VTT TIEDOTTEITA – MEDDELANDEN – RESEARCH NOTES 1914

Hajautusalustan suunnittelu reaaliaikasovelluksessa

Tomi Korpipää

VTT Elektroniikka

VALTION TEKNILLINEN TUTKIMUSKESKUS
ESPOO 1998

ISBN 951–38–5315–2 (nid.)
ISSN 1235–0605 (nid.)

ISBN 951–38–5316–0 (URL: http://www.inf.vtt.fi/pdf/)
ISSN 1455–0865 (URL: http://www.inf.vtt.fi/pdf/)

Copyright © Valtion teknillinen tutkimuskeskus (VTT) 1998

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000, FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Toimitus Kerttu Tirronen

LIBELLA PAINOPALVELU OY, ESPOO 1998

Korpipää, Tomi. Hajautusalustan suunnittelu reaaliaikasovelluksessa [Design of a distribution platform for a real-time application]. Espoo 1998, Valtion teknillinen tutkimuskeskus, VTT Tiedotteita – Meddelanden – Research Notes 1914. 58 s. + liitt. 4 s.

Avainsanat LON, distribution platforms, real-time systems, system architecture

Tiivistelmä

Työssä selvitettiin, millaisilla ohjelmistoarkkitehtuuriratkaisuilla voitaisiin toteuttaa joustavia, helposti muunneltavia ja hajautettavia pullo-, tölkki- ja koripalautus-automaattien ohjelmistoja sekä rakennettiin prototyyppi parhaaksi katsotun arkkitehtuuriratkaisun pohjalta. Joustavan hajautetun järjestelmän kehittäminen vaatii, että ohjelmiston suunnitteluvaiheessa otetaan huomioon sekä sovellusalueen yleiset vaatimukset että tulevaisuuden muutos- ja laajennustarpeet.

Eri vaihtoehtoihin tutustumisen ja syventymisen perusteella päädyttiin hajautus-alustamalliin, joka pohjautuu prosessipohjaiseen sovellusalue-spesifisistä osajärjes-telmäkomponenteista koostuvaan ohjelmistoväylään. Tärkeänä pidettiin komponentti-pohjaisuuden lisäksi tietokeskeisyyttä, rajapintojen standardimaisuutta ja järjestelmän konfiguroitavuutta.

Ohjelmistoväylä on järjestelmäkomponentti, jonka tarkoituksena on huolehtia osajärjestelmien välisestä kommunikoinnista ja kätkeä laitteisto- ja käyttöjärjestelmä-riippuvat ratkaisut muilta järjestelmän komponenteilta. Ohjelmistoväylään perustuva arkkitehtuuri lisää järjestelmän joustavuutta. Sen laajennettavuutta, selkeyttä ja ylläpidettävyyttä lisää komponenttien välisten rajapintojen pitäminen standardimaisena.

Standardilaitteistoratkaisuun toteutettuna ohjelmistoväylä on toimiva ja erottaa toteutuskohtaiset asiat sovelluskohtaisista ratkaisuista hyvin. Väylän saaminen prototyyppiasteelle osoittautui mahdolliseksi noin puolen henkilövuoden työllä. Siitä saadut tulokset ovat rohkaisevia jatkokehittämistä ajatellen.

Korpipää, Tomi. Hajautusalustan suunnittelu reaaliaikasovelluksessa [Design of a distribution platform for a real-time application]. Espoo 1998, Technical Research Centre of Finland, VTT Tiedotteita – Meddelanden – Research Notes 1914. 58 p. + app. 4 p.

Keywords LON, distribution platforms, real-time systems, system architecture

Abstract

The aim of the work presented in this thesis was to find out what kind of a system architecture could be used to realize flexible, easily modifiable software for distributed systems, and to implement a prototype using the best architecture solution found. For a distributed system to fulfil flexibility requirements, special attention must be paid in the design of the software to the generic requirements of the application domain and the future needs for modifications and expansions.

After a careful study of alternatives, a task-based software-bus consisting of application domain specific components was decided to be used. Data-centricity, standard interfaces and the configurability of the system were seen as important requirements to take into consideration.

A software bus is a system component, whose purpose is to handle communication between sub-systems, and to hide hardware and operating system dependent solutions from the rest of the system. An architecture based on a software bus contributes to the flexibility of the system. If interfaces between the components are standard and lightweight, the extendability, robustness and maintenance of the system is made easier.

If realized with a standard hardware, a software bus isolates effectively implementation specific solutions from application specific solutions. The implementation of a prototype of the bus proved to be possible in a half working year of one person, and the results were encouraging for the future development.

Alkusanat

Tämä työ on tehty VTT Elektroniikassa Oulussa ja sen tuloksia on sovellettu Halton System Oy:n tarjoamaan pilottiin. Työn tarkoituksena oli selvittää, millaisilla ohjelmistoarkkitehtuuriratkaisuilla voidaan toteuttaa helposti muunneltavia ja hajautettavia palautusautomaattien ohjelmistoja sekä pilotoida parhaaksi katsottu ratkaisu.

Työ hyväksyttiin opinnäytetyönä Oulun yliopiston Teknillisen tiedekunnan sähkötekniikan osastolla. Esitän parhaat kiitokseni työn valvojalle apulaisprofessori Juha Röningille ja toiselle tarkastajalle, tutkimusprofessori Veikko Seppäselle.

Kiitän työni ohjaajina toimineita tekn.lis. Harri Perunkaa ja fil.maist. Eila Niemelää asiantuntevasta opastuksesta. Esitän myös kiitokseni VTT Elektroniikan dipl.ins. Jouni Heikkiselle pilotin sovellusalueanalyysistä sekä dipl.ins. Jukka Koutaniemelle sarjaliikenneajurien toteuttamisesta.

Lopuksi vielä kiitokset muille työtovereilleni heidän tuestaan.

Oulussa 3.3.1998

Tomi Korpipää

Sisällysluettelo

1. JOHDANTO.. 9

2. HAJAUTETTUJEN JÄRJESTELMÄARKKITEHTUURIEN VAATIMUKSET... 10
2.1— Reaaliaikajärjestelmät 10
2.2— Hajauttamisella saavutettavat hyödyt 11
2.2.1 Teho ja suorituskyky 12
2.2.2 Modulaarisuus 13
2.2.3 Ylläpidettävyys.. 14
2.2.4 Tuoteperheet.. 14
2.3 Hajautuksen suunnittelussa huomioon otettavat seikat 15
2.4 Ratkaisussa huomioon otettavat seikat 17

3. HAJAUTETUN JÄRJESTELMÄN SUUNNITTELU 19
3.1 Suunnittelun pääkohdat 19
3.2 Sovellusalueanalyysi.. 21
3.2.1 Komponenttien jako osajärjestelmiksi 22

4. TOIMINNALLISUUDEN HAJAUTTAMISTA TUKEVAT RATKAISUT 27
4.1 Kommunikointitavat.. 27
4.1.1 Viestipohjaisuus 27
4.1.2 RPC 28
4.1.3 Asiakas / Palvelin 29
4.1.4 ORB 31
4.1.5 Tilaajamalli... 32
4.1.6 Tietokantapohjaiset järjestelmät.. 33
4.2 Sulautettuihin järjestelmiin soveltuvat liityntätavat 34
4.2.1 CORBA 34
4.2.2 CAN 34
4.2.3 LON 35

5. REAALIAIKAJÄRJESTELMÄN HAJAUTUSALUSTAMALLI.......................... 39
5.1 Hajautusmallin kehittäminen.. 39
5.1.1 Hajauttaminen ja muunneltavuus 39
5.2 Hajautusmallin sisältö.. 40
5.3 Hajautusmallin joustavuus ja muut ominaisuudet 42

6. KOEJÄRJESTELMÄ ... 43
6.1 Laitteisto ... 43
6.2 Palautusautomaatin ohjelmistoarkkitehtuurin suunnittelu ... 44
6.2.1 Osajärjestelmäkomponentit ... 45
6.2.2 Ohjelmistoväylä ... 46
6.2.3 Kommunikaatiorajapinta ... 46
6.3 Pilotin toteutus ... 47
6.3.1 Graafinen käyttöliittymä ... 47
6.3.2 LON-liitynnän toteuttaminen ohjelmistoväylään ... 48
6.3.3 Ohjelmistoväylän RS-232-liitynnät ... 51
6.4 Johtopäätökset ... 51
6.4.1 Järjestelmän joustavuus ... 51
6.4.2 Vastaavuus suunnitelmaan ... 52
6.4.3 Arkkitehtuurikonseptin skaalattavuuden arviointi ... 53
6.5 Jatkokehitysmahdollisuudet ... 54

7. YHTEENVETO ... 55

LÄHTEET ... 56

LIITE A

Lyhenteiden selitykset

CAN Controller Area Network. Paikallisväylä.

CORBA Common Object Request Broker Architecture. Yleinen ORB-arkkitehtuuri.

IDL Interface Definition Language. Oliorajapintojen määrittelykieli ORB:issa.

LON Local Operating Network. Paikallisväylä.

MAC Media Access Control Processor. LON:in neuronipiirillä oleva mediaohjainprosessori.

ODM Organisational Domain Modeling. Organisatorinen kohdealue-mallinnus, eräs sovellusalueanalyysiin annettu toteutusmalli.

OMG Object Management Group. CORBA:n kehittänyt yritys.

ORB Object Request Broker. Oliopyyntövälittäjä.

RPC Remote Procedure Call. Etäkutsu.

SQL Structured Query Language. Rakenteellinen kyselykieli, jota käytetään tietokantojen kanssa kommunikoitaessa.

1. Johdanto

Tuotteiden muunneltavuus ja joustavuus ovat valmistajalle merkittävä kilpailuetu. Tuotteen toimittaminen ja käyttöönotto nopeutuvat ja helpottuvat sekä laatu paranee, mikäli voidaan käyttää valmiina olevia osajärjestelmiä komponentteina ja koota niistä haluttu tuote. Myös uusien tuotteiden kehitys nopeutuu, kun tuotteeseen voidaan lisätä tai siitä poistaa uusia komponentteja tarpeen mukaan.

Tämän työn tarkoitus on selvittää erään kaupallisesti tuetun paikallisväylän soveltumista palautusjärjestelmän hajautusalustaksi. Työssä tarkastellaan joustavan toteutus-arkkitehtuurin suunnittelua sulautettuun järjestelmään, jonka toiminnallisuus voidaan tarvittaessa hajauttaa useaan rinnakkaiseen solmuun. Työssä keskitytään erityisesti sellaisiin komponentoituihin ohjelmistoarkkitehtuuriratkaisuihin, jotka mahdollistavat tuotteen muunneltavuuden koko elinkaaren ajan. Tämä edellyttää joustavuuden suunnittelua ja joustavuusmekanismien integroimista itse tuotteeseen.

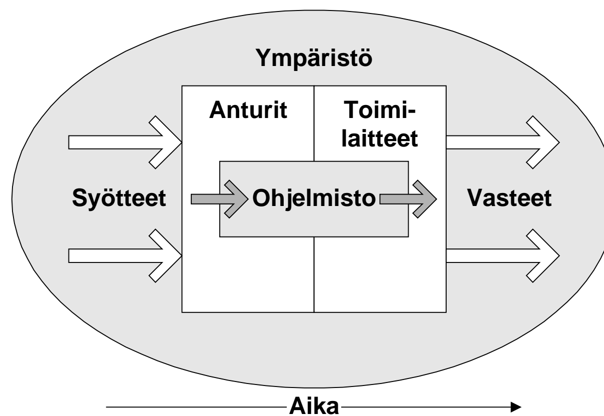
Työn käytännön osuudessa kokeillaan joustavuuden lisäämistä Halton System Oy:n pullo-, kori- ja tölkkipalautusautomaattien ohjelmistoissa. Palautusautomaattien nykyiset järjestelmäarkkitehtuurit ovat keskitettyjä ja siksi niiden muunneltavuus on rajallinen. Eri osajärjestelmien väliset kytkennät on suunniteltu pysyviksi. Muutoksia on varsin vaikea hallita. Myös kaapelointikustannukset ovat huomattavat. Joustavuuden lisäämiseen pyritään sellaisilla teknisillä ratkaisuilla, joilla eritasoiset tuotteet voidaan mahdollisimman pitkälle toteuttaa vakioratkaisuilla. Tavoitteena on lisätä joustavuutta ja muunneltavuutta erityisesti standardoiduilla ja komponentoiduilla ohjelmisto- ja laitteistoratkaisuilla. Hajautuksen myötä kaapelointi- ja asennuskustannusten odotetaan alenevan.

Tässä työssä hajautusalustaksi valittiin kaupallisesti tuettu LON-paikallisverkko, johon on kytketty LON-prosessoreilla toteutettuja solmuja ja PC-yhteensopivia laitteita QNX-käyttöjärjestelmällä varustettuina. Työn tarkoitus oli selvittää, kuinka palautus-automaattien joustavuutta voitaisiin lisätä hajautusalustalla ja arvioida LON-verkon käyttökelpoisuutta hajautusalustan toteutuksessa. Edeltävänä työnä oli jo tehty hajautetun järjestelmäarkkitehtuurin hahmotelma [1], jota käytettiin selvityksen pohjana.

2. Hajautettujen järjestelmäarkkitehtuurien vaatimukset

2.1 Reaaliaikajärjestelmät

Reaaliaikajärjestelmä koostuu yleensä antureista ja toimilaitteista sekä ohjelmasta, joka toimii niiden välillä [2]. Kuvassa 1 on esitetty tyypillinen reaaliaikajärjestelmä. Anturit ovat laitteita, jotka tarkkailevat ympäristöä ja toimilaitteet laitteita, jotka manipuloivat ympäristöä. Ohjelma, joka niiden välillä toimii, tulkitsee antureilta saamiaan arvoja ja käskee toimilaitteiden suorittaa niiden seurauksena tapahtuvaksi haluttavat toimenpiteet.



Kuva 1. Tyypillinen reaaliaikajärjestelmä.

Reaaliaikaisuus käsitteenä tarkoittaa yksinkertaistettuna sitä, että syötteeseen saadaan vaste välittömästi. Reaaliaikaisuuden saavuttamiseen ja suunnitteluun liittyy huomattavasti ongelmia, joista voidaan poimia kolme tärkeintä: rinnakkaisuus (*concurrency*), epädeterministinen käyttäytyminen ja prosessien dynamiikka [2].

Rinnakkaisuus tarkoittaa sitä, että järjestelmään voi tulla samanaikaisesti useita eri syötteitä, jotka on käsiteltävä viiveettä. Tästä seuraa, että ohjelman on kyettävä suorittamaan useaa prosessia samanaikaisesti. Rinnakkaisuutta voidaan myös tarvita optimoitaessa jaettujen resurssien käyttöä. Erään määritelmän mukaan järjestelmä on rinnakkainen, jos sen on suoritettava samanaikaisesti vähintään kahta toisistaan riippuvaa prosessia.

Epädeterministisyys tarkoittaa kyvyttömyyttä ennustaa varmuudella tulevien tapahtumien ajankohtia ja niiden tapahtumisjärjestystä. Täysin ennustettavissa olevaa determinististä järjestelmää on hankala toteuttaa, sillä jopa kaikkein deterministisimmissä järjestelmissä täytyy varautua vähintään muutamiin epädeterministisiin lähteisiin. Näihin lukeutuvat muun muassa ohjelmavirheet ja laitteistoviat, joita ei voida kätkeä.

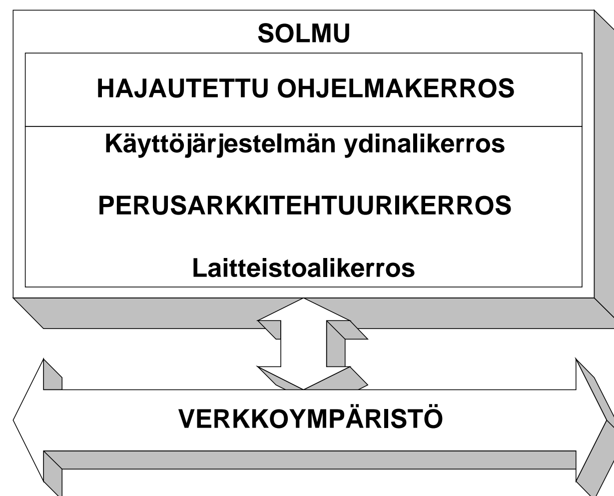
Prosessien dynamiikalla tarkoitetaan ympäristön aiheuttamaa, tilanteen mukaan vaihtelevaa kuormaa, josta järjestelmän on selvittävä. Siksi järjestelmää suunniteltaessa on va-

rauduttava normaalikuormituksen lisäksi kuormahuippuihin, joita saattaa aiheutua ympäristössä tapahtuvien asioiden vuoksi. Tämä on erotettu omaksi ongelmakohdakseen epädeterministisyydestä sen vuoksi, että kuormahuippujen esiintymisajankohtien ja amplitudien ennustaminen on mahdollista jonkinlaisella tarkkuudella. Siksi niihin voidaan suhtautua eri tavalla kuin muihin epädeterministisiin piirteisiin, jotka nimensäkin perusteella ovat täysin ennalta-arvaamattomia.

2.2 Hajauttamisella saavutettavat hyödyt

Ohjelmistojen modularisointi uudelleenkäytettäviksi komponenteiksi on tulossa yhä tärkeämmäksi keinoksi toteuttaa joustavia järjestelmiä kustannustehokkaasti. Komponentoinnin lisäksi joustavuutta olennaisesti lisäävä keino on järjestelmientoiminnallinen hajauttaminen. Kaupallisten verkko- ja hajautusratkaisujen yleistymisen myötä toiminnallinen hajauttaminen nähdään keinoksi toteuttaa joustavia järjestelmiä. Niissä sovellukset voivat sijaita järjestelmän eri osissa mahdollisimman tarkoituksenmukaisella ja optimaalisella tavalla. Tämän lisäksi järjestelmien arkkitehtuurien tulisi tukea kaupallisten valmisohjelmistojen hyödyntämistä sellaisissa osissa, jotka eivät kuulu valmistajayritysten ydinosaamisen alueelle. Tuotekehityksessä voidaan tällöin keskittyä olennaiseen ja tukiosat hankkia ulkopuolisista lähteistä. Tämän onnistumiseksi ohjelmistojen osalta tarvitaan kuitenkin toimintatapaa tukeva ohjelmistoarkkitehtuuri.

Hajautettu reaaliaikajärjestelmä koostuu joukosta autonomisia osajärjestelmiä, jotka suorittavat tehtäviä yhteisten tavoitteiden saavuttamiseksi [3]. Laitteisto ja ohjelmisto voidaan yhdistää useilla eri tavoilla tietyn sovelluksen rakentamiseksi. Kuvassa 2 [4] on esitetty hajautetun ohjelman perusrakenne.

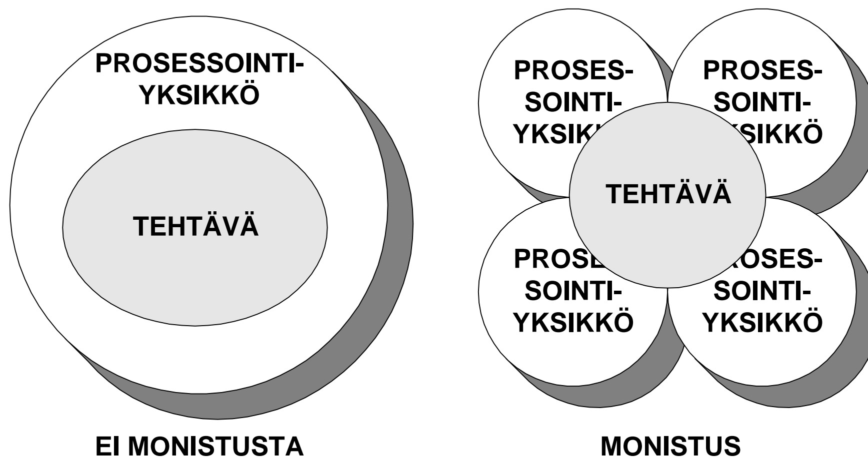


Kuva 2. Hajautetun ohjelman rakenne.

2.2.1 Teho ja suorituskyky

Teho ja suorituskyky ovat merkittäviä tekijöitä järjestelmän toiminnallisuuden ja elinkaaren kannalta. Monissa tapauksissa, varsinkin pienemmissä järjestelmissä, keskitetyllä ratkaisulla saatetaan saavuttaa parempi suorituskyky kuin vastaavalla hajautetulla ratkaisulla. Kuitenkin mentäessä kohti monimutkaisempia järjestelmiä suorituskykyä voidaan kasvattaa hajauttamalla toiminnallisuutta useaan rinnakkaiseen solmuun. Monistus ja tehtävien rinnakkaissuoritus, dynaaminen tehtävien allokointi sekä tietokantojen hajautus ovat esimerkkejä tällaisesta hajauttamisesta.

Erityisesti tehokkuutta ja laskentakykyä vaativia reaaliaikajärjestelmiä voidaan toteuttaa käyttäen monistusta. Monistus tarkoittaa sitä, että samaa laitteistoa otetaan käyttöön useampi kuin yksi kappale, ja jaetaan tehokkuutta vaativa tehtävä suoritettavaksi useiden prosessointiyksiköiden kesken [3]. Tästä on esimerkki kuvassa 3. Samalla yhden prosessointiyksikön tehoa voidaan tarvittaessa vähentää. Suorituskyvyn kasvu ei kuitenkaan moninkertaistu samassa suhteessa laitemäärän moninkertaistamiseen, joten monistusta käytettäessä on otettava huomioon myös kustannuskysymykset.



Kuva 3. Monistamisen periaate.

Läheisessä suhteessa monistuksen kanssa on rinnakkainen tehtävien suoritus. Suorituksen hajauttaminen rinnakkaisesti tehtäviin prosesseihin vähentää kilpailua jaetuista palveluista sekä jonotusviiveitä verrattuna keskitettyyn ratkaisuun [4, 5]. Rinnakkaisten prosessien toteutusta suunniteltaessa on otettava huomioon prosessien välisen kommunikoinnin järjestäminen. Kommunikointitavan valintaan vaikuttavia tekijöitä ovat muun muassa siirrettävä tietomäärä, tiedonsiirtotiheys ja prosessien välisten siirtojen suhde toisiinsa [4].

