Service Discovery Mechanisms: two case studies

J.J. Lukkien, T.Tranmanh, P.H.F.M. Verhoeven, P.J.F. Peters
Eindhoven University of Technology
Department of Mathematics and Computer Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
Email: j.j.lukkien@tue.nl; presenter: Tranmanh, Peters or Lukkien

Abstract

Connecting other devices than workstations to the Internet is becoming commonplace now. In this article we investigate two approaches to the discovery of devices and the services they may provide, viz., the Universal Plug 'n Play standard by Microsoft and the wireless Corba model.

Keywords: *embedded system, architecture, service discovery, middleware, auto-configuration*

1 Introduction

Connecting other devices than workstations to the Internet is becoming commonplace now. Examples are the equipment, traditionally connected to a computer like a printer, a scanner or a disk (file server). In addition, the number of mobile devices like PDA's and telephones increases rapidly. All these devices are capable of providing services to users, which, in turn, might be electronic devices themselves. For example, a printer provides a print service but needs a file server to store large print files.

It may be expected that in the future just about every embedded system becomes connected to a network. (Re)configuration of the system will be needed all the time, even more since many of these systems are mobile. It is mandatory that systems configure themselves without user intervention. This holds in particular for consumer electronics applications. While on the one hand human intervention cannot be counted on, on the other hand there cannot be high expectations of the environment of the system.

The dynamic formation of networks of cooperating devices concerns various layers in the communication, see Figure 1. At the physical layer, it must be possible to establish a connection. This amounts to establishing a point-to-point connection with some base station (e.g., GSM or calling into a modem pool) or joining a network directly (e.g., bluetooth and wireless or wired ethernet). In the latter case the device has to have or receive a MAC address; in the former case this can be postponed by letting the base station serve as a gateway. An example of this is a printer connected to a PC that is made available as a network printer.

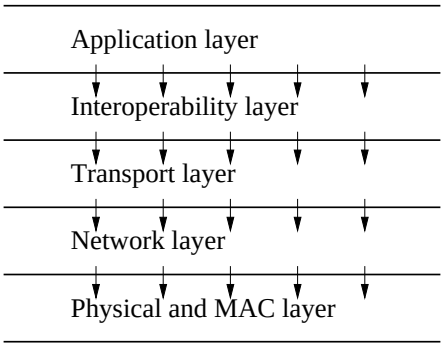

Figure 1: *A simplified picture of the functional layers in a network. The physical and MAC layers establish connectivity. Network and transport layer enable information exchange. Together these for the the communication infrastructure that admits transport of information between arbitrary endpoints. The interoperability layer represents the way applications use the transport layer. It provides services to the applications.*

The network and transport layers then build on this to provide arbitrary information transport between identifiable endpoints. Still the option is open to address each endpoint individually or to let the identification of machines behind a gateway be part of the payload of the messages sent to the gateway. Within the Internet philosophy there is strong support for the end-to-end semantics. However, it requires TCP/IP stacks at all endpoints which might not be useful in all cases. In addition, some network addressing problems are resolved in this way through the technique of *Network Address Translation* (NAT/NAPT, [1]).

The allocation of unique addresses relies on the

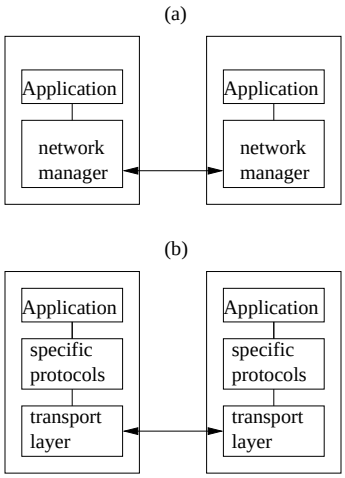

Figure 2: *Two software architectures for interoperability. In case (a) the focus is on functionality which may be provided as an API. In case (b) the functionality is through specific protocols on top of a transport facility.*

cooperation of the attached machines. In *nomadic networking* there is an existing infrastructure to which the new terminal connects. Obtaining a unique address can then be a service on the network. In *ad-hoc networking* there is no such infrastructure and one has to rely on protocols for automatic assignment of addresses. Particular problems in this case are the questions of naming and addressing when two such networks join ([2]).

