSPA 2001)*. IEEE, 2001.

[SMIL 2.1, 2006a] Synchronized multimedia integration language media object modules, 2006. http://www.w3.org/TR/SMIL/extended-media-object.html.

[SMIL 2.1, 2006b] Synchronized multimedia integration language timing and synchronization module, 2006. http://www.w3.org/TR/SMIL/smil-timing.html.

[SQLite, 2006] Sqlite, a small sql database engine, 2006. http://www.sqlite.org/.

[Steinmetz, 1996] Ralf Steinmetz. Human perception of jitter and media synchronization. *Selected Areas in Communications, IEEE Journal on*, 14(1):61–72, 1996.

[SWI-Prolog, 2006] Swi-prolog, a comprehensive free software prolog environment, 2006. http://www.swi-prolog.org/.

[The VRML Consortium Incorporated, 2006] Part 1 of iso/iec 14772-1:1997. information technology - computer graphics and image processing - the virtual reality modeling language (vrml) - part 1: Functional specification and utf-8 encoding, 2006. http://tecfa.unige.ch/guides/vrml/vrml97/spec/.

[Wongwirat & Ohara, 2006] Olarn Wongwirat & Shigeyuki Ohara. Haptic media synchronization for remote surgery through simultation. *IEEE Miltimedia*, 13(3):62–69, July-September 2006.

[Woolridge, 2000] Michael Woolridge. *Reasoning About Rational Agents*. MIT Press London, 2000.

[Xerces C++, 2006] Xerces-c++, a validating xml parser, 2006. http://xml.apache.org/xerces-c/.