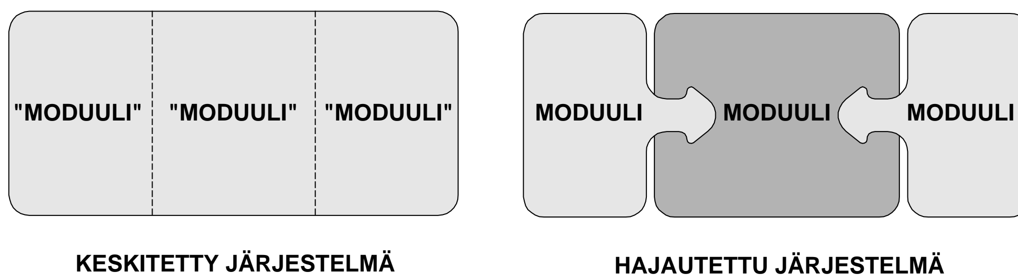
Dynaamisella tehtävien allokoinnilla staattisen sijaan voidaan myös vaikuttaa järjestelmän suorituskykyyn [3]. Dynaamisella allokoinnilla tarkoitetaan kullakin hetkellä va-

paana olevien resurssien käyttämistä tehtävän suorittamiseen sen sijaan, että tehtävän suoritus on määrätty tietylle prosessointiyksikölle. Staattisessa allokoinnissa tehtävä suoritetaan ennalta määrätyssä prosessointiyksikössä riippumatta siitä, onko kyseisellä prosessointiyksiköllä resursseja vapaana sillä hetkellä vai ei. Dynaamista allokointia käytettäessä tehtävän prosessointiyksikkö voi siis vaihdella järjestelmän kuormituksen mukaan.

Myös tietokantojen hajautusta käytetään suorituskyvyn parantamiseen. Tietokantojen hajautuksella tavoitellaan lyhyempiä vasteaikoja ja parempaa käytettävyyttä, jotka perustuvat tietojen fyysiseen läheisyyteen niitä tarvitsevien prosessien ja laitteiden kanssa [6].

2.2.2 Modulaarisuus

Modulaarisuus sallii järjestelmän eri osien toiminnallisen erikoistumisen [5] ja helpottaa järjestelmään tulevaisuudessa mahdollisesti tehtäviä päivityksiä ja laajennuksia. Rajapinnat moduulien välillä määrittävät moduulien kytkemisen toisiinsa. Kuvassa 4 on esitetty periaate moduuleista ja niiden välisistä rajapinnoista keskitetyssä ja hajautetussa järjestelmässä. Rajapintapohjaisen arkkitehtuuriajattelun pitäisi johtaa luonnostaan oikeankokoisten osakokonaisuuksien ja moduulien syntymiseen [6], mikä on tärkeää pyrittäessä ohjelmistoihin sisäänrakennettuun joustavuuteen.



Kuva 4. Modulaarisuus keskitetyssä ja hajautetussa järjestelmässä.

Parhaimmillaan modulaarisuudella voidaan saavuttaa tilanne, jossa minkä tahansa komponentin tai moduulin vikaantuminen häiritsee vain rajattua osaa järjestelmästä [5] muiden osien pystyessä jatkamaan toimintaansa häiriintymättä.

Jos järjestelmä voidaan vielä suunnitella niin, että muutokset voidaan toteuttaa turvallisesti ja asteittain, saavutetaan samalla hyvät valmiudet muutoskustannusten alentamiseen. Tämä taas näkyy kolmella merkittävällä tavalla [7]:

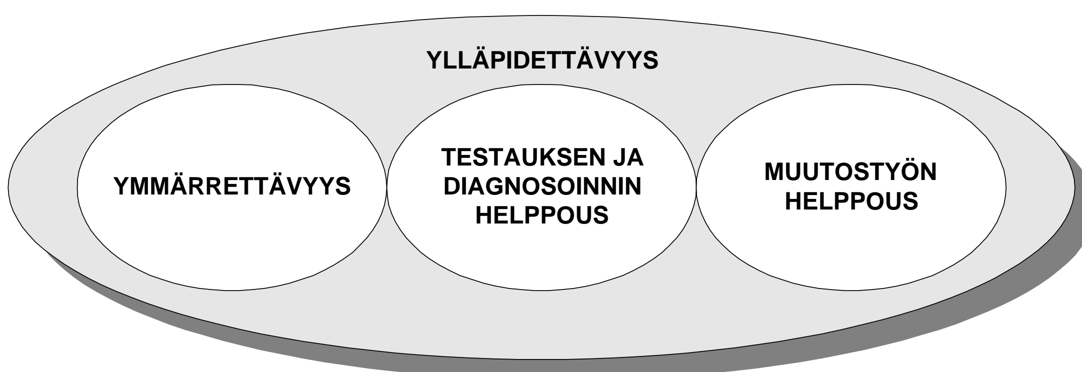
Olemassaolevan järjestelmän jo toimiessa pienen lisäyksen aiheuttamat testikustannukset ovat pienet.

Uudet ja mahdollisesti vialliset lisäykset eivät vaikuta aiempiin toimiviin perustoimintoihin.

Uusien lisäyksien tekeminen on helppoa.

2.2.3 Ylläpidettävyys

Ylläpidettävyys koostuu monesta tekijästä, jotka voidaan jaotella kolmeen luokkaan [8]: ymmärrettävyyteen, testauksen ja diagnosoinnin helppouteen, sekä muutostyön helppouteen. Nämä pääluokat on esitetty kuvassa 5.



Kuva 5. Ylläpidettävyyden pääluokat.

Ymmärrettävyyttä mitataan ulkopuolisen katselmoijan kyvyllä käsittää ohjelmiston rakenne, rajapinnat, funktiot ja ohjelman sisäinen toiminta. Näihin vaikuttavia tekijöitä ovat modulaarisuus, suunnitteludokumentaatio, koodin kommentointi ja suunnitteluperiaate [8]. Diagnosoinnin ja testauksen helppous riippuvat suuresti ymmärrettävyydestä, ja myös dokumentaatio on tässä suhteessa tärkeä. Lisäksi asiaan vaikuttavat olemassa olevien testaus- ja virheenkorjaustyökalujen laatu, ohjelman rakenne sekä aiemmin määritellyt testiproseduurit [8]. Muutostyön helppouteen vaikuttavat muun muassa ohjelmiston sisäiset liitynnät ja rakenne.

2.2.4 Tuoteperheet

Helposti ylläpidettävien hajautettujen järjestelmien tulisi koostua autonomisista ohjelmistopaketeista, joita voidaan mielivaltaisesti liittää tai poistaa järjestelmästä [9, 10]. Tällaista lähestymistapaa kutsutaan tuoteperhelähtöiseksi. Tuoteperheellä viitataan juuri ohjelmistokomponentteihin, joita voidaan yhdistellä ja erotella ilman, että järjestelmään täytyy tehdä suuria erillisiä muutoksia.

Optimaalisessa tilanteessa järjestelmään ei tarvitse tehdä lainkaan muutoksia tuoteperheeseen kuuluvien komponenttien lisäämisen tai poistamisen vuoksi, vaan järjestelmästä saadaan halutunlainen suorittamalla ainoastaan tarvittavat konfiguroinnit komponenteille. Tuoteperheiden tavoitteena on laajennettavuus, skaalattavuus ja joustavuus. Geneeristen komponenttien uudelleenkäyttö johtaa yleensä myös tuotteen laadun paranemiseen. Tuoteperhe yleensä pidentää yksittäisen tuotteen elinikää.

2.3 Hajautuksen suunnittelussa huomioon otettavat seikat

Prosessien välinen kommunikaatio on usein vaikein ja eniten ongelmia tuottava osuus hajautettujen järjestelmien suunnittelussa. Se on kuitenkin myös yksi tärkeimmistä, ellei jopa tärkein kysymys, joten siihen on kiinnitettävä huomiota aivan järjestelmän suunnittelun alkuvaiheista lähtien. On löydetty monia syitä, miksi kommunikaation järjestäminen on niin ongelmallista [11]. Näitä syitä on esitetty taulukossa 1.

Hajautuksen järjestämiseksi on olemassa useita, joskus ristiriitaisiakin standardeja. Yleisesti ottaen standardit määrittelevät vain rajapinnat ja perustuvat pyyntö-vastaus (*request-reply*) -malliin, joka ei ole helposti skaalattavissa laajoihin hajautettuihin järjestelmiin [11]. Lisäksi standardit eivät ota kantaa useisiin kriittisiin kysymyksiin, kuten virheistä toipumiseen, virhesietoisuuteen ja tiedon jakamiseen, jotka ovat erittäin tärkeitä järjestelmän osatekijöitä. Toiminnallinen hajautus on vähäistä, yleensä vain I/O:n hajautus on toteutettu. Nämä puutteet haittaavat laajennettavuutta, sillä vaikka sovittaen saadaan aikaiseksi useita erilaisia järjestelmiä, ratkaisuiden määrä asettaa liikaa rajoituksia. Hajautuksen muunneltavuus on myös hankalaa, järjestelmän kapasiteettia tuhlataan ja ylläpito vaikeutuu.

Joskus hajautuksella ei saavuteta haluttua hyötyä. Syynä voi olla, että ratkaisussa ei ole kyetty selkeästi erottamaan toteutusteknologiaa ja sovellusalueen mallintamista toisistaan. Tällainen vaikuttaa aina järjestelmän laajennettavuuteen ja joustavuuteen, koska sovelluksesta tulee teknologiaan sidottu. Esimerkiksi vajavaisten verkkorajapintamäärittelyiden vuoksi verkkoratkaisu saattaa heijastua sovellukseen.

Taulukko 1. Kommunikaation järjestämiseen liittyviä ongelmia.

Ongelma	**Selitys**
Tiedon muotoilu	Tieto, joka siirretään prosessilta toiselle prosessille, täytyy yleensä koodata sellaiseen muotoon, että vastaanottaja ymmärtää sen. Viestin vastaanotettuaan prosessin täytyy myös tietää, miten viestin sisältämään tietoon päästään käsiksi ja mitä sille tulee tehdä, jotta sitä voidaan hyödyntää.
Tietotyyppien tulkkaus	Erilaiset tietotyypit täytyy pystyä tulkitsemaan kussakin kohteessa, jotta ne ymmärretään.
Usean protokollan tuki	Kun tieto liikkuu prosessista toiseen, ja ehkä myös järjestelmästä toiseen, käytettyä siirtoprotokollaa saatetaan joutua muuttamaan.
Resurssien löytäminen	Suuren hajautetun järjestelmän rakentamiseksi on useista resursseista pidettävä tarkkaan kirjaa. Jos näiden resurssien tulee jakaa tietoa keskenään, niiden yleensä täytyy myös tietää, missä muut resurssit sijaitsevat ja miten niiden kanssa kommunikoidaan. Resurssien sijainti ja asema puolestaan vaihtelevat sitä mukaa, kun uusia laitteita liitetään järjestelmään tai poistetaan siitä. Järjestelmän kykyä löytää ja tarkkailla resursseja kutsutaan yleisesti nimeämispalveluiksi (*naming services*).
Tietovirran valvonta (*flow control*)	Hajautetussa järjestelmässä prosessit saattavat joskus "kadottaa" järjestelmässä liikkuvaa tietoa, esimerkiksi jos ne ovat suorittamassa jotain tehtävää juuri tiedon saapuessa. Ohjelman suunnittelu tällaisten tilanteiden varalta on hyvin vaikeaa, etenkin jos tietovirran intensiteetti vaihtelee suuresti.
Siirrettävyys ja standardi-ratkaisut	On olemassa useita standardeja tiedonsiirtoon hajautetussa järjestelmässä. Yksi yksikäsitteinen standardi puuttuu.
Asynkroniset operaatiot	Prosessin täytyy usein lähettää tietoa jäämättä odottamaan vastausta. Tämä tarkoittaa, että tiedon lähetyksestä seuraava mahdollinen vaste saadaan yhdeltä tai useammalta prosessilta tuntemattomana ajankohtana joskus tulevaisuudessa. Luotettavan asynkronisen kommunikoinnin järjestäminen hajautettuun järjestelmään on suuri ongelma.
Resurssien ja prosessien konfigurointi	Kun käytetään nimettyjä resursseja tai prosesseja, uusien ominaisuuksien lisääminen tai vanhojen poistaminen on vaikeaa.
Laitteiston ja ohjelmiston virheiden käsittely	Järjestelmän osien osittainen tai täydellinen vikaantuminen on yhä suurempi ongelma järjestelmien koon kasvaessa. Virheistä palautuminen voi olla hyvinkin vaikeaa, eikä se usein ole mahdollista täydellisesti tai edes siedettävästi.
Useiden asiakkaiden ja palvelinten hallinta	Hajautetussa järjestelmässä useat prosessit voivat vaihtaa tietoa keskenään eri yhteyksien kautta. Usein prosessin täytyy lähettää viesti monille vastaanottajille samanaikaisesti. Joskus taas prosessin tulisi lähettää viesti koko järjestelmän tai sen määrätyn osan vähiten kuormitetulle prosessille, jotta vaste saataisiin mahdollisimman lyhyessä ajassa. Monien prosessien näennäisen sattumanvaraisen keskinäisen kommunikoinnin mahdollistaminen on hyvin vaikeaa.
Ympäristön vaihto	Järjestelmän verkkoympäristö voi olla tiheään muuttuva tekijä. Kun järjestelmään lisätään tai siitä poistetaan laitteita, verkkoympäristö muuttuu. Tämän automaattinen hallinta on ongelmallista.
Virheenetsintä ja analysointi	Hajautetun järjestelmän tehokas virheenetsintä (*debugging*) vaatii tarkkaa näkymää prosessien väliseen kommunikointiin. Yleisesti prosessien välisen kommunikoinnin käyttöjärjestelmäpohjaisten mekanismien seurantamahdollisuudet ovat lähes olemattomat, ja tämän vuoksi hajautettujen järjestelmien virheenetsintä jää usein vajaaksi. Ongelma vielä korostuu, mikäli ollaan suunnittelemassa skaalattavaa verkon yli hajautettua järjestelmää.

Ohjelmiston jäykillä rakenteilla tarkoitetaan joustavuuden kärsimistä ohjelman rakenteen vuoksi. Tämä voi johtua esimerkiksi puutteellisesti määritellyistä rajapinnoista järjestelmän eri osien välillä tai toimintojen epätarkoituksenmukaisesta sirottelusta järjestelmän eri osiin, vaikka niiden keskittäminen olisi järkevää. Rajapintamäärittelyt voivat vaihdella eri osien välillä, kaikkien ollessa tavalla tai toisella epästandardeja. Usein myös sovellusaluekohtainen kommunikointi on esimerkiksi optimointisyistä siroteltu

sovelluksen eri osiin. Ylläpidon kannalta sen tulisi sijaita erillisessä kommunikointiosajärjestelmässä, joka hoitaa kaiken kommunikaation. Koska hajautustuki on yleensä vain viestienvälitysmekanismi ja siihen liittyvä protokollapino, epäselvät rajapinnat ja toimintojen sirottelu ovat seurausta suunnitteluongelmista, joihin helposti ajaudutaan tuotteen toimituskiireiden ja puutteellisen asiantuntemuksen vuoksi.

2.4 Ratkaisussa huomioon otettavat seikat

Kaupalliset ratkaisut mahdollistavat hajautuskonseptin toteuttamisen järkevästi. Samalla syntyy kustannussäästöjä täysin omien ratkaisujen suunnittelu-, kehitys- ja toteutuskustannuksien vähentyessä. Ratkaisua etsittäessä kaupallisen tarjonnan hyödyntäminen on tuotekehitysprosessia nopeuttava ja kustannussäästöjä edistävä mahdollisuus, joka kannattaa ottaa huomioon varteenotettavana vaihtoehtona.

Monet hajautukseen liittyvistä ongelmista voidaan ratkaista ohjelmistoalustalla (*middleware*). Alustalla tarkoitetaan tässä ohjelmaa, jota käytetään siirtämään tietoa ohjelmalta toiselle [11]. Se on ohjelmakerros, joka kätkee kommunikointiprotokollan, käyttöjärjestelmän ja laitteistoalustat niitä käyttäviltä ohjelmilta. Ohjelmistoalustat voidaan jakaa neljään luokkaan: viestipohjaiset, tietokantapohjaiset, etäkutsupohjaiset (*remote procedure call*) ja oliokutsupohjaiset (*object request broker*). Näitä käsitellään tarkemmin luvussa 4.

Hajautetut reaaliaikajärjestelmät ovat usein heterogeenisia sulautettuja järjestelmiä, joiden vasteaikavaatimukset, muistinkäyttö, käyttöympäristö ja monet muut asiat vaihtelevat eri osien välillä. Ohjelmiston joustavuus, skaalattavuus ja selkeys voidaan saavuttaa kehittämällä sovellusaluekohtainen hajautettu ohjelmistoalusta ja siihen sopivat komponentit. Hajautusalustalle ja komponenteille on olemassa tiettyjä vaatimuksia [9], jotka on esitetty taulukossa 2.

Taulukko 2. Vaatimukset hajautusalustalle ja komponenteille.

Vaatimus	Selitys
Muunneltavuus	Hajautusalustan täytyy sisältää palvelut järjestelmän ja komponenttien konfiguroinnin hallitsemiseksi.
Ohjelmiston riippumattomuus ja laajennettavuus	Sovelluskomponentit täytyy voida sijoittaa vapaasti mihin tahansa solmuun järjestelmässä ja hajautusalustan tulee pystyä tarjoamaan läpinäkyvät kommunikointimekanismit komponenteille. Sovelluskomponenttien tulee lisäksi olla niin autonomisia kuin mahdollista, jotta järjestelmän joustavuus säilyy.
Heterogeeninen toteutusympäristö	Tyypillisen sulautetun reaaliaikasovelluksen muisti- ja ajoitusvaatimukset pitää pystyä takaamaan heterogeenisissä ympäristöissä.

Näiden vaatimusten lisäksi on otettava huomioon tekijät, jotka vaikuttavat järjestelmän hajautettavuuteen [6]. Nämä asiat on esitetty taulukossa 3.

Taulukko 3. Järjestelmän hajautettavuuteen vaikuttavat tekijät.

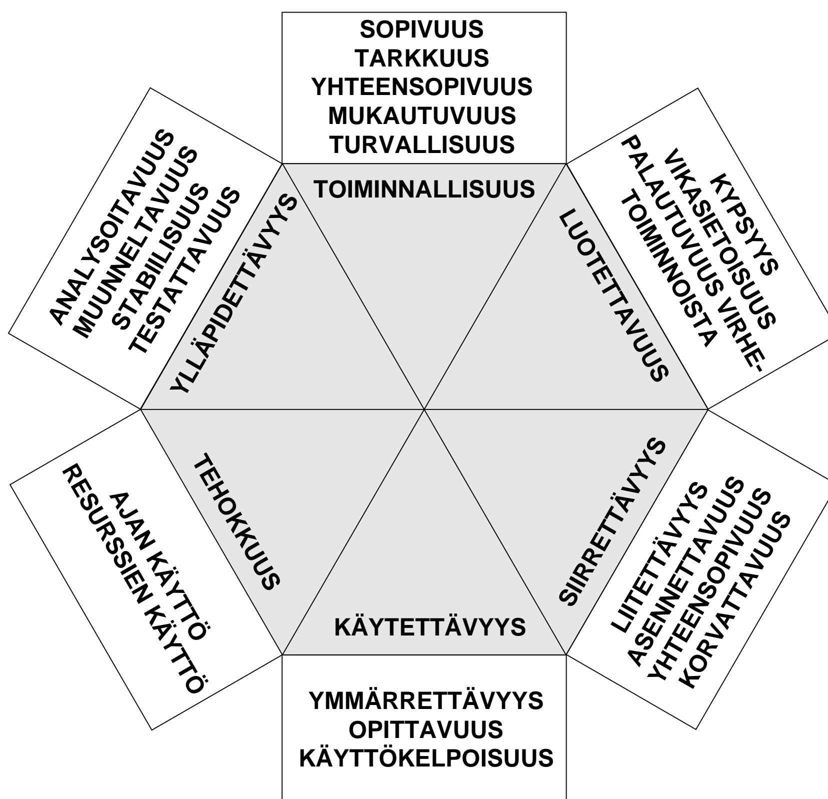
	Hajautettavuustekijä
1	Ulkoisten liittymien lukumäärä ja luonne. On selvitettävä, ovatko liittymät eristettävissä, miten tiedonsiirto hoidetaan sekä haluttu siirron suunta.
2	Todellinen tarve tietojen ajantasaisuuteen sekä ajantasaisesti ylläpidettävien tietojen käyttötavat.
3	Tietovarastojen osittamisen mahdollisuudet, yhteiskäyttöisten tietojen määrä, tietojen väliset suhteet ja tiedonpäivitysreitit.

Lisäksi jo suunnittelussa on otettava huomioon, mihin tarkoituksiin järjestelmän tulee sopia ja millaisia muutoksia tulevaisuudessa on odotettavissa, samoin kuin järjestelmän ylläpidettävyyden helppous ja kustannustekijät [12].

3. Hajautetun järjestelmän suunnittelu

3.1 Suunnittelun pääkohdat

Mitä tahansa ohjelmistoa tai järjestelmää suunniteltaessa on otettava huomioon kuvassa 6 esitettyjä seikkoja, jotka yleensä luokitellaan laatutekijöiksi [13].



Kuva 6. Suunnittelussa huomioon otettavia tekijöitä.

Laatutekijät on syytä pitää mielessä aivan suunnittelun alusta asti, sillä korkea laatu on tuotteen elinehto. Laadunvalvonta ja -suunnittelu onkin tullut tärkeäksi osaksi ohjelmistokehitysprosessia, ja se oli myös osallisena ratkaisumallissa johon tässä työssä päädyttiin.