In principle, applications can use any transport facilities directly. However, because networked applications concern the cooperation of at least two devices which are programmed independently there has been much standardization in this area. This results in an *interoperability layer* between the application and the communication infrastructure, often called the *middleware*. Examples of middleware technologies include Remote Procedure Calls (RPC's), Transaction and message oriented middleware, distributed DBMS's and object oriented middleware of which the Object Request Brokers are most dominant (see e.g., [3]).

In this article we are concerned with one particular middleware issue, viz., *service discovery*. Devices generally provide value to users in the form of a service they deliver. Examples of services are a print service, a display service, a file service but also much simpler services like switching a light. Service providers must be able to advertise their services; service users must be able to find the services. We are involved in two projects that both address this issue but from a different perspective. In one project we are developing an open source Java version of the UPnP standard; the other project uses the wireless Corba standard. In this article we first discuss some general issues concerning service architectures. Then we describe the work we did here (in particular, the demonstrators), we comment on the two standards (describing shortcomings and proposing alternatives) and we compare them.

2 Architectures

There are generally two aspects in the software architectures for interoperability. For developers of applications, the abstraction of the network is provided through an API. The service provided by the API may range from simple transport facilities to the advanced functionality corresponding to remote procedure calls and other middleware techniques. In the first case the interoperability is limited to the exchange of messages (i.e., the transport layer operates as the interoperability layer). In the second case the advantage is just the same as using standard interfaces and library functions: software re-use and portability.

The second aspect concerns the implementation of this API using communication across the network. By standardizing on the used protocols and message formats it is possible for two applications to be developed separately and still be able to cooperate.

Traditional middleware has focused mainly on the API. The result is that typical network issues are kept away from the application. For example, RPC mechanisms usually assume reliable communication; disconnection is regarded as an error. The API is not aware of mobility issues such as the changing physical location and frequent disconnection. The corresponding architecture is as in Figure 2(a). This organization has all the advantages of abstraction through layering. In principle there is no need to have a full transport layer underneath. Instead, this approach just requires the availability of an implementation of the interface. This implementation can be regarded as a "driver" for the remote machine. It can take the form of a library or it can be downloaded from the network (but then some bootstrapping use of the network must be available).

On the other hand, many protocols are described and used without a specific API. For example, the HTTP definition refers to the order, contents and meaning of messages across the Internet. There is no particular API defined and HTTP applications commonly use the transport layer directly. The result is that aspects of the protocol and of the underlying network technology carry through into the

application. This happens in particular when the protocol is advanced or when different protocols are combined in a single application. The resulting architecture is as in Figure 2(b).

The best choice is to do both: the interoperability layer is then defined through the combination of an API and a protocol. The latter may differ for different network technologies. In the design of the API it is important to recognize issues that should be made available to applications rather than make them transparent. These issues include awareness of location, (dis)connection, distribution, and of resources. The result of this awareness in the interface is a different model presented to the application. If we take mobility as an example then the API of mobile middleware should make it possible to separate cooperating applications both in time and space. A separation in space means that they do not need to work on the same data. A separation in time means that they communicate in an *asynchronous* fashion. Roughly this implies that information streams between the two partners are separated. Requests may generate replies but these are treated as a separate event stream. Information may be stored temporarily to overcome connection losses.

3 Services

In general, a service can range from the unstructured use of a device to the structured control of some well-defined functions. For example, a display service might accept a digital video stream or it might offer a structured window-interface like X-windows does. Also, the *integration* of services a device offers may differ. For example, a TV-set might offer audio and video service separately or it may require a single integrated use. The more structure is put into the service definition the more requirements are imposed on the client side.

One of the issues is how the service is made available and can be used. There are mainly three approaches. First, the device can provide a driver that includes the code how to use it. This driver can take the form of a Java program or some other portable code. An advantage is that rather complicated services can be made available easily. However, it requires an advanced platform. Second, the interface available to the application can be described in an interface definition document. Using the described interface is through a standard method like Corba. Third, one can rely on standardized protocols to approach the service. The advantage in this case is the simplicity.

A particular application of services is a *remote user interface*. In one extreme view this user interface can be downloadable code such as a Java Applet; on the other side of the scale an application could try to compose a user interface from the details of the services it finds. Somewhere in the middle is a user interface description through some sort of language in which UI components and their appearance are described. The client does not have to run anything as complicated as a virtual machine while it also does not need the intelligence to compose a user interface. In earlier work[4] we have combined some of the approaches: the protocol for using the service is explicitly available while a user interface in the form of an applet is available as well.

