Support for Ad-Hoc applications in ubiquitous computing

Svensson Fors, David

2006

Support for Ad-Hoc Applications in Ubiquitous Computing

David Svensson

Licentiate thesis, 2006

Department of Computer Science
Lund University

ISSN 1652-4691
Licentiate Thesis 7, 2006
LU-CS-LIC:2006-4

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: david@cs.lth.se
WWW: http://www.cs.lth.se/~david

Typeset using LaTeX 2ε

Printed in Sweden by Tryckeriet i E-huset, Lund, 2006

Abstract

This thesis presents work within the area of ubiquitous computing, an area based on a vision of computers blending into the background. The work has been done within the EU project PalCom that introduces *palpable computing*. Palpable computing puts a new perspective on ubiquitous computing, by focusing on human understandability.

The thesis goals are to allow for ad-hoc combinations of services and non-preplanned interaction in ubiquitous computing networks. This is not possible with traditional technologies for network services, which are based on standardization of service interfaces at the domain level. In contrast to those, our approach is based on standardization at a *generic* level, and on self-describing services. We propose techniques for *ad-hoc applications* that allow users to inspect and combine services, and to specify their cooperation in *assemblies*. A key point is that the assembly is external to the services. That makes it possible to adapt to changes in one service, without rewriting the other coordinated services.

A framework has been implemented for building services that can be combined into ad-hoc applications, and example scenarios have been tested on top of the framework. A browser tool has been built for discovering services, for interacting with them, and for combining them. Finally, discovery and communication protocols for palpable computing have been developed, that support ad-hoc applications.

Preface

This thesis is for the Licentiate degree, a Swedish degree between the MSc and PhD. It consists of an introductory part and four papers. The research papers included in this thesis are:

I. David Svensson and Boris Magnusson. An Architecture for Migrating User Interfaces. *Proceedings of NWPER'2004, 11th Nordic Workshop on programming and Software Development Tools and Techniques*, Turku, Finland, 2004.

II. David Svensson, Boris Magnusson, and Görel Hedin. Composing ad-hoc applications on ad-hoc networks using MUI. *Proceedings of Net.ObjectDays 2005*, Erfurt, Germany, 2005.

III. David Svensson, Boris Magnusson, and Görel Hedin. Discovery and communication protocols for palpable computing. *Submitted for publication*, 2006.

IV. David Svensson, Görel Hedin, and Boris Magnusson. Pervasive applications through scripted assemblies of services. *Accepted for publication at SEPS 2006*, Lyon, France.

Acknowledgments

First of all, I wish to thank my supervisors, Professor Boris Magnusson and Dr. Görel Hedin. Your support, ideas and experience have been absolutely vital for me during the thesis work.

The work has been carried out at the Department of Computer Science, Lund University, and within the PalCom project. Thanks to Torbjörn Ekman for kind help and valuable input, especially related to the PalCom Java compiler, and to Sven Gestegård Robertz for rewarding joint work on PalCom assemblies. Torbjörn Eklund has my gratitude for our cooperation during the first parts of the MUI project. I have learned a lot from protocol discussions with Jacob Frølund and Jeppe Brønsted, and Boel Mattson and

Brice Jaglin have given important feedback to the protocols in their master thesis project. I wish to thank all people in PalCom for creating an inspiring atmosphere that has really made me feel like a *palcomer*, even though the project is spread across six countries.

I would like to thank the people at the department for good company and interesting lunch room conversations. My room mates Richard Johansson, Jonas Wisbrant and Emma Nyman have made it a great working environment. Klas Nilsson deserves a special thanks for introducing me to the ARTES++ graduate school, where I learned about real-time systems and saw research environments at other universities. Anne-Marie Westerberg, Lena Ohlsson, Anna Nilsson, Tomas Richter, Peter Möller and Lars Nilsson have been very helpful with practical things. You really are the backbone of the department.

I am grateful towards my football friends in Stora Harrie IF for providing another world, outside the university, where we focus on different things. My deepest thanks to my family for always supporting me, and to Emma for your love and support, and for your way of looking at small and big things in life.