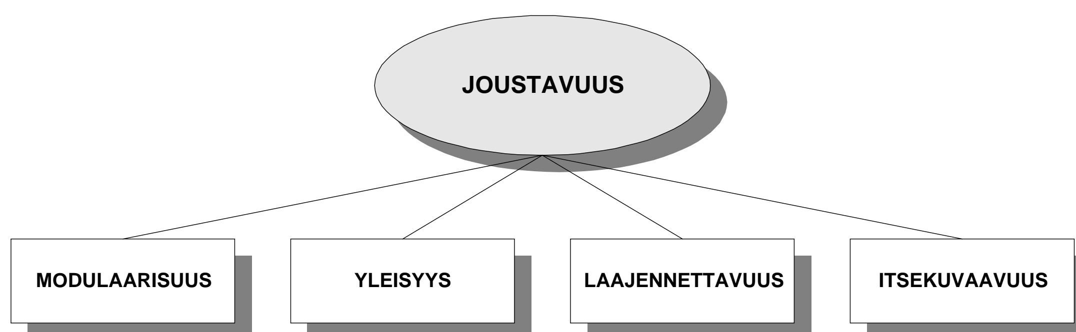
Hajautusratkaisun optimointi on ongelmallista, ja usein suunnittelun näennäisen helppouden vuoksi päädytään joko äärimmilleen hajautettuun tai keskitettyyn ratkaisuun, kun taas optimaalinen ratkaisu olisi jossain näiden välimaastossa. Lisäksi väylästandardien määrä ja jatkuva muuttuminen muodostavat ongelman.

Järjestelmien kehittämiseen liittyy itse järjestelmän suunnittelun lisäksi työtä helpottavien menetelmien ja välineiden suunnittelu [3]. Kehitystyössä on kiinnitettävä huomiota useisiin seikkoihin ja niiden kuvaamiseen. Tärkeitä ovat muun muassa järjestelmän aikakäyttäytymisen määrittely, tehtävien suoritusvälin kuvaaminen, järjestelmän toiminto-

jen mallintaminen, järjestelmän käyttäytymisen mallintaminen sekä mallintamisen tuki hajautuksen suunnittelulle [3].

Koska suunnittelumenetelmät ovat osin selkeytymättömiä, joudutaan usein vuoroin kokeilemaan ja tarkkailemaan kokeilemalla saatuja tuloksia yhä uudestaan, kunnes saavutetaan haluttu lopputulos.

Hajautettujen järjestelmien tapauksessa erityistä huomiota kannattaa kiinnittää joustavuuteen (*flexibility*), joka voidaan vielä jaotella alaryhmiin kuvan 7 [8] mukaisesti.



Kuva 7. Ylläpidettävyyden ja joustavuuden tekijät.

Olennainen osa järjestelmää on sen luotettavuus ja turvallisuus. Näiden ominaisuuksien suunnittelu on aloitettava jo järjestelmän kehitystyön alussa, sillä vaatimusten huomioimatta jättäminen järjestelmän määrittelyvaiheessa vaikeuttaa huomattavasti myöhempää suunnittelua [3]. Luotettavuuden ja turvallisuuden perustana on järjestelmän selkeys, joka hajautettujen järjestelmien tapauksessa korostuu.

Selkeys ja laajennettavuus ovat usein ristiriidassa kustannustehokkuuden ja toiminnallisten optimiratkaisujen kanssa. Suunnittelu on kuitenkin hyvä tehdä alusta alkaen selkeys yhtenä päätavoitteena. Tähän on olemassa suuntaa-antavia ohjeita [4], joita noudattamalla järjestelmän selkeyden pitäisi säilyä. Ohjeet on lueteltu taulukossa 4.

Taulukko 4. Ohjeita selkeyden säilyttämiseen.

	Ohje
1	Omaksutaan vastuu ohjelman turvallisuudesta ja käytettävyydestä. Tällä tarkoitetaan sitä, että ei odoteta muiden osapuolien huolehtivan turvallisuudesta ja käytettävyydestä, vaan huolehditaan niistä itse.
2	Suunnitellaan järjestelmään keskinäisiä liitoksia vain todellisen tarpeen vuoksi, ei siksi että se tuntuu hyvältä idealta.
3	Tuetaan vain ehdottomasti tarpeellisia palveluja.
4	Sisällytetään järjestelmään itsediagnostiikka ja tiedonvarmistusmekanismit.
5	Suunnitellaan järjestelmä vikasietoiseksi olettaen, että virheitä tapahtuu. Vikasietoisuus saavutetaan kehittämällä menetelmä selvitä vikatilanteista tai suorittamalla niiden tapahtuessa hallittu alasajo ennen kuin on liian myöhäistä.
6	Suunnitellaan järjestelmä skaalattavaksi, jotta palvelu- ja resurssivaatimusten kasvaessa ei jouduta umpikujaan ja mahdollisesti uusimaan koko järjestelmä.
7	Pyritään välttämään mekanismeja, jotka voivat aiheuttaa useiden virheiden syntymisen yhden virheen seurauksena.

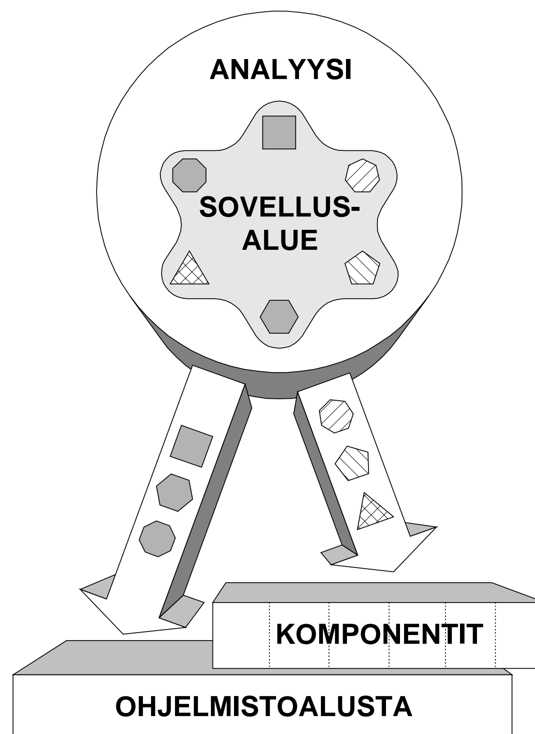
Selkeyden toteuttaminen saattaa osoittautua kalliiksi, koska usein sen lisääminen järjestelmään aiheuttaa kustannuksia ja rajoituksia suorituskyvylle [4].

Tilannetta kannattaa kuitenkin tarkastella tapauskohtaisesti ja pidemmällä aikavälillä. Joustava ja laajennettava järjestelmä tuo kustannussäästöjä ja kilpailukykyä tuotteen ylläpidon aikana, koska uusien ominaisuuksien lisääminen ja olemassa olevien muuttaminen voidaan tehdä tehokkaasti.

3.2 Sovellusalueanalyysi

Sovellusalueanalyysissä selvitetään asiakkaan tarpeet sekä ohjelmiston ja järjestelmän toteuttamiseen soveltuvat teknologiat ja kehitysprosessit. Analyysissä pyritään erityisesti löytämään sovellusalueen tuotteista yhteisiä piirteitä sekä tunnistamaan poikkeavat piirteet ja tuotesovellusten väliset suhteet [15, 16, 17]. Kuvassa 8 on esitetty sovellusalueanalyysin periaate. Analyysin tulisi selvittää erityisesti seuraavat asiat:

- Uudelleenkäytettävien komponenttien kehittämiseen tarvittava taustatieto.
- Ohjelmistoalustan kehittäminen komponenteista kootulle järjestelmälle.



Kuva 8. Sovellusalueanalyysi.

Sovellusalueanalyysin suorittamiseen on olemassa periaatteellisia ohjeita ja toteutusmalleja, joista yksi on ODM (Organisational Domain Modeling) [18]. ODM-menetelmän mukaan suoritettu sovellusalueanalyysi koostuu useasta vaiheesta:

- Sovellusaluetietämyksen hankkiminen. Kerätään sovellusaluetta koskevat tiedot tutkimalla järjestelmävaatimuksia ja -dokumentaatiota sekä keskustelemalla asiantuntijoiden kanssa.

- Kuvaavan mallin kehittäminen. Mallinnetaan järjestelmäkonsepti kiinnittäen erityistä huomiota seikkoihin, jotka ovat vakioita järjestelmän sisällä, sekä seikkoihin, jotka ovat muuttuvia.

- Sovellusaluemallin tarkentaminen. Yhdistetään erilliset kuvaavat mallit yhdeksi aukottomaksi malliksi.

- Ohjelmistoalustan suunnittelu komponenteille.

- Komponenttien toteutus. Toteutetaan komponentit ohjelmistoalustan mukaan ja luodaan perusrakenne komponenttien organisointiin.

Ohjelmistoalustan tehtävä on tarjota perusta, jonka "päälle" komponenteista voidaan rakentaa halutut vaatimukset täyttävä kokonaisuus ilman laitteistovaatimusten ja ohjelmiston sekoittamista toisiinsa. Ohjelmistoalustan kehittämisessä on otettava huomioon analyysistä saadut tiedot sekä laitteiston, käyttöjärjestelmän ja verkkoratkaisun asettamat vaatimukset ja mahdollisuudet. Alustan tulee sisältää tieto, miten nämä liittyvät toisiinsa ja komponentteihin.

3.2.1 Komponenttien jako osajärjestelmiksi

Komponenttien tai toimintojen ryhmitteleminen osajärjestelmiksi tyypin mukaan on hajauttamista helpottava menetelmä [10, 20]. Yksi osajärjestelmä voi koostua tarpeen mukaan yhdestä tai useammasta rinnakkaisesta prosessista.

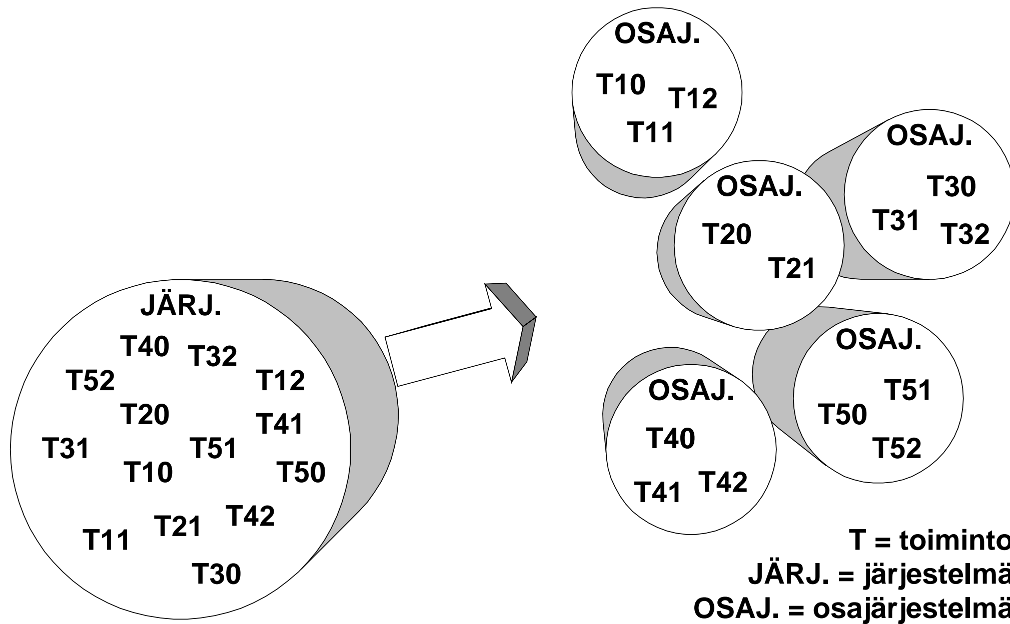
Osajärjestelmäjako voidaan suorittaa analyysistä saatujen tulosten perusteella esimerkiksi toiminnallisten kokonaisuuksien mukaisesti, toimintojen suoritustyylin mukaisesti tai toteutustavan mukaisesti [14]. Seuraavaksi käsitellään esimerkkinä jakoa toiminnallisuuden perusteella.

Toiminnot voidaan ryhmitellä osajärjestelmiksi niiden toiminnallisen luonteen perusteella taulukossa 5 esitettyjen suuntaviivojen mukaan [20].

Taulukko 5. Osajärjestelmäryhmittely toiminnallisuuden perusteella.

Toiminnallisuus	Selitys
Reaaliaikainen ohjaus tai rinnakkaisohjaus	Tämäntyyppisessä osajärjestelmässä ohjataan tyypillisesti rajattua järjestelmän osaa. Tapauksissa, joissa on useampia reaaliaikaohjausjärjestelmiä, on järjestettävä mekanismi näiden ohjaukseen.
Tiedon kerääminen tai analysointi	Tavallisesti reaaliaikajärjestelmän yhtenä tehtävänä on tietojen kerääminen ympäristöstä ja prosessointi. Näistä huolehtivat toiminnot kannattaa sijoittaa omiin osajärjestelmiinsä.
Palvelimet	Palvelinosajärjestelmät koostuvat toiminnoista, jotka suorittavat muiden osajärjestelmien pyytämiä palveluita. Tietojen hallinta ja I/O ovat tyypillisiä palvelinosajärjestelmän toimintoja.
Käyttäjän palvelut	Käyttöliittymä ja mahdollisesti myös muita järjestelmän käyttämiseen liittyviä toimintoja kannattaa koota omaksi osajärjestelmäkseen.
Järjestelmätason palvelut	Tällaiseen osajärjestelmään kannattaa sijoittaa käyttöjärjestelmäriippuvat toiminnot, kuten esimerkiksi tiedostojen hallinta ja verkkopalvelut.

Kuvassa 9 on esitetty esimerkki osajärjestelmäjaottelusta toiminnallisuuden perusteella.



Kuva 9. Toimintojen ryhmitteleminen osajärjestelmiksi.

Osajärjestelmien välinen tiedonsiirto kannattaa hoitaa sanomapohjaisena, koska tällöin osajärjestelmät voi kommunikaation osalta sijoittaa vapaasti eri solmuihin, eli verkkoon liitettyihin prosessointiyksiköihin [20].

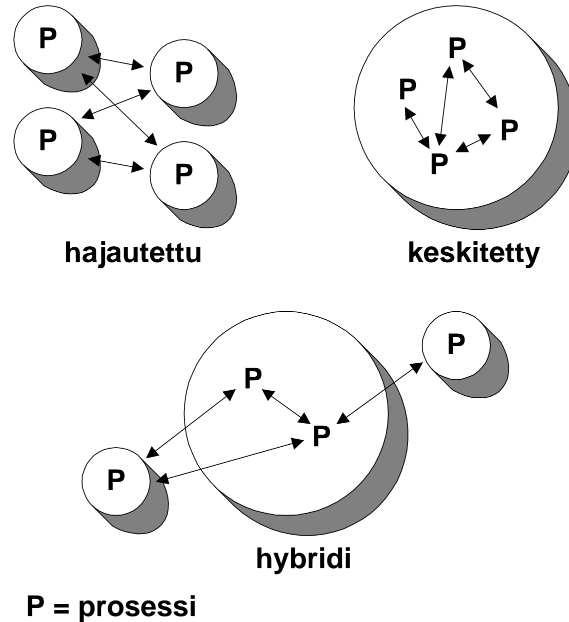
Osajärjestelmien määrittelyn jälkeen voidaan harkita niiden allokointia järjestelmän eri solmuille. Periaatteessa kaikki osajärjestelmät voidaan allokoida yhteen solmuun, jolloin tuloksena on keskitetty järjestelmä [3]. Tämä ei kuitenkaan ole hajautuksen tarkoitus, vaan tarkoituksena on sijoittaa osajärjestelmät erillisiin prosessointiyksiköihin opti-

maalisesti. Käytännössä allokoinnissa kannattaa ottaa huomioon taulukossa 6 esitetyt seikat [20]. Taulukko 6. Allokoinnissa huomioitavat seikat.

Taulukko 6. Allokoinnissa huomioitavat seikat.

Asia	Tarkennus
Järjestelmän solmujen fyysinen sijainti	Yhtenä syynä hajautukselle on usein se, että järjestelmän eri prosessointiyksiköt sijaitsevat etäällä toisistaan. Tällöin voi olla kannattavaa sijoittaa esimerkiksi tietoa käsittelevä osajärjestelmä tiedon lähteen yhteyteen, jotta tiedonsiirtoon kuluva aika ei pääse haittaamaan järjestelmän toimintaa.
Paikallinen autonomia	Osajärjestelmä ohjaa järjestelmän tiettyä sijaintiriippuvaa osaa joko ylemmän tason alkuunpanevan ohjeen mukaan tai täysin autonomisesti. Olettaen, että osajärjestelmä toimii muista solmuista riippumattomasti, järjestelmän vikasietoisuus paranee huomattavasti, koska koko järjestelmän toimivuus ei ole riippuvainen tietystä ehkä vain väliaikaisesti toimimattomasta solmusta.
Suorituskyky	Aikakriittisten tehtävien suoritus omissa solmuissaan on usein hyvä keino suoritusaikavaatimusten saavuttamiseksi. Samalla solmujen välisen kommunikoinnin tarve vähenee, mikäli myös osajärjestelmäjako on onnistunut.
Laitteistoratkaisut	Laitepalvelimien sijoittaminen erillisiin solmuihin on hyvä ratkaisu, jos kyseessä on erikoistunut, erillisiä palveluita tarvitseva laite tai laitteiston käsittely vaatii runsaasti kapasiteettia.
Käyttöliittymät	Vaatimuksista riippuen sijoittaminen omaan solmuun voi olla perusteltua, koska näin voidaan esimerkiksi taata jonkinlainen vaste käyttäjälle minimiajassa.
Palvelimet	Palvelimen sijoittaminen omaan solmuunsa on erityisen perusteltua tapauksissa, joissa kyseessä on tiedostopalvelin, joka joutuu esimerkiksi käsittelemään suuria tietokantoja tai I/O-palvelin, joka palvelee jatkuvasti ulkoisia pyyntöjä.
Solmujen välisen kommunikoinnin minimointi	Kommunikoinnin vähentäminen lisää järjestelmän suorituskykyä ja luotettavuutta. Vähentämistapoja ovat saman toiminnon tekeminen useissa solmuissa ja solmujen autonomisuuden lisääminen. Järjestelmän toimintakyky vikatilanteissa paranee.

Tehtävien allokointi eri prosessointiyksiköille voidaan hoitaa kolmella tavalla, keskitetysti, hajautetusti tai edellisten yhdistelmällä, hybridillä. Tavallisin vaihtoehto on hybridi. Allokointi voidaan toteuttaa joko staattisesti tai dynaamisesti, joskin dynaaminen allokointi on huomattavasti vaativampi suunniteltava ja toteutettava. Allokointitavat on esitetty kuvassa 10 [19].



Kuva 10. Prosessien allokointitapoja.

Allokoinnin vaikutuksia suorituskykyyn on arvioitu, ja erään havainnon mukaan suorituskyvyn paraneminen riippuu sekventiaalisen koodin ja prosessointiyksiköiden määrästä [19, 21] seuraavan kaavan [19] mukaan :

$$\text{Suorituskyvyn paraneminen} = \frac{1}{R + \frac{1 - R}{N}} \quad \text{, missä}$$

R = residuaalisen sekventiaalisen koodin määrä ja
N = prosessointiyksiköiden määrä.

Kaavalla laskien saadaan arvioita prosessointiyksiköiden vaikutuksesta suorituskyvyn paranemiseen. Taulukossa 7 [3] on esitetty muutamia kaavalla laskettuja teoreettisia maksimiarvoja. Lisäksi on otettava huomioon synkronoinnin tarve, ongelman luonne ja yhteisten resurssien käyttö, jotka luonnollisesti huonontavat taulukossa esitettyjä arvoja.

Taulukko 7. Hajautetun järjestelmän suorituskyvyn teoreettinen yläraja.

Prosessointiyksiköiden lukumäärä	Sekventiaalisen koodin osuus	Suorituskyvyn parannus
3	0,1	2,5
3	0,2	2,1
3	0,3	1,9
5	0,1	3,6
5	0,2	2,8
5	0,3	2,3
20	0,05	10,3
20	0,1	6,9
20	0,2	4,2

Koska hajautuksen lisääminen tehostaa tehtävän suoritusta, mutta toisaalta myös lisää kommunikoinnin tarvetta (*overhead*), on periaatteessa löydettävissä optimaalinen ratkaisu tehtävän suoritusajan minimoimiseksi eli suorituskyvyn maksimoimiseksi.

4. Toiminnallisuuden hajauttamista tukevat ratkaisut

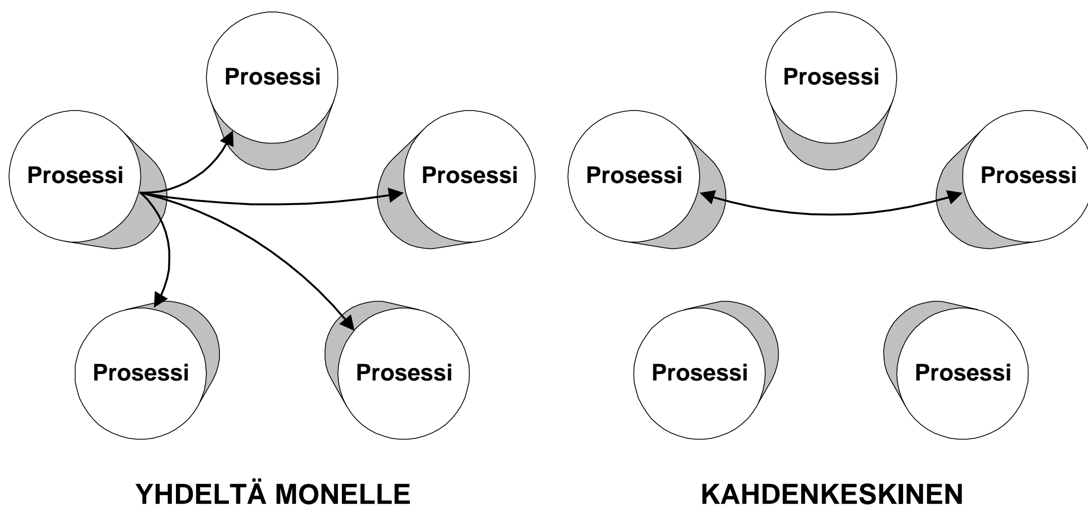
4.1 Kommunikointitavat

Prosessien välisen kommunikaation toteuttamiseksi on olemassa useita eri menetelmiä. Niillä kullakin on omat etunsa ja haittansa, joiden perusteella ne soveltuvat käytettäviksi erilaisissa järjestelmissä.