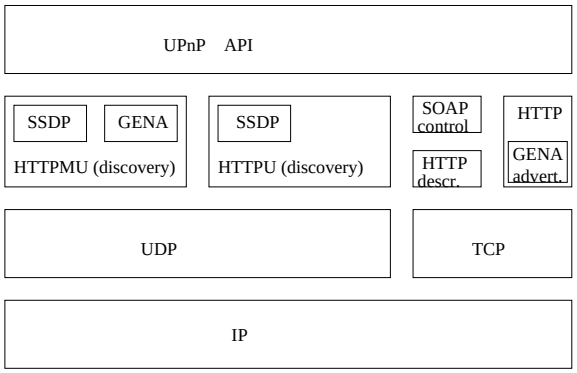

Figure 3: *Universal Plug 'n Play Architecture. There are generally two* roles *in a UPnP application: a* device *point and a* control *point. These are similar to the notions of server and client though it is quite common that a single piece of equipment has both roles.*

Service discovery mechanisms can be classified as distributed or centralized. In the centralized case there is a server that records the available services as well as methods to use them. An example of this is Jini[5], based on the language Java. Service providers store a reference to the service and a Java "driver" in a database. Service users can retrieve and use this. In the distributed case there is no such service store but there are methods to make services known onto the network. Finding services as well as advertising them looks in principle the same. They amount to some form of broadcasting.

4 UPnP

Universal Plug and Play was designed as an architecture for peer-to-peer network connectivity of intelligent appliances, wireless devices, and PCs of all form factors. Through UPnP a device can dynamically join a network, convey its capabilities, learn about the presence and capabilities of other devices

and leave a network smoothly and automatically. UPnP comes without device drivers; common protocols are used instead. The UPnP architecture[6] is illustrated in Figure 3. As can be seen from this figure the UPnP architecture relies heavily on Internet protocols like IP, TCP, UDP and HTTP (plus extensions). A good reason for employing HTTP is that it brings all the advantages from current Internet technology: e.g. the device can be controlled directly using a Web browser. The use of HTTP within UPnP is mainly as a transport facility transporting XML-data structures (SOAP and GENA) and message acknowledgments; this could be easily changed to some other transport mechanism.

Inheriting the IP protocols into the architecture makes UPnP widely applicable and all the advantages of these well known protocols can be benefited from. At the downside this implies inheriting the problems mentioned previously, e.g. naming and addressing problems. The additional UPnP layers and functionality do not add any extra naming or addressing problems.

UPnP communication can be split into several levels: Application, API, HTTP, UDP/TCP and IP. Plain messaging can be synchronous or asynchronous. The real mode of communication is determined by the actual implementation. Any problems that exist in the underlying protocol (HTTP, TCP, UDP, IP) will be present in UPnP as well.

The UPnP API focuses mainly on the functionality that a device or a control point designer needs and hides the protocols behind it. As such it presents itself in a way as in Figure 2(a). Building a user interface for a device or a service is not a part of the UPnP API, but there are various ways this could be done.

- derive a generic user interface from the device- and service description and build a user interface dynamically,
- define a specific interface in an XML interface structure definition and read this from the device,
- download a specific interface or define a specific interface within the device's control point.

UPnP has a distributed discovery mechanism where devices and services are found by discovery messages sent by control points and by advertisement messages sent by devices. The mechanism used to implement this is UDP multicasting.

We have developed an UPnP API in Java that we apply to some industrial cases in order to evaluate UPnP. The UPnP API includes an API for client (Control Point) and an API for server (Device). Both haves been built based on the general UPnP architecture (Figure 3) defined by the UPnP forum. Then we apply for the first use case:

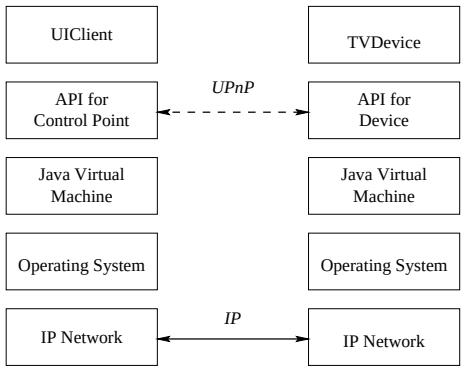

Figure 4: *The TVControl protocol stack*

TVControl prototype. This prototype is a simulation of a user interface (named UIClient) running in a computer to control a television (named TVDevice). UIClient has following ability:

a *Switch TVDevice on/off*
b *Change channel number*
c *Change volume number*
d *Change color effect*
e *Change contrast effect*