Seuraavaksi käsitellään kolmea asynkronista ja kolme synkronista kommunikointimenetelmää. Asynkronisia menetelmiä ovat viestipohjaisuus, asiakas/palvelin- ja tilaajamalli ja synkronisia etäkutsu eli RPC, oliopyyntövälittäjä eli ORB ja tietokantapohjainen menetelmä.

4.1.1 Viestipohjaisuus

Viestipohjaisuus tarkoittaa, että siirrettäväksi tarkoitettu tieto välitetään prosessista toiseen sanomassa [11]. Viestit voidaan lähettää asynkronisesti, eli lähettävän prosessin ei tarvitse odottaa vastausta lähettämäänsä viestiin. Viestipohjaiset järjestelmät ovat asiakas-palvelin -tyyppisiä järjestelmiä, joissa tietyntyyppiseksi muotoiltu viesti lähetetään kaikille vapaille vastaanottajille, jotka päättävät itse, vastaavatko saamaansa viestiin vai eivät [12]. Aluksi kommunikaatio voi olla yhdeltä monelle -tyyppistä (*one-to-many*), mutta vastauksen saapuessa viestin lähettäjä ja yksi tilaajista jatkavat usein kommunikointia kahdenkeskisenä (*one-to-one*) [12].



Kuva 11. Yhdeltä monelle ja kahdenkeskinen viestinvälitys.

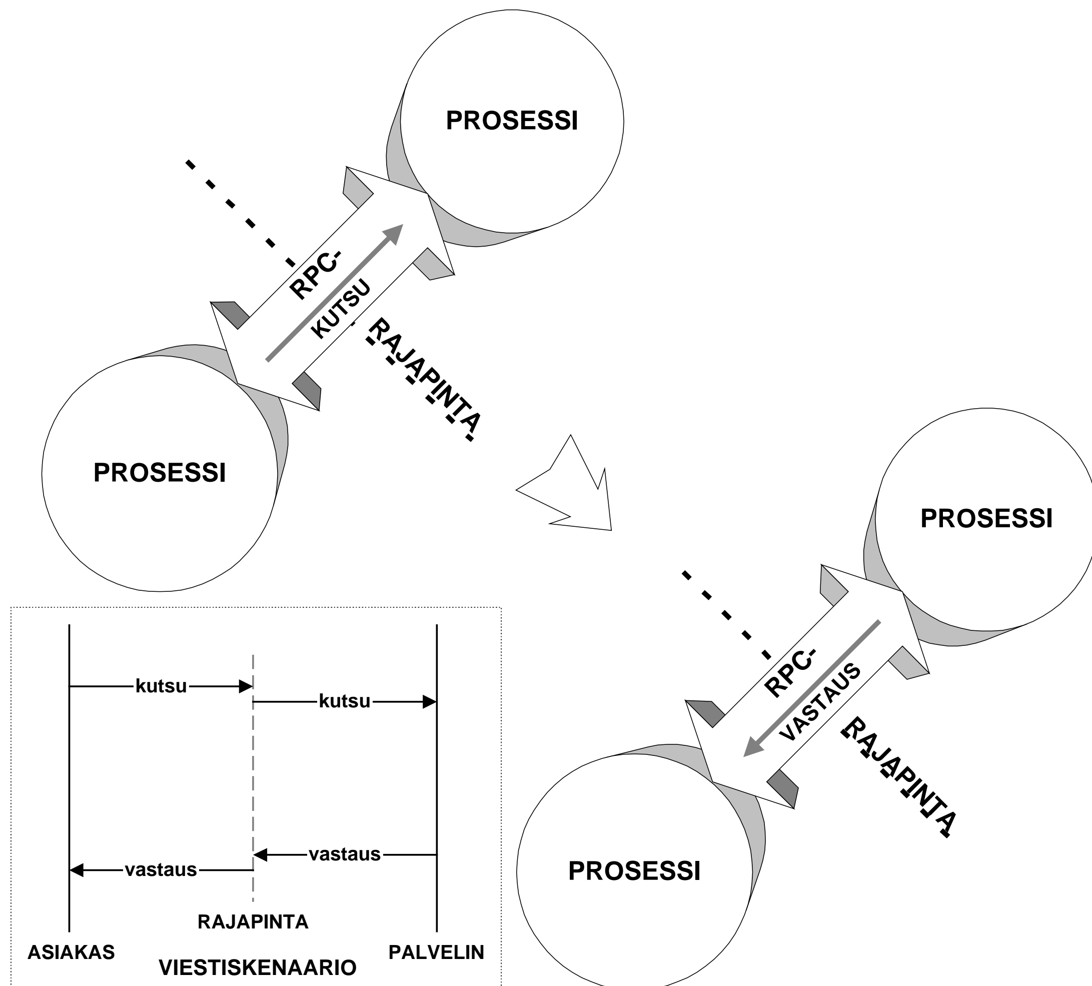
Viestipohjaisuuden etuna on automaattinen oletus, ettei käytetty tiedonsiirtomenetelmä ole luotettava, minkä vuoksi luotettavuuden ja etenkin sen puutteiden aiheuttamat tilanteet pyritään käsittelemään itsenäisesti [11].

Viestipohjaisuus antaa parhaan joustavuuden hajautuksen suunnittelulle, ja sitä käytetäänkin hajautetuissa tehtäväkriittisissä sovelluksissa [11]. Ongelmana on standardoinnin puuttuminen, eli välitettäville viesteille ei ole olemassa standardia, vaan viestin sisältö ja tyyppi vaihtelevat sovelluksesta toiseen.

Erikoistapaus viestipohjaisuudesta on viestijonoon (*message queue*) perustuva tiedonvälitys. Jono mahdollistaa prosessien välisen tiedon lähetyksen ja vastaanottamisen ilman suoraa yhteyttä [11, 22]. Prosessi yksinkertaisesti antaa viestit jonopalvelijalle määräten ainoastaan jonon, johon se haluaa viestin joutuvan. Jonopalvelu toimii välittäjänä ja sen toteutus on täysin kätketty sovelluksilta. Viestijono tarjoaa turvallisen tietovaraston ja on käyttökelpoisin tilanteissa, joissa prosessit eivät voi olla suoraan toisiinsa kytkettyjä. Viestijonojen heikkous on niiden käyttöönoton monimutkaisuus, samoin kuin huono suorituskyky.

4.1.2 RPC

RPC:n eli etäkutsun perusidea on laajentaa funktiokutsujen käsite hajautettuihin järjestelmiin [4, 22]. RPC:n semantiikka onkin lähes identtinen perinteisen funktiokutsun kanssa, eikä kutsuvan prosessin tarvitse edes tietää suoritetaanko pyydetty toimenpide samassa, vai jossakin muussa suoritusyksikössä. Yksinkertaisimmillaan kyseessä on kahden prosessin välinen kommunikaatio, jossa kutsuva prosessi jää odottamaan vastausta etäprosessilta (Kuva 12). Lähetetyssä kutsuviestissä välitetään tarvittavat argumentit, joiden mukaan etäprosessi toimii. Kun kutsun mukainen toiminto päättyy, tulokset palautetaan kutsuvalle prosessille ja kutsuja jatkaa toimintaansa täsmälleen samoin kuin jos suoritus olisi ollut paikallinen [4]. RPC on siis synkroninen kommunikointimenetelmä ja toimii siksi hyvin varsinkin pienissä ja yksinkertaisissa järjestelmissä, joissa asynkronista kommunikaatiota ei tarvita ja viestien välitys on kahdenkeskistä [11].



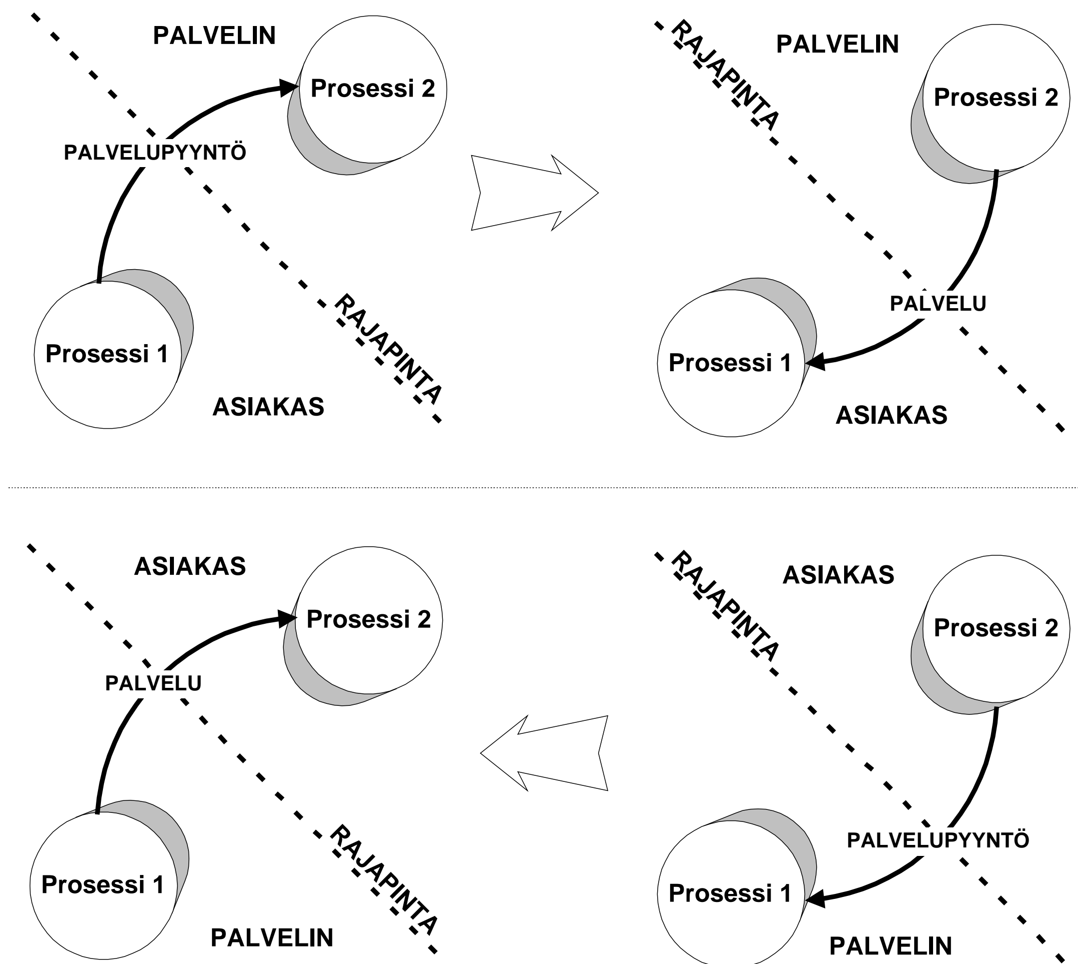
Kuva 12. Remote Procedure Call, RPC.

Perinteinen funktiokutsu tapahtuu prosessin sisäisesti kahden proseduurin välissä samassa järjestelmässä. RPC tapahtuu sen sijaan erillisen "asiakasprosessin" ja "palvelinprosessin" välillä, jotka voivat olla vaikka erilliset järjestelmät kytkettyinä samaan verkkoon. RPC-malli kuvaa siis kuinka yhteistyössä toimivat prosessit voivat kommunikoida ja koordinoida toimintoja keskenään. RPC:n yksinkertaisuus saattaa tehdä siitä kiinnostavan vaihtoehdon kommunikaatioksi, mutta joissakin tapauksissa sen suorituskyky ei ole riittävä. Suorituskyvyn optimoimiseksi täytyy tehdä valinta latenssin, eli funktion kutsun ja sen suorituksen välisen viiveen, ja suoritustehon (*throughput*) minimoinnin välillä. Ongelmallista on, että toisen pienentäminen kasvattaa toista [4], eikä molempia välttämättä saada riittävän pieniksi järjestelmän vaatimuksiin nähden.

4.1.3 Asiakas / Palvelin

Asiakas/palvelin (*client/server*) -arkkitehtuuri on rajapintapohjainen ohjelmistoarkkitehtuuri, jossa tietojenkäsittelytehtävät jaetaan palveluja hyödyntäviin osiin (client) ja palveluja tuottaviin osiin (server) ja näiden välinen tiedonvälitys tapahtuu selkeiden

rajapintojen kautta sanomapohjaisesti [6]. Olennaista arkkitehtuurissa on, että tehokas työnjako palvelua käyttävän prosessin ja palvelimen välillä haetaan tilannekohtaisesti. Kaksi prosessia toimii yhdessä tietyn tehtävän suorittamiseksi: asiakasprosessi pyytää tehtävän suoritettavaksi ja palvelinprosessi suorittaa sen. Palvelusuhde kestää vain palvelun suorittamisen ajan, ja toisessa tilanteessa roolit voivat vaihtua (kuva 13). Osallisina olevat prosessit voivat sijaita samassa tietokonejärjestelmässä tai ne voivat välittää toisilleen tietoa käyttäen hyväksi verkkopalveluja.

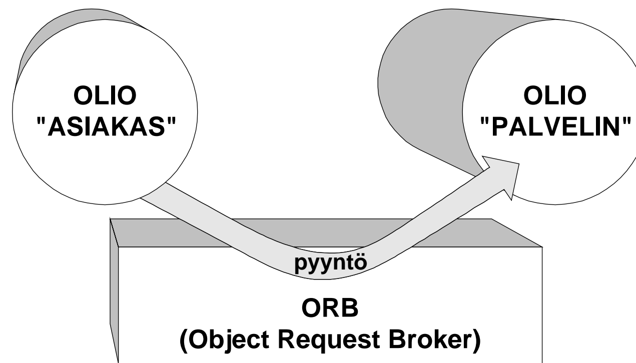


Kuva 13. Asiakas / palvelin -arkkitehtuuri

“Aitojen” asiakas/palvelin -järjestelmien ytimen muodostaa rajapintapohjainen ohjelmistoarkkitehtuuri, jossa eri osatehtävät eriytetään ja jossa sovellusosien välinen tiedonvälitys tapahtuu sanomapohjaisesti standardoituja rajapintoja hyödyntäen [6].

4.1.4 ORB

ORB (Object Request Broker) tarjoaa mekanismin, jolla hajautetut oliot voivat lähettää ja vastaanottaa pyyntöjä ja vasteita. Se antaa yhteistyömahdollisuuden sovelluksille, jotka toimivat eri koneissa hajautetuissa ympäristöissä. Oliomalli määrittää olion pyynnön ja siihen liittyvän vasteen. Pyyntö nimeää operaation ja antaa sille tarvittavat parametrit, jotka voivat olla myös tietyn olion määritteleviä [23]. ORB huolehtii siitä, että pyyntö menee vastaanottajalle käsiteltäväksi oikealle oliolle ja että kun pyyntö on käsitelty, se toimittaa saadun vastauksen pyynnön lähettäjäoliolle (kuva 14).



Kuva14.Object Request Broker, ORB.

ORBilla itsellään ei tarvitse olla kaikkia tietoja, joita tarvitaan pyyntöjen välittämiseen, vaan se voi käyttää hyväkseen käyttöjärjestelmän palveluja. ORBit on usein toteutettu RPC:n tai viestipohjaisuuden avulla, jolloin ne luonnollisesti kärsivät pohjana käyttämiensä menetelmien ongelmista [22]. ORBlta vaaditaan seuraavia ominaisuuksia OMG:n (Object Management Group) asettamien teknisten vaatimusten täyttämiseksi [23]:

- Pyynnön ohjaaminen oikeaan paikkaan olion nimen tai tunnisteen perusteella.

- Oikean metodin käyttäminen. Oliomalli ei vaadi, että pyyntö menisi millekään tietylle oliolle, vaan se voi käyttää jotain muuta metodia, joka sitten käyttää tarvittavaa metodia.

- Parametrien koodaus. ORB:in pitää pystyä muodostamaan arvot vastaanottajan ymmärtämään muotoon.

- Pyyntöjen toimittaminen perille.

- Synkronointi ja useiden samanaikaisten pyyntöjen käsittely.

- Aktivointi eli olion saattaminen sellaiseen tilaan, että sitä voidaan hyödyntää.

- Poikkeustilanteiden käsittely.
- Suojausmekanismit, jotka varmistavat pyyntöjen ja tiedon perillemenon.

ORB on suunniteltu käytettäväksi oliopohjaisissa järjestelmissä, joissa olioiden käyttö on ainoa ratkaisu [11]. ORB:issa käytetään rajapinnan määrittelykieltä (*IDL, Interface Definition Language*) olioiden välisten liityntöjen kuvaamiseen. ORB toimii parhaiten, kun olioiden väliset rajapinnat eivät muutu usein. Yleensä ORB:it ovat synkronisia ja toimivat "pisteestä pisteeseen".

4.1.5 Tilaajamalli

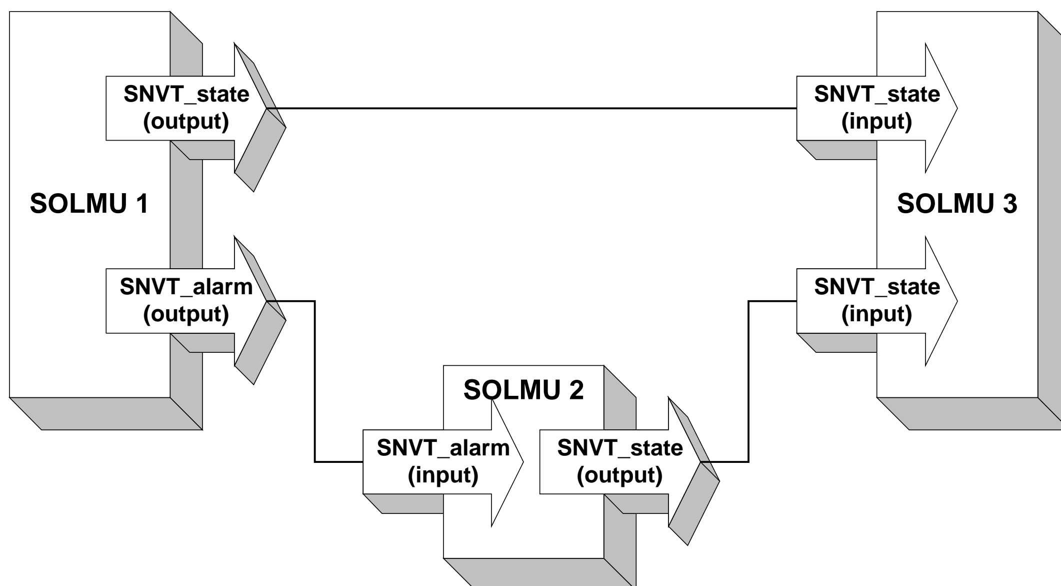
Tilaajamalli perustuu viestinvälitykseen, jossa tieto lähetetään siitä kiinnostuneille [11]. Mallin periaate on se, että tietyt prosessit ilmoittautuvat tietyn tiedon tilaajiksi, ja toiset puolestaan julkaisevat tietoja. Jos prosessi on ilmoittautunut tilaajaksi tiedolle, jota jokin toinen prosessi julkaisee, se saa automaattisesti tiedon aina tiedon päivittyessä. Julkaisijat (*publisher*) toimivat tiedon lähteinä ja tilaajat (*subscriber*) tiedon käyttäjinä. Julkaisija voi olla myös tilaaja, ja päinvastoin [7, 24]. Tietoyksiköillä on tunniste, jonka perusteella tietoa osataan antaa ja hakea. Samalle tunnisteelle voi olla useita julkaisijoita ja useita tilaajia. Julkaisijan ei tarvitse tietää, kuinka monta tilaajaa tiedolle on, eikä tilaajan vastaavasti tarvitse tietää, kuinka monta julkaisijaa tiedolla on. Ainoa asia, josta tilaajien ja julkaisijoiden täytyy sopia, on tunniste, jota tiedon yhteydessä käytetään.

Tilaajamalli tarjoaa prosessien sijainnin läpinäkyvyyden, eli tiedon tuottajan ja tilaajan ei tarvitse olla tietoisia toistensa sijainnista [11]. Tiedon kuluttajan ei myöskään tarvitse tietää tiedon tuottajaa ja päinvastoin [7, 24]. Kommunikaatioprotokolla on niinikään näkymättömissä sovellukselle.

Tilaajamalli soveltuu parhaiten korkeasti hajautettuihin järjestelmiin, joissa virhesietoisuus ja suorituskyky ovat tärkeitä [11]. Se toimii hyvin tilanteissa, joissa tietoa tulee siirtää nopeasti useille prosesseille. Se ei sovellu puolestaan järjestelmiin, joissa järjestelmän osia voidaan poistaa verkosta pitkiksi ajanjaksoiksi tai joissa siirrettävä tieto kulkee usean verkkosegmentin kautta ja se on talletettava niissä kaikissa ennen siirtymistä seuraavaan [11].

Esimerkkinä tilaajamallista käsitellään LON-verkkoa (Local Operating Network), ja keskitetään huomio sen verkkomuuttujiin. LON:issa tunnistetta, jota joko julkaistaan tai tilataan, kutsutaan verkkomuuttujaksi (*network variable*). Verkkomuuttujan arvo on se LON-verkossa "näkyvä" arvo, jolla on merkitystä vain niille verkkosolmuille, joiden sovellusohjelmassa kyseinen muuttuja on määritelty eli jotka ovat ilmoittautuneet kyseisen tiedon tilaajiksi. Kuvassa 15 on esitetty esimerkki verkkomuuttujien käytöstä ja sidonnasta LON:issa. Kuvassa olevat *SNVT_state* ja *SNVT_alarm* ovat Echelon-yhtiön

määrittämiä verkkomuuttujien standardityyppejä. Aina kun verkkomuuttuja päivittyy, tilaajat saavat muuttujan arvon verkosta ja toimivat sen mukaan sovellusohjelmassa määrätyllä tavalla.



Kuva 15. Esimerkki LON-verkkomuuttujista.

Myös ORB:ien toteutukset sisältävät tilaajamallin, joka on toteutettu osana tapahtumahallintaa.

4.1.6 Tietokantapohjaiset järjestelmät

Tietokantapohjaisuus tarkoittaa tietokantayhdyskäytävien (*database gateway*) ja tietokantojen käyttöä kommunikointiin [11]. Asiakasohjelma antaa SQL (Structured Query Language) -pyynnön yhdyskäytävälle, joka puolestaan toimittaa pyynnön kohdetietokantaan. Yhdyskäytävä tulkkaa pyynnön kohdetietokannan ymmärtämään muotoon. Tietokantapohjainen järjestelmä viittaa synkroniseen, pisteestä pisteeseen (*point-to-point*) -kommunikointiin. Tämä lähestymistapa ei toimi kunnolla suurta suorituskykyä vaativissa sovelluksissa, koska tietokanta saavuttaa nopeasti tukahtumispisteen useiden sovelluksien yrittäessä kommunikoida samanaikaisesti [11]. Järjestelmä ei myöskään toimi verkkoyhteyksien pettäessä, sillä yhteyden katketessa tietokantaan kommunikointi ei ole mahdollista.

4.2 Sulautettuihin järjestelmiin soveltuvat liityntätavat

4.2.1 CORBA

OMG on valinnut standardirajapinnan ORB:lle, ja se on nimetty CORBA:ksi (Common Object Request Broker Architecture). CORBA:n päätavoite on integrointituki monille oliojärjestelmävaihtoehdoille, ja joustavuus on siksi valittu sen perusominaisuudeksi. CORBA:n ydin on sen oliomalli [12]. Oliomalli tukee suoraan asiakkaan ja palvelimen yhteistyökonseptia, eli tilanteesta riippuen asiakkaan ja palvelimen roolit vaihtelevat. Oliomalli sisältää käsitteet olion luomisesta, ainutlaatuisesta tunnisteesta, operaatioista, tyypeistä, olioiden toteuttamisesta, metodeista, suoritusyksiköistä ja olioaktivoinneista [12].

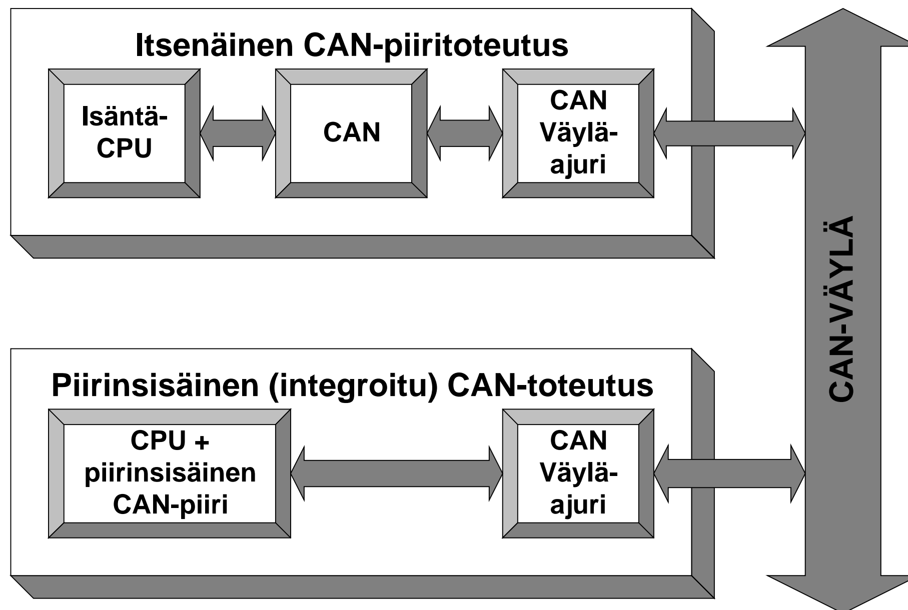
CORBA ei sellaisenaan ole sulautettuun ympäristöön sopiva, mutta on olemassa kaupallisia CORBA-toteutuksia, joissa sulautetun järjeselmän suorituskyky- ja muistirajoitusvaatimukset on otettu huomioon. Yksi esimerkki tällaisista toteutuksista on Ionan RT-ORBIX.

Perusluonteeltaan malli keskittyy asiakasolioihin ja niiden palvelupyyntöihin palvelujen toteuttamismekanismin asemesta. Palvelujen toteuttamismekanismit on jätetty niin avoimiksi kuin mahdollista joustavuuden säilyttämiseksi. Palvelupyynnöt perustuvat operaatiotunnisteisiin (*operation signatures*), jotka määritellään osana rajapintatyyppejä [12]. Jokainen rajapintatyyppi sisältää kokoelman operaatioita tai palveluja, joita asiakas voi pyytää, toteutusten ollessa täysin kätkettyjä asiakkaalta. CORBA:n toinen tärkeä piirre on sen sisältämä rajapintojen kuvauskieli IDL, jonka avulla sovellusten välinen kommunikointi tapahtuu ORB:n ohjelmistoväylässä.

4.2.2 CAN

CAN (Controller Area Network) on sarjaväyläjärjestelmä, joka on erityisesti suunniteltu “älykkäiden” laitteiden verkottamiseen. Alunperin se kehitettiin ajoneuvoihin, mutta sen käyttö on lisääntynyt ja laajentunut teollisuuteen ja rakennusautomaatioon. CAN-protokolla on avoin standardi, ja tällä hetkellä protokollapiireillä on ainakin kahdeksan eri valmistajaa.

Protokolla on toteutettu joko erillisenä CAN-ohjainpiirinä, joka on rajapinnan kautta kytketty ulkoiseen prosessoriin, tai CAN-liitännäislaitteena, jossa CAN-piiri on integroitu isäntäprosessoriin. Protokollan rakenne on esitetty kuvassa 16 [25]. CAN-väyläajuri (*CAN bus driver*) tai lähetin-vastaanotin (*transceiver*, *trans*mitter re*ceiver*) huolehtii väyläjännitteen ylläpidosta ja väyläsignaalien muuntamisesta CAN-liitännäislaitteen hyödyntämään muotoon [25].



Kuva 16. CAN-protokollan rakenne.

CAN-protokollapiirien tarjoamia peruspalveluita ovat enintään kahdeksan tavun mittaisen sanoman yleinen lähetys, lähetyspyynnön lähetys, sanomien vastaanotto ja lähetyspyyntöjen vastaanotto. Muita mahdollisia kommunikointipalveluja ovat lohkosiirto, tapahtumien signalointi, kuittauksellisen sanoman lähettäminen, multipleksatut sanomat ja verkonhallintapalvelut [26].

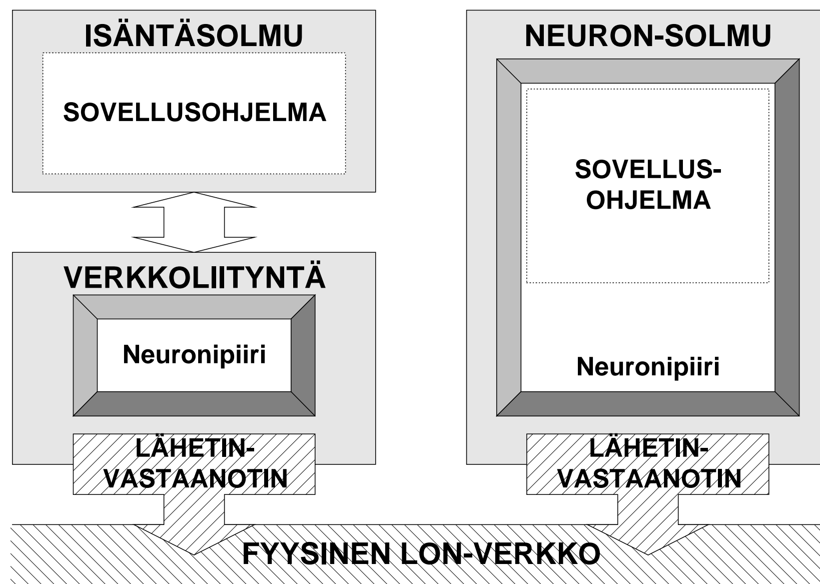
CAN-protokolla määrittelee väylään lähetettävien viestien muodon, lähetettyjen bittien ajoituksen, sovitteluprioriteetit (*arbitration priorities*) eli samanaikaisesti lähetettyjen viestien lähetysjärjestyksen, ja virheiden havaitsemisen. Protokolla tukee kahta erityyppistä kommunikaatioviestioliota (*communications message object*), lähetystä ja vastaanottoa [25]. Viestioliot toimivat eräänlaisten CANin tarjoamien postilaatikkojen välityksellä. Postilaatikot toimivat tilaajaperiaatteella eli ovat verkkomuuttujia, joiden arvoa tai tässä tapauksessa sisältöä jokin prosessi muuttaa ja jokin toinen lukee.

CANissä suurin tiedonsiirtonopeus on 1 Mbit/s, jolloin verkon pituus saa olla korkeintaan 40 metriä. Pidemmillä etäisyyksillä tiedonsiirtonopeutta täytyy laskea. Esimerkiksi nopeuden ollessa 125 kbit/s päästään 500 metriin saakka ja 50 kbit/s kilometriin saakka.

4.2.3 LON

Echelon Corporationin LONin (Local Operating Network), peruskomponentti on neuronipiiriksi (*neuron chip*) kutsuttu mikroprosessori, joka sisältää kolme eri prosessoria. Prosessorit ovat sovellusprosessori (*Application processor*), verkkoprosessori (*Network processor*) ja mediaohjainprosessori (*Media Access Control processor, MAC*) [27].

Neuronipiiri tarjoaa rajapinnat ja palvelut, joita tarvitaan toisten neuronipiirien kanssa kommunikointiin, sekä sisältää pienen sisäisen muistin. Muistiin voidaan ladata ja siitä ajaa pieniä sovellusohjelmia. Sovellusohjelmat kehitetään *neuron C* -kielellä, joka on laajennettu ANSI C:stä lisäämällä siihen I/O:n ja tapahtumien käsittelyn tuki, sanomanvälitys sekä hajautetut tietorakenteet. Piiri sisältää myös rajapinnan ulkoisen, sovellusta ajavan prosessorin liittämistä varten sekä perus-I/O-valmiudet. Kullakin neuronipiirillä on ainutlaatuinen tunniste, *neuron id*, jonka avulla piiri voidaan aina tunnistaa. Neuronpiirin fyysiseen verkkoon liittämiseen tarvitaan toinen viestien lähetyksen ja vastaanottamisen hoitava komponentti, lähetin-vastaanotin (*transceiver*). LON- verkon rakenne on esitetty kuvassa 17.

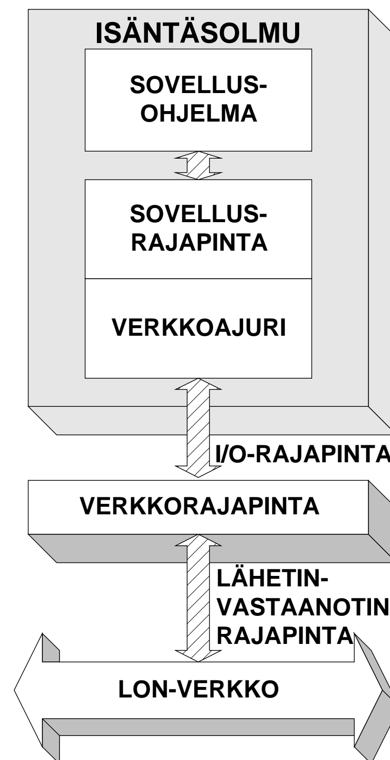


Kuva 17. LON-verkon komponentit.

Fyysisenä verkkona voidaan käyttää useita erityyppisiä medioita, kuten esimerkiksi sähköverkkoa, kierrettyä parikaapelia, valokaapelia ja radiotaajuuksia. Verkkoa vaihdettaessa ainoa vaadittu muutos on lähetin-vastaanottimen vaihtaminen, sillä verkon tyypillä ei ole neuronipiirille merkitystä. LONissa tiedonsiirtonopeus riippuu fyysisen verkon tyypistä ja käytetystä lähetin-vastaanottimesta. Parikaapeliverkon ja valokaapelin kyseessä ollessa tiedonsiirtonopeus on 78,1 kbit/s - 1,25 Mbit/s, radiotaajuuksilla 4,9 kbit/s ja sähköverkossa 625 kbit/s [3]. LON-verkko jakaantuu alueisiin (*domain*), alaverkkoihin (*subnet*) ja solmuihin (*node*) [3, 26]. Yhdellä alueella voi olla 255 alaverkkoa, joissa kussakin korkeintaan 127 solmua. Tästä yhden alueen maksimikooksi saadaan 32 385 solmua.

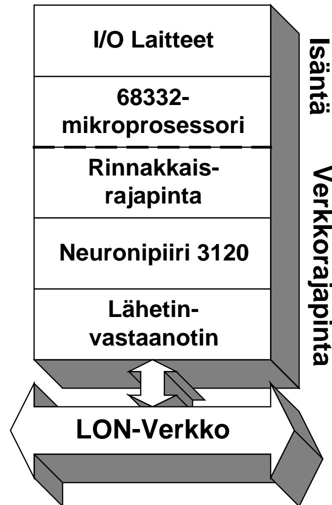
LONissa on valmiudet kahdentyyppisten verkkosolmujen liittämiseen. Kuten edellä mainittiin, neuronipiirillä on rajapinta ulkoisen sovellusohjelmaa ajavan prosessorin liittämistä varten tavanomaisen neuronipiiripohjaisen verkkosolmun lisäksi. Tätä rajapintaa hyväksikäyttäviä solmuja kutsutaan yleisesti isäntäsolmuiksi (*host node*), ja niissä

ajettavia sovelluksia isäntäsovelluksiksi (*host application*). Tällaisia isäntäsolmuja ovat esimerkiksi sovellusohjelmaa suorittavat PC-tietokoneet, jotka on kytketty LON-liitynnän tarjoavan PCLTA-verkkoliityntäkortin kautta LON-verkkoon. Isäntäsolmun yleinen arkkitehtuuri [28] on esitetty kuvassa 18.



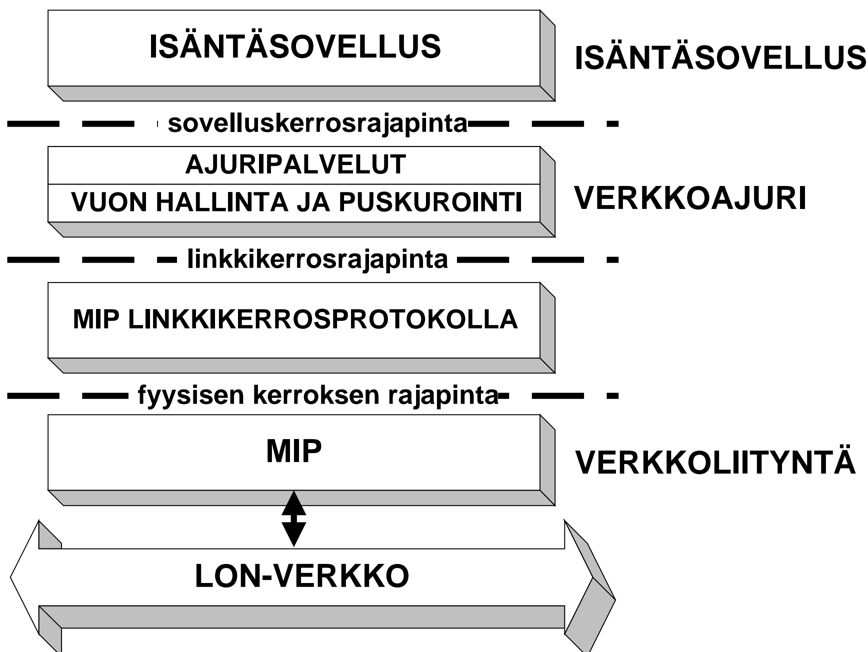
Kuva 18. Isäntäsolmun arkkitehtuuri.

Verkkoliityntä koostuu fyysisesti neuronipiiristä ja siihen kytketystä elektroniikasta, joka mahdollistaa liittämisen haluttuun ulkoiseen suorittimeen kuten esimerkiksi PC-tietokoneeseen. Fyysinen liittyminen verkkoon tapahtuu lähetin-vastaanottimen kautta, kuten tavanomaisten solmujen tapauksessakin. Esimerkki mahdollisesta isäntäsolmun rakenteesta on esitetty kuvassa 19 [28].



Kuva 19. Isäntäsolmun esimerkkirakenne.

Verkkoliityntä koostuu useasta protokollakerroksesta, jotka yhdessä toimivat isäntäsovellukselle ja LON-verkolle näkymättömästi järjestäen tarvittavat palvelut. Verkkoliitynnän protokollakerrokset on esitetty kuvassa 20 [28].



Kuva 20. Verkkoliitynnän protokollakerrokset.

5. Reaaliaikajärjestelmän hajautusmalli

5.1 Hajautusmallin kehittäminen

Ohjelmistoarkkitehtuurilla on suuri merkitys ohjelmiston joustavuudelle. Sovellusalueen asettamat vaatimukset muodostavat arkkitehtuurin pääpiirteet, ja tuoteperheeseen sisällytettävät ohjelmistovariaatiot asettavat vaatimukset tarpeellisille konfigurointimenetelmille [1, 9]. Jotta vaatimukset saataisiin selville, on suoritettava sovellusalueanalyysi. Sovellusalueanalyysin tavoite on tunnistaa itsenäisiä toimintoja suorittavat autonomiset ohjelmistokomponentit ja komponenttien väliset riippuvuudet [9]. Tällä tarkoitetaan sitä, että kunkin eritellyn komponentin perusfunktion tulisi olla selvä, sen valinnaisten ominaisuuksien tiedossa ja sen rajapinnat muiden komponenttien kanssa riittävällä tarkkuudella määritellyt. Näin eritellyt komponentit asettavat myös rajoitukset sovellettavalle ohjelmistoarkkitehtuurille [9].

Ohjelmiston muunneltavuus on otettava huomioon jo ohjelmiston suunnittelussa. Ominaisuuspohjainen konfigurointi voi perustua komponenttivalintaan ja/tai komponenttikonfigurointiin [9]. Komponenttivalintainen konfigurointimenetelmä tarkoittaa sitä, että järjestelmä kootaan tuoteominaisuuksien perusteella valituista komponenteista. Komponenttikonfigurointi puolestaan tarkoittaa geneeristen komponenttien parametrien asettamista siten että haluttu toiminta tai ominaisuus saavutetaan. Myös näiden yhdistelemi-nen on mahdollista halutun konfiguroitavuuden saavuttamiseksi.

5.1.1 Hajauttaminen ja muunneltavuus

Hajauttaminen ei saisi näkyä lainkaan toiminnallisille ohjelmakomponenteille. Hajautusalustan tulisi olla pelkästään mekanismi, joka mahdollistaa autonomisten komponenttien vapaan sijoittelun toimintaympäristöön, suorittaa tiedonsiirron kommunikoivien komponenttien välillä ja ohjaa toimintaa reitittämällä implisiittisiä pyyntöjä niiden eksplisiittisille vastaanottajille.

Ohjelmistoväylälle ja käytettävälle ohjelmistoarkkitehtuurille asetettavat vaatimukset on esitetty taulukossa 8.

Taulukko 8. Ohjelmistoväylälle ja arkkitehtuurille asetetut vaatimukset.

	Vaatimus	Ratkaisu
1	Uusien sovelluskomponenttien eli osajärjestelmien lisäämisen järjestelmään täytyy onnistua niin pienellä työllä kuin mahdollista.	Tähän tavoitteeseen päästään toteuttamalla sovelluskomponentit omatoimisina prosesseina, jotka suorittavat omia tehtäviään jättäen järjestelmätason päätökset ja toimenpiteet ohjelmistoväylälle. Näin toteutettuja prosesseja voidaan myös kutsua *agenteiksi*.
2	Sovellusten täytyy olla sijainti-riippumattomia.	Tämä onnistuu, kun osajärjestelmät käyttävät välittäjää saavuttaakseen keskinäisen yhteentoimivuuden ja niillä on standardit rajapintamäärittelyt liitynnöilleen alustaan. Rajapintamäärittelyt toteutetaan rajapintakomponenteilla ja ohjelmistoväylän palvelurajapinnalla.
3	Laitteistomuutokset eivät saa vaikuttaa sovelluskomponent-teihin.	Ohjelmistoväylän tehtävä on kätkeä laitteisto täysin sovelluksen kompo-nenteilta, jolloin tavoite saavutetaan.
4	Sovellusten täytyy olla löyhästi kytkettyjä (*loosely coupled*).	Ohjelmistoväylä toimii sovellusriippuvana välittäjänä, joka tarjoaa peruspalvelut sovelluskomponenttien väliseen kommunikointiin.
5	Samaa ohjelmistoalustaa täytyy voida käyttää kaikkien ohjelmistovariaatioiden kanssa.	Tähän pyritään käyttämällä kerrostettua ohjelmistoarkkitehtuuria, joka myös auttaa tuoteperheperiaatteen saavuttamisessa. Kerrostettu arkkitehtuuri voi perustua tietoabstraktiotasoihin, toiminnalliseen hajautukseen tai vastuualuejakoon.