To define TVDevice, we divide above functionalities into two services: this first one, named ControlService, which fulfils the first three functions (*a*, *b* and *c*) and the second one, named ScreenService, which fulfils the lasts (*d*, *e*). Three XML files have been construct in order to specify TVDevice. One file specifies TVDevice and two others specify its two services. After that, we have built TVDevice and UIClient, based on the UPnP API. The protocol stack for TVControl is described in Figure 4. In the beginning, we built one UIClient and one control TVDevice. Five steps that UPnP must go through are:

- UIClient and TVDevice use SSDP to discover each other
- UIClient retrieves the TVDevice description and get a list of associated services
- UIClient retrieves service descriptions for interesting services
- UIClient subscribes to the service's event source. Any time the state of the service changes, the event server will send an event to UIClient
- UIClient raises actions to control the service

The work of SSDP[7] in this case includes TVDevice sending advertisements about itself and UIClient searching for TVDevice. This is a very robust approach. To search for TVDevice, the user has to type its identification number or its type, as the search target ST can accept following values:

ssdp:all : Search for all devices and services.
upnp:rootdevice : Search for root devices only.

uuid:device-UUID : Search for a device with its identification.

urn:schemas-upnp-org:device:*deviceType:v* : Search for any device with its type and version.

urn:schemas-upnp-org:service:*serviceType:v* : Search for any service with its type and version.

Doing in this way is not convenience for the user in home networking environment (the environment that UPnP is designed for). He has to remember this information. In practise, the user tends to use the "brow-and-click" way in searching for devices. For example, to find his television, first he may choose the first floor of his house then comes to living room and picks the television. He doesn't have to remember about the identification number of his television. The system remembers it for him. In general, how to create a suitable naming mechanism in UPnP is an open question.

Context-based mechanism is the way to make UIClient understand about and later to control TVDevice. 5 commands (a,b,c,d and e) are specified in two XML-based files, so that UIClient can parse these and quickly interact with TVDevice. Using context-based method ensures that UIClient and TVDevice are platform independent. However, we see that UPnP is still lacking the way of specifying real-time constraints, using XML. If UPnP has this specification, UIClient can set-up real-time channels between it and TVDevice, in order to display television content in the client site. Research on this direction may lead to develop an QoS for UPnP technology, also a new way of specification for devices.

The interface of UIClient has been generated automatically by the UPnP API from three above XML files. With only one UIClient to control TVDevice, the system works very well. Next, to consider the scalability and performance, we increased the number of UIClient(s) gradually from 10 to 100. The system still worked well. But when there were more than 100 UIClient(s), we saw that several UIClient(s) worked inconsistent with TVDevice, especially in the discovery part. For example, we started-up 101st UIClient and let it do the discovery. In the first time it could not detect TVDevice and in the second time it does. During this discovery, all other UIClient(s) are still in active mode. The reasonable explanation for this problem could be by increasing number of operations in the network, too much traffic might be created. If many devices and control points send multicasting messages at the same time, some messages could be lost. This way of operation could degrade the network performance.

In conclusion, using UDP multicasting for discovery is suitable if there are not so many devices in the network. There could be a model to evaluate the balance between number of devices in the network and network performance requirement. UDP multicasting is a way for service discovery, but it is not an only method.

In UPnP, basically all devices and control points are HTTP(extension) servers. So the security concern in UPnP is delegated to the problem of maintaining security in each HTTP(extension) server. In fact, UPnP doesn't address this problem clearly. Instead, it let the developer freely to decide how security they are for HTTP(extension) servers.