5.2 Hajautusmallin sisältö

Tässä työssä päädyttiin hajautusalustamalliin, joka pohjautuu prosessipohjaiseen sovellusalueriippuvista osajärjestelmäkomponenteista koostuvaan ohjelmistoväylä-konseptiin. Tärkeinä pidettiin komponenttipohjaisuuden lisäksi tietokeskeisyyttä, rajapintojen standardimaisuutta ja järjestelmän konfiguroitavuutta.

Ohjelmistoarkkitehtuuri ja kommunikointisäännöt muodostavat vakaan muuttumattoman pohjan järjestelmään, ja konfigurointi on sallittu vain ohjelmistoväylälle asetettuja sääntöjä noudattaen. Nämä yhdessä konfigurointisääntöjen kanssa muodostavat järjestelmän ohjelmistoalustan. Järjestelmää voidaan siis laajentaa lisäämällä uusia komponentteja, jotka käyttävät samoja kommunikointisääntöjä kuin jo olemassaolevat komponentit. Muunneltavuus on saavutettu alustaan sisällytettyjen konfigurointimekanismien avulla. Uusia ominaisuuksia voidaan lisätä muuttamalla osajärjestelmien sisäisiä funktioita, eli joko lisäämällä tai konfiguroimalla niiden sisäisiä komponentteja.

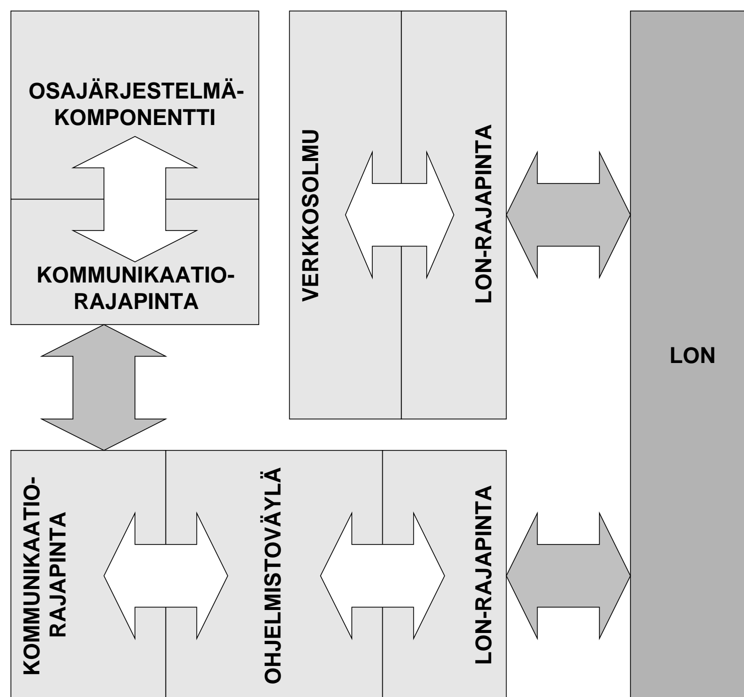
Osajärjestelmien sisäinen kommunikointi perustuu palvelupyyntöperiaatteeseen (*service-on-request*) eli osajärjestelmän halutessa tietoa jostain se jättää pyynnön ja jää odottamaan vastausta, mikäli sitä ei ole heti saatavilla. Kommunikointi on järjestetty käyttöjärjestelmätasolla tuettujen viestijonojen avulla. Jokainen osajärjestelmä koostuu kommunikointikomponentista ja yhdestä tai useammasta sovellusagentista.

Sovelluskomponenttien konfiguroinnin hallinta on yksinkertainen ja perustuu tiedon konfigurointiin. Tieto konfiguraatioista, jotka ovat tuoteperheen sisällä mahdollisia, on

talletettu ohjelmistoväylään, ja aktiiviset järjestelmän osat määritellään sen alustamisen yhteydessä.

Työn esimerkkinä olleen palautusautomaatin ohjelmistoarkkitehtuuri suunniteltiin ja toteutettiin aikaisemmin laaditun spesifikaation [1] pohjalta. Keskeisenä suunnittelu- ja ositusperiaatteena käytettiin sovellusaluekohtaista osajärjestelmäjaottelua ja ohjelmistoväyläkonseptia.

Sovellusaluekohtainen osajärjestelmäjaottelu noudattaa määrittelyvaiheen aikana tehdyn sovellusalueanalyysin tuloksia. Arkkitehtuuria suunniteltaessa kiinnitettiin huomiota myös syntyvien osajärjestelmien uudelleenkäytettävyyteen. Sovellusalueanalyysin ja arkkitehtuurisuunnittelun tuloksena määritellyt osajärjestelmät ovat mahdollisimman itsenäisiä ja tiettyä rajattua tehtävää suorittavia kokonaisuuksia. Uudelleenkäytettävyyttä parannettiin kiinnittämällä huomio osajärjestelmien välisiin rajapintoihin (Kuva 21), jolloin osajärjestelmä on korvattavissa uudella, mikäli korvaavan ja korvattavan osajärjestelmän rajapinnat ovat samanlaiset. Osajärjestelmien sisällä uudelleenkäyttöä ja joustavuutta voidaan tukea mm. ohjelmiston käyttäytymisen konfigurointiparametrien avulla.



Kuva 21. Käytetty rajapintaperiaate.

Toinen keskeinen suunnitteluperiaate, ohjelmistoväyläkonsepti, tukee osajärjestelmien kommunikointia niiden sijainnista riippumatta. Ohjelmistoväylä toimii hajautusalustana, joka mahdollistaa palautusautomaatin osajärjestelmien joustavan hajauttamisen verkkoympäristössä. Kommunikointi- ja hajautustuen lisäksi ohjelmistoväylään "upotettiin" myös järjestelmän konfigurointia ja tilatietoa sisältävät tietorakenteet. Ohjelmistoväylää

voidaan siksi pitää sovellusaluekohtaisena hajautusalustana, mikä lisää väyläkonseptin käyttökelpoisuutta merkittävästi. Väylä sisältää sekä hajautuksen tuen että sovellusalueelle ominaiset yhteiset ja yleiset palvelut kaikkien osajärjestelmien käyttöön.

5.3 Hajautusmallin joustavuus ja muut ominaisuudet

Edellisessä kappaleessa mainitut työssä käytetyt suunnitteluperiaatteet lisäävät järjestelmän joustavuutta. Niiden haittapuolena on suorituskyvyn heikkeneminen joustavuuden lisääntyessä.

Ohjelmistoväylä tarjoaa useita hyödyllisiä ominaisuuksia [9], kun ohjelmistokehitystä tarkastellaan tuoteperhenäkökulmasta. Ominaisuudet on esitelty taulukossa 9.

Taulukko 9. Ohjelmistoväylän ominaisuudet.

_	Ominaisuus
1	Järjestelmää on helppo laajentaa uusia sovelluksia lisäämällä.
2	Sovelluks___et ovat sijaintiriippumattomia läpinäkyvästi järjestettyjen kommunikaatiomekanismien vuoksi.
3	Kuhunkin solmuun sijoitettu keskitetty kommunikointikomponentti yksinkertaistaa liityntöjä muihin sovelluksiin.
4	Kevytrakenteinen kommunikaatiorajapinta on sopiva vaativiin reaaliaikasovelluksiin, joissa on rajoitetut muisti- ja muut resurssit.
5	Toteutusriippuvat osat on kätketty ohjelmistoväylään eivätkä ole lainkaan näkyvissä sovelluskomponenteille.
6	Muunneltavuus on hyvä, koska konfigurointimekanismit on keskitetty yhteen paikkaan, ohjelmistoväylään.

Joustavat hajautetut ohjausjärjestelmät tarvitsevat tukea, jotta muunneltavuus, laajennettavuus ja skaalattavuus saavutettaisiin. Tämä on mahdollista kehittämällä konfiguroituva ohjelmistoväylä ja geneerinen rajapintatekniikka sovelluksille [9]. Ohjelmistoväylä tukee hajautusta läpinäkyvällä kommunikoinnilla ja ominaisuuspohjaisella konfiguroinnilla. Ohjelmistoväylä tarjoaa älykkään viestinvälityksen ja konfigurointituen sovelluskomponenteille ja fyysiselle kommunikaatioverkolle. Sovellusalueen vaatimukset on täytetty ilman ylimääräisiä, tuoteperheeseen soveltumattomia piirteitä [9].

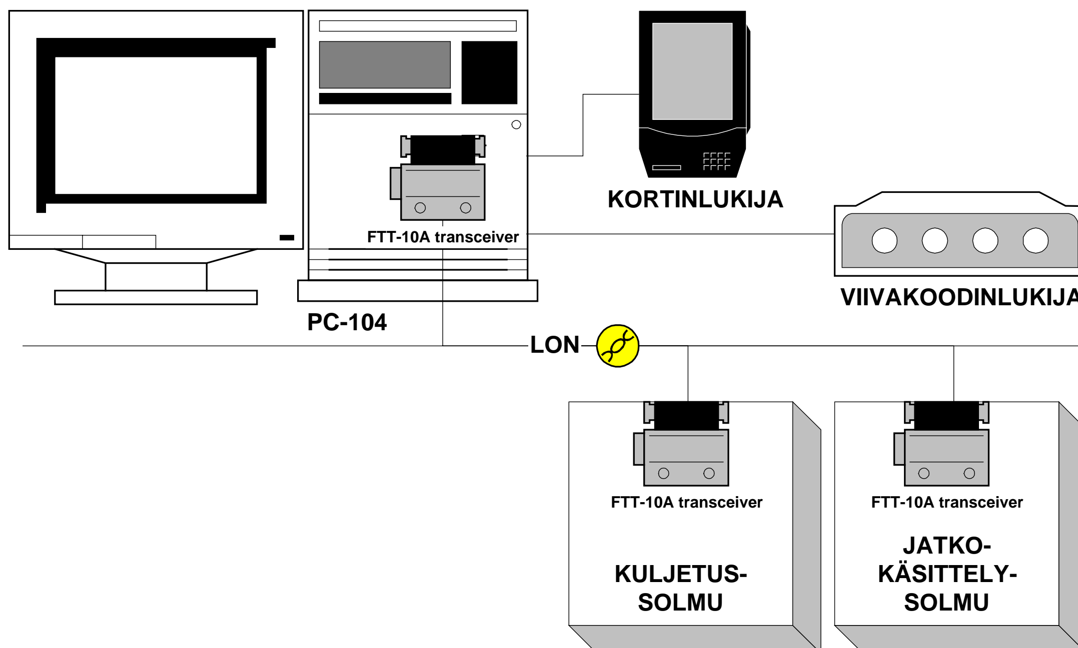
6. Koejärjestelmä

6.1 Laitteisto

VTT Elektroniikassa toteutettiin vuosina 1996–1997 Dynamo-niminen projekti, jossa selvitettiin sulautettuihin reaaliaikajärjestelmiin soveltuvia joustavia ja skaalattavia ohjelmistoarkkitehtuureja.

Dynamo-projektissa pilotoitiin palautusautomaattien joustavaa hajautusarkkitehtuuria Halton System Oy:lle. Toteutetussa pilotissa selvitettiin, millaisilla ohjelmistoarkkitehtuuriratkaisuilla voitaisiin toteuttaa helposti muunneltavia ja hajautettavia palautusautomaattien ohjelmistoja. Käytännön hajautusratkaisuna kokeiltiin PC-104- ja LON-verkon solmuista koostuvaa koejärjestelmää.

Koejärjestelmä eli pilotti toteutettiin kaupallisesti saatavilla komponenteilla niin pitkälle kuin mahdollista. Tavoitteena oli suunnitella joustava ohjelmistoarkkitehtuuri ja arvioida suunnitellun arkkitehtuurin käyttökelpoisuutta yhdessä kaupallisesti hankittujen valmisohjelmistojen ja -korttien sekä hajautusalustana toimivan LON-verkon kanssa. Samalla tavoitteena oli kerätä käytännön kokemuksia lähestymistavasta, jossa hyödynnetään mahdollisimman paljon kaupallisia valmiskomponentteja niin ohjelmistoissa kuin laitteistoissakin. Pilotissa käytetty laitteisto on esitetty kuvassa 22.



Kuva 22. Pilot-laitteisto.

Pilotin verkkoratkaisun olennainen osa oli Echelonin LON-verkon soveltuvuuden testaaminen hajautusalustana. Verkon kehitystyökaluna käytettiin LonBuilderia [29]. LonBuilderin valintaperuste oli sen monipuolisuus sekä yksittäisten solmujen että koko verkon kehittämisessä ja konfiguroimisessa. Verkkosolmuja päätettiin tehdä kaksi, yksi

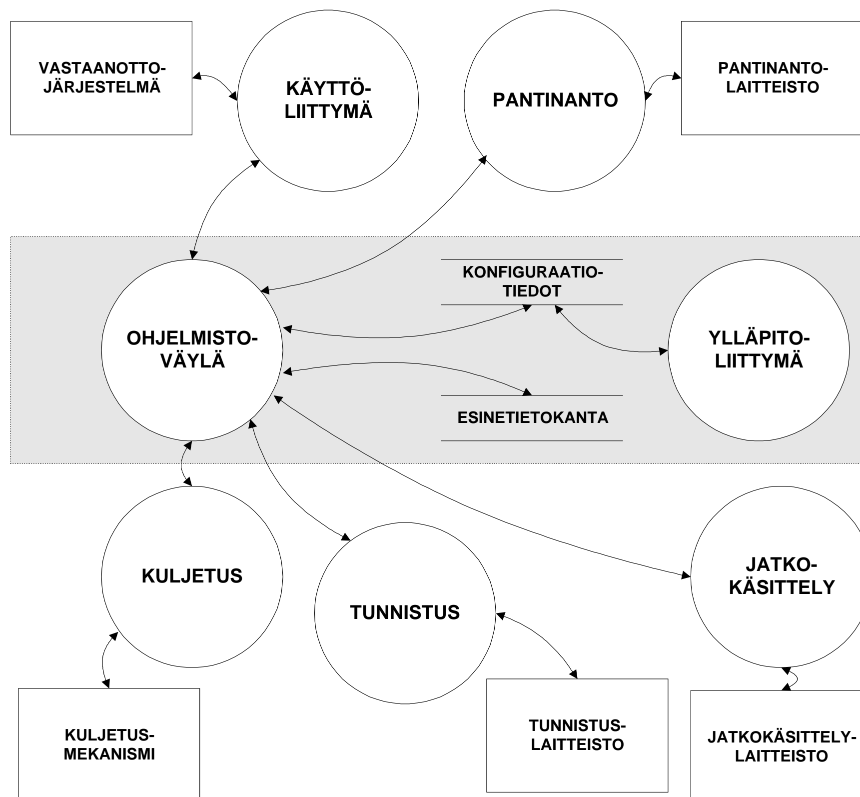
kuljetusosajärjestelmälle ja yksi jatkokäsittelijälle, sekä käyttää verkkoliityntäkorttia tietokonejärjestelmän liittämiseen isäntäsolmuksi verkkoon.

Tietokoneeksi valittiin PC-104-kortti-PC (Ampro), koska se kompaktin kokonsa ja I/O-, verkko- ja näyttöliityntöjensä vuoksi tarjoaa hyvän alustan toteuttaa sulautettuja sovelluksia koviin olosuhteisiin, mutta samalla mahdollistaa kaupallisten valmisohjelmistojen laajamittaisen hyödyntämisen. PC-104-tietokoneen tehtävänä pilotissa oli palautusjärjestelmän toiminnan koordinointi ja keskeisen ohjelmiston suorittaminen sekä LON-verkon solmujen ohjaus LON-verkkoliityntäkortin välityksellä. Kortti-PC:hen haluttiin myös liitettävän kortinlukija pantinantoa varten sekä viivakoodinlukija palautusjärjestelmän vaunujen tunnistusta varten.

Tietokonejärjestelmälle asetettujen vaatimusten perusteella käyttöjärjestelmäksi valittiin QNX ja siihen liitetty graafinen käyttöliittymä toteutettiin Photon-käyttöliittymän sovelluskehittimellä [30], jotka molemmat toimivat PC-ympäristössä.

6.2 Palautusautomaatin ohjelmistoarkkitehtuurin suunnittelu

Palautusjärjestelmän osajärjestelmäpohjainen ja ohjelmistoväylään perustuva ohjelmistoarkkitehtuuri on esitetty kuvassa 23.



Kuva 23. Palautusautomaatin ohjelmistoarkkitehtuuri.

Ohjelmiston ydin on ohjelmistoväylä, joka toimii älykkäänä viestien välittäjänä PC-104-solmussa ja LON-verkkosolmuissa sijaitsevien osajärjestelmien välillä.

6.2.1 Osajärjestelmäkomponentit

Osajärjestelmäkomponentit ovat sovellusriippuvia ohjelmia, jotka huolehtivat ainoastaan itselleen kuuluvista toiminnoista. Ne hoitavat keskinäisen yhteydenpitonsa kommunikaatiokomponentin avulla ohjelmistoväylän kautta, mutta eivät tiedä, kuinka tämä viestien välitys käytännössä tapahtuu eivätkä aina myöskään viestin vastaanottajaa.

Tunnistus on osajärjestelmä, joka huolehtii saapuneen kappaleen tunnistamisesta ja tunnistustietojen antamisesta hyödynnettäviksi. Se koostuu primääri- ja sekundääritunnistimista ja referenssitietokannasta. Ensisijaisia tunnistimia ovat viivakoodinlukija ja kamera, joita ei ole toteutettu pilotissa, vaan joista käytetään simuloituja tiedostoissa olevia tietoja. Toissijaisten tunnistimien antamia tietoja ovat nopeus, paino ja väri, jotka ovat pilotissa myös simuloituja ja saadaan tiedostoista.

Referenssitietokannasta on tunnistetietojen perusteella saatavissa esineen tyyppi ja tiedot. Tietokanta on toteutettu pilotissa koodinsisäisesti ja sisältää vain simuloitujen esineiden tiedot.

Kuljetus huolehtii esineiden siirtämisestä tarvittaviin kohteisiin, eli vastaanotosta tunnistukseen ja tunnistuksesta jatkokäsittelyyn. Jatkokäsittelijä ja kuljetus toimivat läheisessä yhteistyössä jatkokäsittelijän tarvittaessa kertoessa kuljetukselle, minne esine tulee siirtää.

Jatkokäsittely huolehtii toimenpiteiden suorittamisesta järjestelmässä oleville esineille. Se noutaa esineelle sen tyypin mukaan määrättyjä toimenpiteitä yksi kerrallaan järjestelmän sisällä olevien esineiden esinetietokannasta ja suorittaa tai suorituttaa toimenpiteet. Esine, jonka toimenpide haetaan ja suoritetaan, määräytyy sen sijainnin perusteella. Esineen sijainti määritetään antureiden avulla. Pilotissa käytettiin viivakoodiskanneria, jonka lukemien arvojen perusteella suoritettiin esineen lajittelu.

Pantinanto huolehtii pantin antamisesta joko kortille tai kuittina esineiden syötön loputtua. Pantti annetaan automaattisesti kortille, mikäli kortti on asetettu kortinlukijaan. Muutoin tulostetaan tekstikonsolille kuitin summa. Ohjelmistoväylä hoitaa liitynnän kortinlukijaan.

6.2.2 Ohjelmistoväylä

Osajärjestelmät kommunikoivat lähettämällä pyyntöjä ohjelmistoväylään, vastaanottamalla viestejä tai pelkästään tapahtumia (*events*), joita ne ovat erikseen pyytäneet eli joiden tilaajiksi ne ovat rekisteröityneet.

Ohjelmistoväylä hoitaa viestien reitityksen oikeaan kohteeseen riippumatta siitä, onko kohde tietokonejärjestelmän sisäinen prosessi vai LON-verkossa oleva verkkosolmu. Pilotissa toteutettiin järjestely, jossa viestin vastaanottaja päätellään viestin sisällön perusteella. Lisäksi ulkoiset liitännäislaitteet, kuten pilotissa kortinlukija pantinantoa varten sekä viivakoodinlukija pullojenkuljetusvaunun tunnistusta varten, on myös liitetty ohjelmistoväylään. Käytännössä edellämainitut liitännäislaitteet on liitetty RS-232-portteihin, joita varten ohjelmistoväylään toteutettiin erilliset ajurit. Näin ollen esimerkiksi tunnistuksen ja pantinannon kommunikointi liitännäislaitteisiin tapahtuu lähettämällä tai vastaanottamalla viestejä ohjelmistoväylään tai ohjelmistoväylältä.

Pilotissa ohjelmistoväylään liitettiin myös sovelluskohtainen osuus esinetietokannan ylläpitoon ja käsittelyyn. Palautusjärjestelmä käsittelee järjestelmässä kullakin hetkellä olevia esineitä suorittamalla esineille niiden tyypin mukaan toimintoja tai toimintosekvenssejä. Esinetietokannassa on keskitetysti kaikki järjestelmässä kullakin hetkellä olevat esineet. Tunnistuksen jälkeen esineeseen liitetään sille suoritettavat toimenpiteet, joiden suorituksen ohjaamisesta ohjelmistoväylä pitää huolen luomalla tarvittavat sanomasekvenssit eri osajärjestelmille. Näin on saatu aikaiseksi tietokeskeinen eli tiedolla ohjautuva esineiden käsittely eri osajärjestelmien välillä. Esinetietokannan tehtävänä on kätkeä kaikki osajärjestelmäriippuvaiset toiminnot muilta osajärjestelmäkomponenteilta. Toimenpidettä pyytävän osajärjestelmäkomponentin ei tarvitse tietää vastausta kysymykseen ‘miten’, vaan ainoastaan pyytää palveluita, kun niitä tarvitaan.

Osajärjestelmät kutsuvat ohjelmistoväylää ns. kommunikointikomponentin avulla, johon on toteutettu viestien luku ja kirjoitus käyttäen QNX-käyttöjärjestelmän viestijonopalveluita.

6.2.3 Kommunikaatiorajapinta

Osajärjestelmien kommunikaatiorajapinta käsittelee kaiken kommunikaation osajärjestelmän ja ohjelmistoväylän välillä. Kommunikaatiorajapintoja on kahta eri tyyppiä, tyypin määräytyessä osajärjestelmän luonteen mukaan. LON:in verkkosolmuissa kommunikaatiorajapinta on yksinkertaisesti verkkokuva, joka sisältää tiedon verkkomuuttujista ja niiden sidoksista. Muissa osajärjestelmissä kommunikaatiorajapinta on hieman monimutkaisempi ja ohjaa kaikki osajärjestelmien väliset viestit ohjelmistoväylään, joka toimittaa viestin oikeaan kohdeosajärjestelmään.

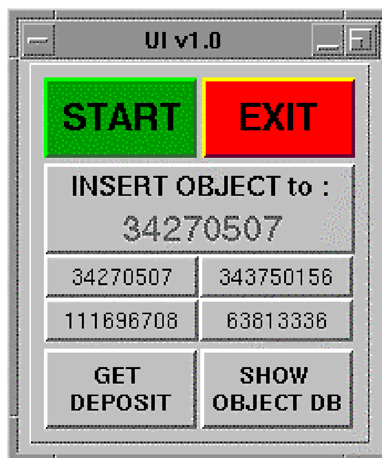
6.3 Pilotin toteutus

6.3.1 Graafinen käyttöliittymä

Pilotin graafinen käyttöliittymä käsittää kaksi erillistä ohjelmaa, käyttäjälle tarkoitetun liittymän, joka on nimetty *interface*ksi, ja ylläpitäjälle tarkoitetun liittymän, joka on nimetty *configurer*iksi. Edellinen on varsinainen käyttöliittymä, jolla järjestelmää ohjataan. Jälkimmäinen on ylläpitäjälle tarkoitettu työkalu, jolla voi muuttaa järjestelmän asetuksia ja esinetyyppien tietoja.

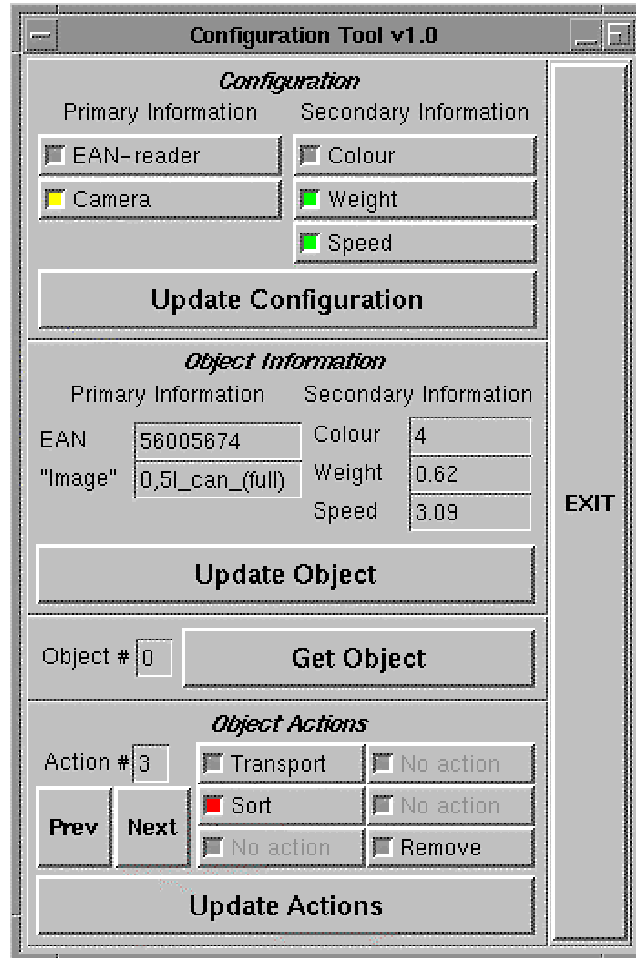
Pilotin käyttäjälle näkyvä osajärjestelmä, käyttöliittymäsimulaattori, toteutettiin englanninkielisenä, jotta mahdollisiin Suomen ulkopuolisiin demonstraatioihin ei tarvitse tehdä erillistä ohjelmistoversiota.

Käyttöliittymä järjestelmään on esitetty kuvassa 24. Pilotissa sillä myös käynnistetään ja suljetaan järjestelmä. Käyttäjälle kohdistettuja toimintoja ovat uuden esineen syöttäminen järjestelmään ja pantin pyyntö. Liittymällä tehdään myös ensimmäisen viivakoodinlukijan simulointi, jossa määritetään manuaalisesti viivakoodi vaunulle, johon uusi esine sijoittuu järjestelmässä. Ohjelman tulostukset tapahtuvat tekstikonsolille. Mukana on myös koko esinetietokannan tulostus, jolla voi tarkkailla kaikkien järjestelmässä olevien esineiden tietoja.



Kuva 24. Pilotin käyttöliittymä.

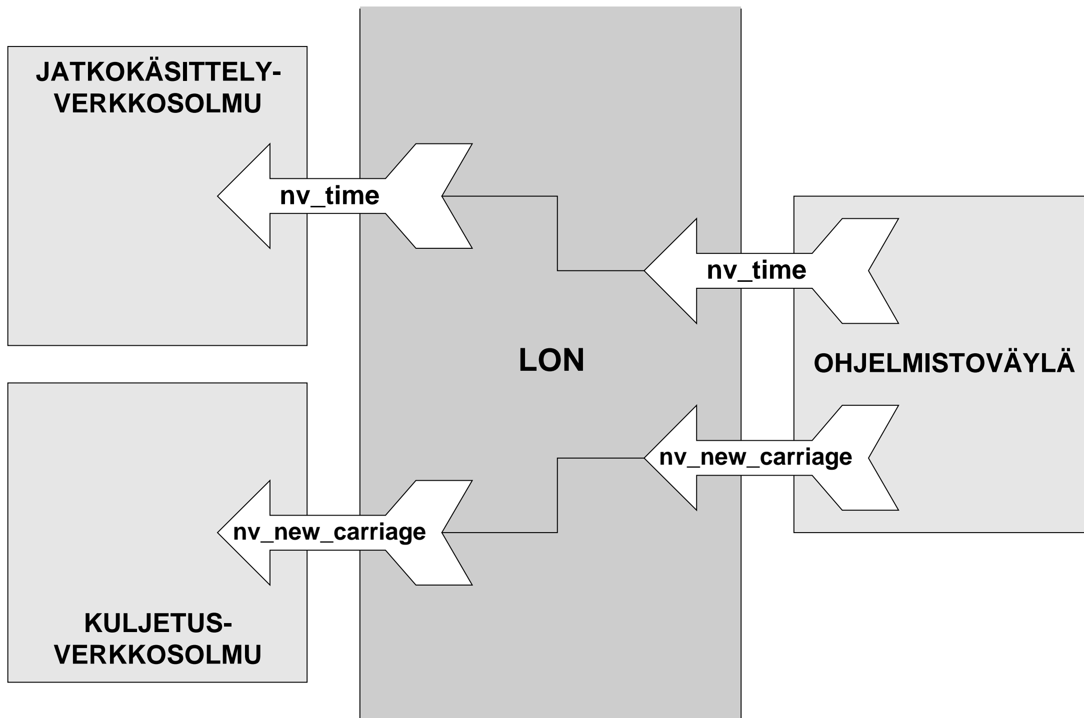
Konfiguroija on järjestelmän ylläpitäjän konfigurointintityökalu, kuva 25. Konfiguroijan kautta voidaan tarkastella ja muuttaa järjestelmän tunnistuslaitekonfiguraatiota, simuloitujen esineiden tietoja sekä niiden jatkokäsittelytoimenpidelistoja.



Kuva 25. Konfiguroija.

6.3.2 LON-liitynnän toteuttaminen ohjelmistoväylään

LON-verkon toiminta perustuu verkkomuuttujien käyttöön. Verkkomuuttuja on verkossa “näkyvä” arvo, jolla on merkitystä vain verkkosolmuille, joiden sovellusohjelmassa kyseinen muuttuja on määritelty. Kuvassa 26 on esimerkki verkkomuuttujista LON-verkossa. Verkkoon liitetyt solmut voivat muuttaa omassa sovellusohjelmassaan määriteltyjen verkkomuuttujien arvoa, jolloin muut vastaavaa verkkomuuttujaa tarkkailevat solmut saavat tämän arvon ja osaavat sovellusohjelmansa puitteissa toimia sen mukaisesti.



Kuva 26. Verkkomuuttujat.

Pilotissa käytetään tiettyä verkkomuuttujaa lajittelijan aktivoimiseksi jatkokäsittelijän toimesta. Verkkomuuttujan nimeksi on määritelty *nv_time* ja sen tyyppi on *SNVT_lev_disc*. Tälle verkkomuuttujalle annettu arvo kuvaa 500 millisekunnin jaksoissa aikaa, jonka kuluttua lajittelija tulee käynnistää. Pilotissa mahdollinen skaala on 1–9, eli 500ms - 4500ms. Kun jatkokäsittelijäsovellus PC-104-koneessa päivittää verkkomuuttujan, jatkokäsittelijäsolmu näkee sen LON-verkosta, ja kun muuttujassa määrätty aika on kulunut, se käynnistää lajittelijan lähettämällä sille vaaditut syötteet I/O-linjojensa kautta.

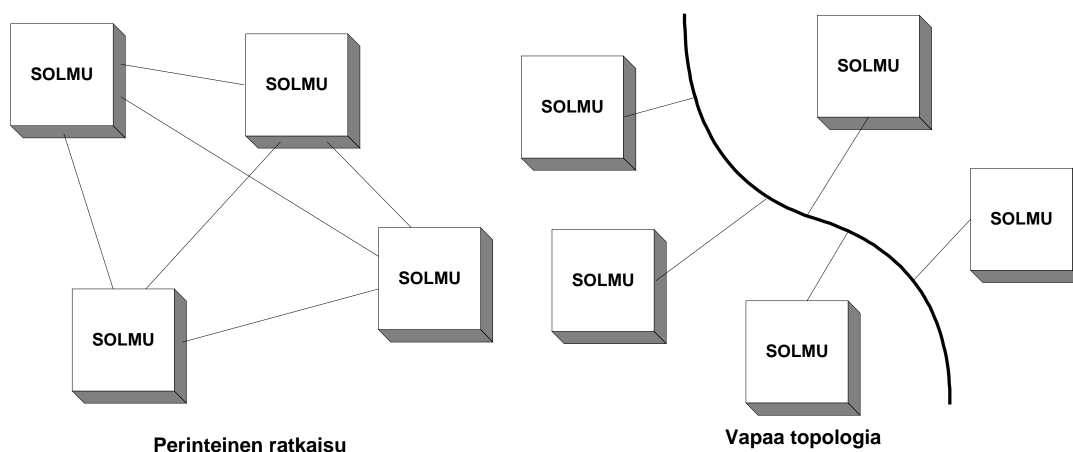
Hajautusratkaisu toteutettiin hyödyntäen valmiita kaupallisia ratkaisuja mahdollisimman laajasti. LON-verkon toteuttamiseksi tarvittiin:

- PC-104-liitynnällä varustettu PCLTA-kortti (Vista Electronics),
- PCLTA-kortin ajuri QNX -käyttöjärjestelmään (Steinhoff),
- LON-ohjainmoduulit kahdelle verkkosolmulle (tyyppiä TP/FT-10 Flash) ja
- LonBuilder.

LonBuilder on kehitystyökalu verkkosolmujen luomiseen ja koko verkon kokoamiseen. Sillä voi tehdä verkkosolmujen ohjelmat ja ladata ne solmuihin sekä muodostaa verkkokuvan, joka käsittää verkkomuuttujien kytkeytymisen verkon yli.

Ohjainmoduuli on laite, joka liittää solmun verkkoon, mutta voi ohjata myös solmun toimintaa. Ohjainmoduulissa on LON-verkon ydin, neuronipiiri. Neuronipiiri sisältää joko vain sisäistä tai sekä sisäistä että ulkoista muistia ja 11 konfiguroitavaa I/O-linjaa. Solmun laitteiden syötteet lähetetään ja vastaanotetaan näiden I/O-linjojen kautta. Ohjainmoduuliin voidaan ladata kehitysvaiheessa sovellusohjelma, joka toimii verkosta saapuvien verkkomuuttujapäivitysten ja niille määrättyjen toimenpiteiden mukaisesti.

Aluksi pilotissa kokeiltiin TP/FT-10 flash -ohjainmoduuleja, joissa on tyypin 3150 neuronipiiri ja ulkoinen flash-muisti. Näiden osoittauduttua viallisiksi vaihdettiin tilalle uusien vastaavien liian pitkän toimitusajan vuoksi 3120E2 neuronipiiri -pohjaiset FTT-10A lähetin-vastaanotinta käyttävät ohjainmoduulit ilman ulkoista muistia. Näiden ohjainmoduulien kanssa ei esiintynyt mitään ongelmia. Molempien ohjainmoduulien tapauksissa fyysisenä verkkona toimii vapaata topologiaa toteuttava parikaapeliyhteys. Kuvassa 27 on esitetty perinteinen verkkoratkaisu sekä vapaata topologiaa noudattava ratkaisu [27].



Kuva 27. Perinteinen ratkaisu ja vapaan topologian ratkaisu.

Sulautettuun PC:hen hankittu PC-104-liitäntäinen PCLTA-kortti (Vista LM104-P50-6 PCLTA) [32] toimii verkkoliityntänä LON:in ja PC:n välillä. Toimituksen mukana tulee LON-verkon ajuriohjelma, joka toimii PC:ssä. Verkkoajuri huolehtii PCLTA-kortin ja isäntäohjelman välisestä kommunikoinnista. PC:ssä toimivan isäntäohjelman avulla LON-verkkomuuttujia voidaan päivittää, jolloin verkkosolmut osaavat toimia sisäisten ohjelmiensa mukaisesti.

Kokemukset LONista ja LonBuilderista eivät olleet rohkaisevia. VTT:ssä aikaisemmin saadut kokemukset esimerkiksi CAN-väylästä ovat olleet positiivisempia. LON-järjestelmän kokoamiseen vaadittavat tuotteet eivät ole helposti asennettavia, ‘plug and play’ -periaatteella toimivia. Osittain huonot kokemukset johtuvat myös tietämyksen puutteesta, sillä LON:ista ei ollut laajaa käytännön kokemusta aikaisemmista projekteista. Aikaisemmat kokemukset perustuivat selvityksiin ja valmiiden

tuotepakettien avulla tehtyihin pieniin evaluointeihin. Toisaalta huonot kokemukset johtuvat osin myös maahantuojan kokemattomuudesta LONin kanssa.

QNX PCLTA-verkkoajuri osoittautui aluksi toimimattomaksi. Viikon vianselvittelyn jälkeen käytetty PC-verkkokortteineen ja -ajureineen lähetettiin Steinhoffille Saksaan vianselvitykseen. Ilmeni, että ajuri ei vakuutteluista huolimatta toimi kyseisenlaisessa kokoonpanossa, ja Steinhoff teki vaaditut muutokset. Muutostyön jälkeen ajuri toimi moitteetta.

Saatujen kokemusten perusteella voidaan todeta, että LON-pohjaiset tuotteet eivät vielä ole kaikilta osin kypsiä sovellettaviksi uusissa toimintaympäristöissä.

6.3.3 Ohjelmistoväylän RS-232-liitynnät

RS-232-liitännöillä tarkoitetaan tässä yhteydessä PC:hen sarjaporttien välityksellä kytkettyjä laitteita, jotka eivät ole suoraan tekemisissä LON-verkon kanssa. Pilotissa liitännäislaitteina olivat Miotec Oy:n toimittama ORGA:n kortinlukupääte [31, 33] ja DataLogic:in DS41-10 viivakoodiskanneri.

Pilotissa demonstroitiin yhtenä pantinantomuotona muistillista sirukorttia. Liityntä toimii siten, että mikäli kortinlukijaan asetetaan kortti, sen olemassaolo tunnistetaan automaattisesti ja pantinannon yhteydessä pantti summataan suoraan kortilla olevaan entiseen summaan.

Pilottia varten toteutettiin erillinen QNX-käyttöjärjestelmässä toimiva kortinlukijan ajuri toimituksen mukana saadun DOS-ohjelmaversion pohjalta. Valmista liityntää ei Miotec Oy tarjonnut, mutta kortinlukijan ajurin toteuttaminen sarjaväylään osoittautui kuitenkin suhteellisen suoraviivaiseksi tehtäväksi.

EAN-viivakoodiskannerin tehtävänä pilotissa oli tunnistaa esineitä kuljettavat vaunut. Viivakoodiskannerin ohjelmisto piti myös tehdä erikseen. Apuna käytettiin Halton System Oy:ltä saatua protokollaohjetta.

6.4 Johtopäätökset

6.4.1 Järjestelmän joustavuus

Ohjelmistoväylään perustuva arkkitehtuurikonsepti lisää palautusautomaattipilotin joustavuutta. Standardilaitteistoratkaisuun toteutettuna se on toimiva ja erottaa toteutusriippuvat asiat sovelluskohtaisista ratkaisuista.

LON-väylä soveltuu palautusautomaatin kuljetus- ja jatkokäsittelyosan hajauttamiseen erilleen varsinaisesta kappaleiden tunnistusosasta. LON-verkossa voidaan siirtää tehokkaasti lyhyitä kuljetus- ja jatkokäsittelyosan tarvitsemia sanomia. Pitkien sanomien, esimerkiksi kuvainformaation, siirtoon LON ei kenttäväyläluonteensa perusteella sovellu.

Varsinaisten sovellusten tekeminen LON-verkkoon on suoraviivaista, kun itse verkko ajureineen on saatu toimivaksi. Kyse on verkkomuuttujien avulla tapahtuvasta osajärjestelmien välisestä tiedonsiirrosta, jonka LON hoitaa itsenäisesti (pilotissa ohjelmistoväylän alaisuudessa). Saadut kokemukset LON-verkon käyttöönotosta ovat sensijaan huonot. LON-verkon käyttöönotossa oli paljon pikkuongelmia ja epämääräisyyksiä, joihin ulkopuolisen teknisen tuen saaminen oli hankalaa.

Konseptin toteuttaminen prototyyppiasteelle osoittautui mahdolliseksi 6–7 henkilötyö-kuukauden panostuksella. Sen toteuttaminen tuotteeseen on luonnollisesti paljon vaati-vampi tehtävä. Kaupallisia valmisohjelmistoja hyödyntämällä voidaan tosin säästää ai-kaa. Esimerkiksi pilotissa toteutettu simulaattori voitiin toteuttaa nopeasti Photonilla. Pilotti osoitti kuitenkin, että PC-104-ympäristöön toimitettavat valmisohjelmistot ovat usein puutteellisesti testattuja eivätkä toimi heti ensimmäisellä kerralla. Näin kävi mm. PC-104-kortin LON-liitynnän osalta. Kannattaa huomata myös, että käyttöjärjestelmän valinnalla on suuri merkitys valmisohjelmistojen saatavuuteen. Käytettäväksi valitun QNX:n tilanne on tässä suhteessa kuitenkin keskimääräistä parempi.

6.4.2 Vastaavuus suunnitelmaan

Pilotti täyttää tehdyssä hajautusmallisuunnitelmassa asetetut taulukossa 8 esitetyt vaatimukset hyvin. Vaatimukset ja toteutuksessa käytetyt ratkaisut on esitetty taulukossa 10.

Taulukko 10. Vaatimusten toteutuminen.

	Vaatimus	**Toteutettu ratkaisu**
1	Uusien sovelluskomponenttien eli osajärjes-telmien lisäämisen järjestelmään täytyy onnistua niin pienellä työllä kuin mahdollista.	Osajärjestelmät on toteutettu vakiorajapintaa käyttävinä itsenäisinä komponentteina, jotka voidaan helposti liittää järjestelmään LON:in verkkomuuttujien mahdollistamana.
2	Sovellusten täytyy olla sijainti-riippumattomia.	Verkkomuuttujaluonteen vuoksi ei ole tarpeen tietää missä tilaaja tai julkaisija sijaitsee (ks. kohta 1).
3	Laitteistomuutokset eivät saa vaikuttaa sovelluskomponentteihin.	Kaikki laitteistoriippuvat toiminnot on sijoitettu ohjelmisto-väylään niin, ettei osajärjestelmien tarvitse tietää niiden olemassaolosta mitään.
4	Sovellusten täytyy olla löyhästi kytkettyjä (*loosely coupled*).	Ohjelmistoväylä osaa viestin sisällön perusteella toimia halutulla tavalla, eli sovellukset ovat löyhästi kytkettyjä.
5	Samaa ohjelmistoalustaa täytyy voida käyttää kaikkien ohjelmistovariaatioiden kanssa.	Ohjelmistoalustan muuttamattomuus kaikkien ohjelmisto-variaatioiden kanssa perustuu kaikkiin edellisiin kohtiin.

6.4.3 Arkkitehtuurikonseptin skaalattavuuden arviointi

Pilot-toteutus tehtiin laajan järjestelmän tarpeita ajatellen. Eniten se näkyy keskusprosessointikortin tehokkuutena, RAM/FLASH-muistin määrässä ja liityntöjen monipuolisuutena (SVGA, näppäimistö, rinnakkaisportti, 4*RS-232, Ethernet ja LON-liityntä, sekä muutamia binäärisiä I/O-linjoja). Ohjelmistoa suunniteltaessa ajatuksena oli, että kehitettyä ohjelmistoarkkitehtuuria voitaisiin soveltaa myös pienissä kustannustehokkuutta vaativissa, yksinkertaisissa järjestelmissä. Siksi itse ohjelmistoväylään ja siihen liittyvien osajärjestelmiin perustuva arkkitehtuurikonsepti pidettiin mahdollisimman yksinkertaisena ja siirrettävänä.