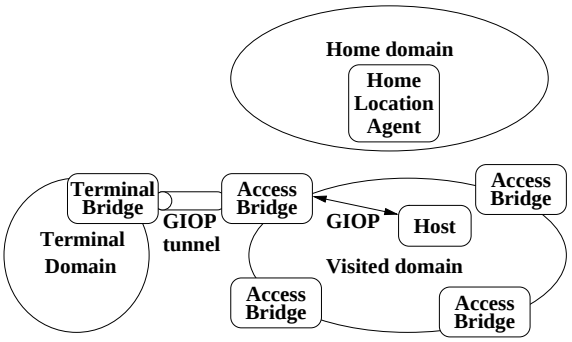

Figure 5: *The Architecture for Wireless CORBA. An Access Bridge acts as a gateway for the terminal and forwards the GIOP connection in case of handovers. The GIOP tunnel abstracts from the physical layer. The Home Location Agent redirects traffic to the correct Access Bridge.*

5 Wireless CORBA

CORBA provides an object oriented abstraction of a network and its architecture is classified as in Figure 2(a). However, CORBA requires reliable connections between fixed end-points, which does not apply to mobile devices like mobile phones or PDAs. To address the problems that occur when end-points are moving, like changing network addresses and hand-over issues, the wireless CORBA specification of OMG[8] provides solutions to most of these problems. The architecture underneath the protocol is giving in Figure 5, which resembles the architecture for mobile phones, with mobile terminals and access points. The terminal bridge is located on the mobile terminal and makes connections with access bridges. An hand-over to a different access-bridge is either initiated on request of the terminal, which discovers that a different access bridge is available, or on request of the access bridge, which discovers that the terminal is within range of a better access bridge. Connection recovery is handled by making a connection to an access bridge, which might be different from the access

bridge that was used last time. Both hand-over and connection recovery might result in a changed network address for the terminal and the protocol describes how existing connections are relayed to the new location. Although CORBA mainly uses TCP/IP at the transport layer, it is possible to connect two ORBs using a different transport protocol, as long as it is reliable and connection oriented. In the case of wireless CORBA, the terminal and the access bridge can use a dedicated transport protocol for the connection between them, depending on the physical layer that is used, such as InfraRed, BlueTooth, IEEE 802.11, or GSM. Furthermore, different access bridges can used different physical layers, thereby allowing network roaming, where a mobile terminal selects the network technology depending on the situation.

In order to keep track of the location of the mobile terminal, the current access point is registered at a home location agent of the terminal, such that it can redirect the network traffic to the correct access bridge (resembling the home address in MobileIP[9]). It is possible to have a terminal without a home location agent, although it has implications for the lifetime of the services on the terminal, as the address of the access bridge is used as a temporary home location agent.

By default, the mode of communication within CORBA is synchronous, where every method call waits for a proper reply or exception. However, an application can use event channels to have asynchronous communication. For example, in wireless CORBA, event channels are used to send mobility events whenever terminals connect to or disconnect from their access bridge.

The available services can be discovered using the standard CORBA method, which is the use of a naming service to register or retrieve references to service objects. Additionally, each access bridge provides a method to retrieve the services which are available at that access bridge. After a client discovers a service, it can access it by calling the methods available in the API for that service. This requires that the client has a priori knowledge about that API, unless the client supports all the dynamic features of CORBA, like object construction and method invocation based on the interface definitions retrieved from the CORBA interface repository. However, the dynamic features are usually not available in highly optimized ORBs.

Within the Vivian project[10], wireless CORBA is used as the middleware platform to access services from a mobile phone. One of the components within Vivian enables remote user interface(RUI) services, where a static application on the terminal is used to access a RUI service. RUI is based on the data driven interface of HAVi[11], the protocol to connect audio and video devices. After a terminal discovers a RUI service, it subscribes itself to the service and receives a user interface description in XML format. The terminal renders this user interface to allow the user to interact with the service. User actions result in method invocations on the service, which returns XML descriptions that define how the interface should be updated. That is, the interface is incrementally updated instead of replaced completely, which requires less resources and improves the user experience. Furthermore, due to the subscription, the service can send notifications to the terminal to update the user interface, for example to reflex state changes in the service that is accessed. The main advantages of this way of interaction are the static API interface of the service (every RUI service provides the same API, only the UI description differs), no downloable code (no VM is required), no service knowledge on the terminal (the terminal only displays the interface) and device independent (the terminal can decide how to present the interface description, possibly in combination with style sheets). One of the disadvantages is that you can not use it for machine-to-machine services, as the user has to interpret the meaning of the presented user interface.

The TVDevice and the UIClient, as described in the UPnP section, are also implemented as a RUI service on top of CORBA. The TVDevice defines an abstract interface for the five functionalities, where two different panels are used for the ControlService and the ScreenService. This interface is made available by providing it as a RUI service through the CORBA naming service.