Pienissä järjestelmissä ydintoiminta säilyy samanlaisena. Edelleenkin on kyse kappaleiden, pullojen, tölkkien ja korien tunnistamisesta. Liitynnät ulkomaailmaan sensijaan ovat pienissä järjestelmissä kustannussyistä rajoitetumpia. Tähän ei tarjota niin monipuolisia vaihtoehtoja, vaan osajoukkoa kaikista liityntämahdollisuuksista. Myöskään monipuoliset jatkokäsittelytoiminnot eivät ole olennaisia. Tämä näkyy ohjelmistoväylän sisällä.

Kun pilotissa kehitettyä ohjelmistoarkkitehtuuria sovelletaan pieniin järjestelmiin, ohjelmistoväylä ja osajärjestelmiin pohjautuva jaottelu säilyy samana. Pienissä järjestelmissä käytetty tehokas Pentium-pohjainen PC-104-tietokone korvataan luonnollisesti halvemmalla ratkaisulla. Ohjelmien siirrettävyyden varmentamiseksi kannattaa kuitenkin pitäytyä mahdollisimman PC-yhteensopivassa ympäristössä, esimerkiksi Intelin x86 -perheeseen kuuluvissa sulautetuissa prosessoreissa tai 286/386-pohjaisissa PC-104-korteissa, jotka suurissa kappalemäärissä saattavat osoittautua hinnaltaan kilpailukykyisiksi, sillä ne sisältävät laajennettavuutta ajatellen monipuoliset liityntämahdollisuudet.

Arkkitehtuuri edellyttää prosessipohjaista käyttöjärjestelmää. Pilotissa käytetty QNX-käyttöjärjestelmä on korvattavissa kevyemmällä reaaliaikakäyttöjärjestelmällä. Moniajon lisäksi käyttöjärjestelmän tulee tukea

- prioriteetteihin perustuvaa pre-emptiivistä skedulointia,
- riittäviä ajastinpalveluita, eli käyttöjärjestelmän tukemien ajastimien tulee toimia vähintään millisekunnin tarkkuudella,
- keskeytyspalveluiden hallintaa myös käyttöjärjestelmätasolla,
- taskien luomista käynnistyksen yhteydessä,
- sanomajonoja, ja sanomien priorisointimahdollisuus on etu.

Edellä mainitut vaatimukset täyttyvät useimmissa kaupallisissa pienissäkin reaaliaikaytimissä. Ajoaikaisten lisenssien hinta, lähdekoodin saatavuus, kehitystuki eli virheenetsintäohjelmat (*debugger*) ja kääntäjät, tekninen tuki ja kolmansien osapuolien valmisohjelmistot voivat olla teknisiä ominaisuuksia merkittävämpiä valintakriteerejä.

6.5 Jatkokehitysmahdollisuudet

Tehty ohjelmistoväylätoteutus on pilot-käyttöön soveltuva. Jatkossa ohjelmistoväyläkonseptia tulee soveltaa uusiin kohteisiin, jotta saadaan tietoa siitä kuinka ohjelmistoväylän geneeriset piirteet voidaan erottaa sovelluskohtaisista. Toisaalta ohjelmistoväylän tekninen toteutus on prototyyppiasteella ja sen laadun parantaminen prototyyppiä vastaavaan käyttöön soveltuvaksi on myös tarpeen.

Käytännössä jatkokehityksen ensimmäisenä vaiheena voisi olla ohjelmistoväylän älykkyyden lisääminen kehittämällä konfigurointituki, joka mahdollistaa järjestelmän autonomisen konfiguroitumisen hierarkkisiin verkkoihin ja niiden heterogeenisiin aliverkkoihin. Tällä saavutetaan järjestelmän parempi mukautuvuus ja laajennettavuus. Tähän liittyy läheisesti myös käyttöliittymän kehittäminen.

Ohjelmistoväylässä tulisi erottaa sovelluskohtaiset osat selkeästi geneerisistä osista sekä lisätä yleisiä perustoimintoja, kuten lohkosiirto ja kehittyneempi muistinhallinta. Myös käytetyn siirtoprotokollan joustavan vaihdon mahdollistaminen on tulevaisuuden tavoitteena.

7. Yhteenveto

Työssä selvitettiin, millaisilla ohjelmistoarkkitehtuuriratkaisuilla voidaan toteuttaa joustavia, helposti muunneltavia ja hajautettavia ohjelmistoja sekä toteutettiin hajautetun palautusautomaatin pilotti.

Eri vaihtoehtoihin tutustumisen ja syventymisen perusteella työssä päädyttiin käyttämään sovellusaluekohtaista hajautusalustamallia, joka pohjautuu prosessipohjaiseen sovellusalueriippuvista osajärjestelmäkomponenteista koostuvaan ohjelmistoväyläkonseptiin.

Ohjelmistoväylään perustuvan arkkitehtuurin havaittiin lisäävän palautusautomaattipilotin joustavuutta. Komponenttien välisten rajapintojen pitäminen standardinomaisena ja yksinkertaisena lisäsi järjestelmän laajennettavuutta, selkeyttä ja ylläpidettävyyttä.

Fyysiseksi hajautusalustaksi valittiin LON-verkko, jossa voidaan siirtää tehokkaasti lyhyitä osajärjestelmien kontrollointiin tarvittavia sanomia. Sovellusten tekeminen LON-verkkoon oli suoraviivaista, koska kyse on verkkomuuttujien avulla tapahtuvasta osajärjestelmien välisestä tiedonsiirrosta, jonka LON hoitaa itsenäisesti. LON-verkon käyttöönottoon liittyvät kokemukset ovat huonohkot, koska verkon käyttöönotossa oli paljon pieniä ongelmia ja epämääräisyyksiä, joihin ulkopuolisen teknisen tuen saaminen oli hankalaa ja maahantuojan tietämys olematonta.

Valitun arkkitehtuurin havaittiin olevan toimiva ratkaisu palautusautomaattijärjestelmien hajautukseen. Standardilaitteistoratkaisuun toteutettuna ohjelmistoväylä oli toimiva ja erotti toteutusriippuvat asiat sovelluskohtaisista ratkaisuista tehokkaasti. Pilotin toteuttaminen prototyyppiasteelle vaati noin puolen henkilötyövuoden työpanoksen. Valmiin tuotteen kehittämiseen kuluva aika tulee kuitenkin olemaan huomattavasti pitempi.

LÄHTEET

[1] Heikkinen, J. & Perunka, H. (1997). Palautusautomaatin järjestelmäarkkitehtuurin suunnitelma. VTT Elektroniikka. 18 s.

[2] Selic, B. & Ward, P. (1996). The Challenges of Real-Time Software Design. Embedded Systems Programming, Vol.9, No.11, October 1996. s. 66–82.

[3] Korhonen, J., Heikkinen, J., Pyhäluoto, T., Niemelä, E. & Perunka, H. (1995). Hajautetun reaaliaikaohjelmiston suunnittelu. VTT Elektroniikka. 36 s.

[4] Mullender, S. (1989). Distributed Systems. New York: ACM Press. 458 s.

[5] Booth, G. (1981). The Distributed System Environment. S. l.: McGraw-Hill Book Company. 294 s.

[6] Lamminmäki, S. & Hannus, J. (1993). Avoimet ja hajautetut tietojärjestelmät. Jyväskylä: HM&V Research Oy, Gummerus Kirjapaino Oy. 253 s.

[7] Gagliardi, M., Rajkumar R. & Sha, L. (1996). Designing for Evolvability: Building Blocks for Evolvable Real-Time Systems. Second IEEE Real-Time Technology and Applications Symposium, June 1996. S.100–109.

[8] General Electric. (1986). Software Engineering Handbook. New Work: McGraw-Hill Book Company. n. 200 s.

[9] Niemelä, E., Perunka, H. & Korpipää, T. (1998). A Software Bus as a Platform for a Family of Distributed Embedded System Products.

[10] Östlund, L. & Forssander, S. (1996). Management of Flexible Software Production Based on Reusable Components. Fifth European Conference on Software Quality Conference Proceedings. s. 395–407.

[11] Laffey, T. (1997). Making the Message Clear. Software Development, Vol.5, No.7, July 1997. s. 47–53.

[12] Loftus, C., Sherratt, E., Gautier, R., Grandi, P., Price, D. & Tedd, M. (1995). Distributed Software Engineering. Padstow, Cornwall, Great Britain: Prentice Hall. 260 s.

[13] ISO/IEC. (1991). Information technology - Software product evaluation - Quality characteristics and guidelines for their use. International standard 9126, ISO/IEC.

[14] Biggerstaff, T. & Perlis, A. (1989). Software Reusability Vol.1 Concepts and Models. New York: ACM Press. 425 s.

[15] Hooper, J. & Chester, R. (1991). Software Reuse Guidelines and Methods. New York: Plenum Press. 180 s.

[16] Gulu Gambhir, S. (1997). Use of Domain Analysis to Implement the Developer Off-The-Shelf Systems (DOTSS) System Acquisition Approach, Software Engineering Notes, Vol.22, No.2, March 1997, s. 48–53.

[17] Davis, M. & Williams, R. (1997). Software Architecture Characterization, Software Engineering Notes, Vol.22, No. 3, May 1997, s. 30–38.

[18] Lam, W. & McDermid, J. (1997). A Summary of Domain Analysis Experience By Way of Heuristics, Software Engineering Notes, Vol.22, No.3, May 1997, s. 54–64.

[19] Lawson, H. (1992). Parallel Processing in Industrial Real-Time Applications. New Jersey: Prentice Hall, Englewood Cliffs. 514 s.

[20] Gomaa, H. (1993). A reuse-oriented approach for structuring and configuring distributed applications. Software Engineering Journal, March 1993, s.61– 71.

[21] Amdahl, G. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing. AFIPS Conference Proceedings, Vol.30. Washington D.C: Thompson Books. s. 483–485.

[22] Henderson, K. (1996). Mastering Middleware. Software Development, Vol.4, No.11, November 1996, s. 38–44.

[23] OMG. (1995). The Common Object Request Broker: Architecture and Specification. Object Management Group. Framinghman, USA.

[24] Rajkumar, R., Gagliardi, M. & Sha, L. (1995). The Real-Time Publisher/Subscriber Communication for Inter-Process Communication in Distributed Real-Time Systems. The First IEEE Real-Time Technology and Application Symposium, May 1995.

[25] Schill, J. (1997). An Overview of the CAN Protocol. Embedded Systems Programming, Vol.10, No. 9, September 1997, s. 46–62.

[26] Alanen, J. & Virtanen, A. (1994). Ylemmän kerroksen CAN-kommunikointi-arkkitehtuurit. Espoo: Valtion teknillinen tutkimuskeskus. 61 s. (VTT Tiedotteita 1561.)

[27] Korhonen, V. (1996). Beginner's Guide to LonWorks. Vaasa, Finland: ABB. 30 s.

[28] Echelon Corporation. (1993). LonWorks Host Application Programmer's Guide. Echelon Corporation, Palo Alto, California.

[29] [verkkojulkaisu]Saatavissa :http://www.echelon.com/Products/datasheets/pdtools.htm

[30] QNX Software Systems Ltd. (1995). Photon microGUI PhAB User's Guide. QNX Software Systems Ltd, Canada. 204 s.

[31] Gaubatz, J. & Reimann, R. (1994). Programming Set for ORGA Card Readers. Paderborn, Germany: ORGA Kartensysteme GmbH.

[32] [verkkojulkaisu] Saatavissa: http://www.vistaelectronics.com/lm104-p50.htm

[33] Gaubatz, J., Reimann R. & Held, D. (1994). Standard Communication Protocol for ORGA Chip Card Readers. Paderborn, Germany: ORGA Kartensysteme GmbH. 146 s.

PIKAOPAS JÄRJESTELMÄN KÄYTTÄMISEEN

Laitteiston valmistelu

LON-verkko

LON-verkon saattaminen toimintakuntoon käsittää virran kytkemisen järjestelmään sekä verkkoajurin käynnistäminen.

PC-104-kortti-PC:ssä oleva PCLTA-kortti täytyy käynnistää ja alustaa verkkoyhteyden saattamiseksi toiminnalliseksi. Tämän vuoksi tietokone on käynnistettävä ja siihen on kirjoittauduttava sisään. Verkkoajurin käynnistys on liitetty osaksi järjestelmän käynnistyssekvenssiä, joten ajuria ei tarvitse erikseen käynnistää.

Jos kuitenkin käynnistystietoihin tehdään myöhemmin muutoksia esimerkiksi käyttöjärjestelmäversion päivityksen yhteydessä eikä ajuri käynnisty itsekseen, sen saa tehtyä manuaalisesti kirjoittamalla juurihakemistossa *londrv -i 10 -p 4 &*. Verkkokortti on konfiguroitu IRQ:lle 10 ja porttiin 320-32F.

Verkkosolmujen toimintakuntoon saattamiseksi niihin täytyy ainoastaan kytkeä käyttöjännite.

Liitännäislaitteet

Liitännäislaitteiden toimintakuntoon saattaminen käsittää niiden liittämisen PC-104--tietokoneeseen sekä jännitteen kytkemisen. Hiiren oletetaan olevan sarjaportissa COM2. PC-104 kortti-PC:n keskeytysasetusten on oltava jakamattomat (*non-shared*), eli jokaiselle COM-portille on oma keskeytyslinja. COM1:n ja COM2:n keskeytyslinjat ovat vakiot, IRQ 4 ja IRQ 3, mutta COM3:n keskeytyslinja voi vaihdella kokoonpanosta riippuen. Pilotin kortti-PC:ssä COM3:n keskeytyslinja on IRQ 12.

Kortinlukija on liitettävä sarjakaapelilla PC:n sarjaporttiin COM1. Tämän jälkeen sen jännitelähde on kytkettävä verkkojännitteeseen.

Viivakoodiskanneri on liitettävä sarjakaapelilla PC:n sarjaporttiin COM3. Tämän jälkeen se on kytkettävä +11 - +30V tasajännitteeseen. Viivakoodiskanneri lukee lasersäteen avulla, joten sitä käytettäessä on oltava huolellinen, ettei lasersäde osu kenenkään silmiin.

Käyttöliittymä

Ohjelmiston suoritus aloitetaan käynnistämällä QNX:n graafinen käyttöliittymä Photon. Photonissa avataan tekstikonsoli *pterm* tai *shell*, jossa hakemisto vaihdetaan ohjelmahakemistoksi. Kirjoitetaan tekstikonsolille *interface*, joka avaa näytölle käyttöliittymä-ikkunan. Käyttöliittymän toiminnot on esitetty taulukossa 1.

Taulukko 1. Käyttöliittymän ja konfiguroijan toiminnot ja näkymät.

Toiminto tai näkymä~~Toiminto tai näkymä~~	**Tarkoitus**
Käyttöliittymä	
Käynnistys	Ohjelmisto käynnistetään painamalla painiketta *START*.
Uuden esineen syöttäminen	Järjestelmään voidaan lisätä esineitä *INSERT OBJECT* -painikkeella.
Esinetietokannan tulostus	Kaikki järjestelmässä olevat esineet saadaan näkyviin painamalla *SHOW OBJECT DB* -painiketta.
Ohjelmiston sammutus	Ohjelmisto sammutetaan painamalla *EXIT*--painiketta.
Konfiguroija	
Tunnistuslaitekonfiguraatio	Järjestelmän senhetkinen tunnistuslaitekonfiguraatio näytetään lohkossa *Configuration*.
Esinetietojen luku	Esinetiedon luku suoritetaan kirjoittamalla kohtaan *Object #* esineen numero ja painamalla *Get Object* -painiketta.
Esinetiedot	Haetut esinetiedot näytetään lohkossa *Object Information*.
Esinetoimenpiteet	Haetut esinetoimenpiteet näytetään lohkossa *Object Actions*.
Konfiguroijan sulkeminen	Konfiguroija suljetaan painamalla *EXIT*–painiketta.

Ohjelmisto käynnistyy painamalla hiirellä painiketta *START*, jonka jälkeen on tarpeen odottaa noin 5 sekuntia. Tämä siksi, että *START*--painikkeen painalluksen seurauksena käyttäjältä piilossa ajetaan ohjelma *start*, joka käynnistää kaikki osajärjestelmät vuorollaan.

Ohjelmiston käynnistyttyä voidaan järjestelmään lisätä esineitä *INSERT OBJECT* --painikkeella painamalla siinä näkyvää tekstiä, ei numeroa. Painikkeessa näkyvä numerosarja on järjestelmässä olevan vaunun, johon esine lisätään, viivakoodi. Tätä viivakoodia käytetään vaunutunnisteena jatkokäsittelyä varten. Painikkeessa olevaa viivakoodia voidaan muuttaa *INSERT OBJECT* -painikkeen alapuolella olevia pieniä painikkeita painamalla. Omia viivakoodeja voidaan lisätä painamalla *INSERT OBJECT* -painikkeessa näkyvän numerosarjan päälle ja kirjoittamalla haluttu viivakoodi. Pilotissa on huomattava, että hyväksytty viivakoodi on EAN-tyyppinen, ja siitä poistetaan kaksi ensimmäistä numeroa sekä kaikki 9:nen numeron jälkeiset numerot.

Kaikkien järjestelmässä olevien esineiden tiedot saadaan tulostettua tekstikonsolille painamalla *SHOW OBJECT DB* -painiketta. Tämä näyttää sekä esineiden tiedot että vapaana olevat esineindeksit.

Ohjelmisto sammutetaan painamalla *EXIT*–-painiketta. Tällöin suoritetaan *slay_all* --skripti, joka sulkee vuorollaan kunkin osajärjestelmän sekä viimeiseksi *interface*:n.

Ylläpitoliittymä

Konfiguroijan suoritus aloitetaan käynnistämällä Photon, mikäli se ei ole jo käynnissä. Photonissa avataan tekstikonsoli *pterm* tai *shell*, jossa hakemisto vaihdetaan ohjelmahakemistoksi. Tekstikonsolille kirjoitetaan *configurer*, joka avaa näyttöön konfiguraattori-ikkunan. Järjestelmätietojen konfigurointi kannattaa suorittaa käyttöliittymän, ja täten järjestelmän, ollessa suljettuna tiedostojen luku- tai /kirjoitus–-konfliktien välttämiseksi. Tietoja voi myös muuttaa ajon aikana, mutta tämä ei ole suositeltavaa. Konfiguroijan toiminnot ja näkymät on esitetty taulukossa 1.

Konfiguroija näyttää järjestelmän senhetkisen tunnistuslaitekonfiguraation ylimmäisessä lohkossaan (*Configuration*). Konfiguraatiota voi muuttaa vapaasti painikkeita painelemalla, painikkeissa olevien valojen näyttäessä senhetkiset asetukset. Muutetut tiedot päivitetään painamalla *Update Configuration* -painiketta.

Konfiguroijan käynnistyessä yhdenkään esineen tiedot eivät ole näkyvissä. Esinetiedot ja -toimenpiteet täytyy erikseen lukea tiedostoista joihin ne on talletettuina. Tämä tapahtuu kirjoittamalla halutun esineen numero *Object #* -tekstilaatikkoon ja painamalla vieressä olevaa *Get Object* -painiketta.

Kun esinetiedot on haettu, ne tulostuvat keskimmäiseen lohkoon (*Object Information*). Näitä tietoja voi muuttaa painamalla kyseiseen tekstilaatikkoon ja kirjoittamalla siihen haluamansa arvon. On hyvä muistaa, että viivakoodi saa olla korkeintaan 9 merkkiä pitkä, vaikka sitä ei ole syöttäessä rajoitettu mitenkään. Kuvainformaatio on pilotissa simuloitu esineen kuvailevalla tekstillä. Tässä tekstissä ei saa olla välilyöntejä eikä tulostumattomia erikoismerkkejä. Esinetiedot päivitetään painamalla *Update Object* --painiketta.

Kun esinetiedot on haettu, tulostuu esineen ensimmäinen toimenpide alimmaiseen lohkoon (*Object Actions*). Esineen toimenpiteitä voi selata *Prev*- ja *Next*-painikkeilla, tai kirjoittamalla toimenpiteen numero suoraan tekstilaatikkoon *Action #*. Toimenpiteen voi valita painamalla oikeassa laidassa olevia painikkeita, joissa oleva valo ilmaisee valittuna olevan toimenpiteen. Mikäli toimenpiteitä halutaan lisätä, kirjoitetaan tekstilaatikkoon *Action #* haluttu toimenpidemäärä, ja sitten määritetään toimenpide kullekin toimenpidenumerolle käyttämällä *Prev*- ja *Next*-painikkeita. Viimeiseksi toimenpiteeksi on aina määritettävä *Remove*, tai esinettä ei koskaan poisteta järjestelmässä olevien esineiden tietokannasta, mikä aiheuttaa virhetilanteita ennemmin tai myöhemmin. Toimenpidetiedot päivitetään painamalla *Update Actions* -painiketta.

Konfiguroija suljetaan painamalla *EXIT*–-painiketta. On huomioitava, että tehtyjä muutoksia ei talleteta ohjelmasta poistuttaessa automaattisesti, vaan ne on talletettava erikseen kussakin lohkossa olevalla *Update*-painikkeella.

Virheenkäsittely

Ohjelmistoon on toteutettu virheenkäsittely sekä virheloki. Virheenkäsittely on alkeellinen ja käsittää ainoastaan virheen aiheuttaneen toiminnon uudelleenyrityksen tiettyjä kertoja, jonka jälkeen, virheen yhä esiintyessä, virheestä ilmoitetaan tekstikonsolille ja ohjelmisto suljetaan hallitusti.

Virhelokia päivitetään automaattisesti aina virheen tapahtuessa, ja siihen päivittyvät myös tiedot uusintayrityksistä. Lokia ylläpidetään tiedostossa *errorlog.txt*, johon talletetaan virheen tapahtumishetki sekä kuvaus itse virheestä. Koska viivakoodinlukija tuottaa silloin tällöin virheitä, joista ei ole syytä huolestua, on lokitiedosto hyvä tuhota tietyin aikavälein, jottei sen koko kasva liian suureksi.