The UIClient uses the CORBA naming service to discover the service, and will access the interface by subscribing to that service. The returned XML user interface description, which is similar to

```
<container id="c1" label="Control">
 <button id="onoff" label="On"/>
 <range id="ch" label="Channel"
  minvalue="1" maxvalue="35" valueset="3"/>
 <range id="vol" label="Volume"
  unitlabel="%" valueset="40"/>
</container>,
```

is presented to the user in a way suitable for the UIClient. The shown interface allows the user to access the functionality and sends actions to the TVDevice, which interprets them, adjusts its settings and returns the changes that have to be applied to the interface on the UIClient. For example, upon selecting the 'On' button, the UIClient sends an 'onoff' action, the TVDevice turns itself on, returns an XML description like

```
<changeattribute element="onoff"
```

`attribute="label" changeto="Off"/>`,
which results in a change to the button on the UIClient.

In the current setup, wireless CORBA is not used yet, as it requires significant changes to the internals of an existing ORB, both for the terminal bridge and the access bridge of wireless CORBA. The main difference of using wireless CORBA will be that the UIClient is able to determine when it connects to a different access bridge and use its local naming service to discover the available services automatically, instead of on an explicit request from the user. Also, the hand-over between access bridges might influence the performance and stability of the UIClient.

For demonstration purposes, a Java based UIClient was installed on a Psion Revo PDA, together with a Java ORB, which was selected after a survey of freely available ORBs (see [12]). The responsiveness of the user interface was sufficient, given the limited hardware resources, like memory, network and processor speed. It is expected that a C++ based UIClient and ORB would be both faster and smaller with respect to the total amount of resources used.

Unlike UPnP, wireless CORBA does not define how to broadcast available services within the network. Instead, it relies on the infrastructure already available within CORBA to publish services, like the use of the naming service. However, with the naming service, the client is required to regularly check for newly registered services, unless the naming service is connected to an event service, where terminals can subscribe to the appropriate event channel. In wireless CORBA, mobility events are used to indicate that terminals are connecting to or disconnecting from the network. This could be used as an indication that the set of services might have changed.

6 Conclusion

UPnP and CORBA use clearly different models for service discovery. In UPnP, the available services are regularly broadcasted by the devices, which makes it robust at the expense of additional network traffic. In CORBA, a central naming service is used, where services are registered and devices discover them when needed. In wireless CORBA, a naming service is used in combination with hand-over events, which allows the device to discover services local to the access bridge that is being used, thereby allowing the device to use different discovery strategies.

The discovery mechanism in UPnP does not scale to many devices since communication is in principle between all pairs. A better solution is to dynamically build and maintain a backbone structure and to perform advertisement and discovery through this backbone. We want to address this in future research. In addition we want to investigate alternatives for the naming issue.

For wireless CORBA, it is still unclear how services are easily discovered on other terminals that are nearby. Several approaches are possible, like a naming service with an event channel or the new multicast based CORBA protocol. Also the stability of the RUI service with respect to hand-overs is not properly tested yet.

References

[1] Lisa Phifer. The trouble with nat. *The Internet Protocol Journal*, 3(4), December 2000.

[2] P. van der Stock. Issues for zeroconf working group, 2001.

[3] P.A. Bernstein. Middleware, a model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996.

[4] P.J.F. Peters J.J. Lukkien, M.F.A. Manders and L.M.G. Feijs. An architecture for web-enabled devices. In *Proceedings of the 2001 International Conference on Internet Computing, Las Vegas*, June 2001.

[5] S.I. Kumaran. *Jini technology: an overview.* Prentice-Hall, 2001.

[6] www.upnp.org.

[7] Yaron Y. Goland et al. Simple service discovery protocol/1.0. Internet Draft, October 1999

[8] Borland, Highlander, Nokia, and Vertel. Wireless access and terminal mobility in CORBA. Technical Report dtc/2001-06-02, OMG, May 2001.

[9] C. Perkins. IP mobility support. Technical Report RFC 2002, IETF, October 1996.

[10] Vivian - opening mobile platforms for the development of component-based architectures. Online: `http://www-nrc.nokia.com/Vivian/`.

[11] HAVi - home audio/video interoperability. Online: `http://www.havi.org`.

[12] P.H.F.M. Verhoeven, J. Huang and J.J. Lukkien. Network middleware and Mobility In *Proceedings of the second PROGRESS workshop, Veldhoven*, October 2001.