Contents

Introduction

1 Background

The vision of *ubiquitous computing* was introduced by Mark Weiser in 1991 [27]. When that vision is realized, computation blends into the environment: computers are there to assist us when we need them, but do not require constant attention. This is a shift from the focus on desktop computers, towards computers of many different form factors. The ubiquitous computers may be virtually invisible, such as wearable computers [21] or computers in furniture [13], but they may also be handheld devices, such as PDAs or mobile phones, or larger devices, such as wall-sized displays. The important thing is that they are at hand when we need them, but disappear from human attention when not used.

Ubiquitous computing takes advantage of the ongoing rapid improvements in areas such as network technology and embedded systems. A key factor is the increasing number of devices that use wireless communication. Wi-Fi, Bluetooth, and similar technologies let these devices connect and form local ad-hoc networks, independent of a central network infrastructure. In these networks, services can become available to users when needed. As an example, consider a user that carries his handheld computer and comes into the vicinity of a particular device. The device might be a DVD player in his home, a ticket vending machine at the train station, or a printer at the office. Thanks to the wireless communication, services from these devices can be brought to the handheld computer. The services can be presented on its screen, and the user can interact with the devices remotely through the handheld.

There are two other terms, *pervasive computing* and *ambient computing*, that are used for the same vision as ubiquitous computing. In this thesis, the three are treated as synonyms. They all convey the sense of computers being in the background, everywhere around us. In addition, the concept of *ambient intelligence* has been introduced. This denotes a vision of ubiquitous computers acting more autonomously, making intelligent decisions and thereby providing enhanced user experiences [1].

The work in this thesis targets problems within the area of ubiquitous computing, towards the following overall goals:

Non-preplanned interaction In order to make adequate use of services in a ubiquitous computing context, special preparation of personal devices, such as handhelds, must not be needed each time you want to use a service. Instead, services should ideally just emerge on the handheld, ready for immediate use.

Ad-hoc combinations It should be possible to combine previously unknown services into new applications, in order to make use of functionality that is not given by any of the individual services themselves.

These observations are particularly true for ubiquitous computing systems, compared to systems in a traditional setting. That has to do with the scale of the systems. Like has been noted, e.g., in [9], ubiquitous computing systems can be expected to offer many more services than what are available in current networks. The rate at which new versions of services will be offered, and at which completely new services will become available, will also be much higher. Therefore, unnecessary configuration for each new service, or for each new version of a service, would become too much of a burden. It would also be very beneficial to be able to combine services, instead of having to wait for a dedicated service with the wanted combined functionality. As a common factor here, there is a challenge of *interoperability* between services.

The rest of this introduction is structured as follows. In the next section, we will discuss how service interoperability has traditionally been sought. Section 3 goes into more detail about our objectives, and Section 4 presents MUI and PalCom, the projects within which we work. Then follow Section 5, with presentations of the four papers that are the main part of the thesis, and Section 6, where we list what we see as the main contributions of our work: among them an architecture and a framework, supporting non-preplanned interaction and ad-hoc combinations of services. Section 7 contains an evaluation of the implementation so far, and Section 8 relates to other work in the area. Section 9 concludes the introduction, and points out directions for future work.

2 Traditional approaches to interoperability

The traditional approach for achieving interoperability between services in networks is standardization *at the domain level*, i.e., specifically for the application area where the services are used. Jini, UPnP, and Bluetooth all rely on this, in different ways. In Jini [26], Sun's technology for network services, a client program obtains a proxy (Java) object from a service, and

invokes operations on that proxy object for interacting with the service. Therefore, in order to write a Jini client program, you need to know the type, or interface, of the proxy object. For example, in order to be able to use a printer, you need to know the exact type of printer proxy objects, including the names and parameter formats of all the operations you can perform on a printer. This means that standardization of service types is needed, so that independently written service and client programs can interoperate. The Jini community has started a process for standardizing common service types. Up to this point, that process has resulted in a standard type for printers, but no standards for other domains, except standards closely tied to the core of the Jini technology [14].

In UPnP [24], a set of protocols for networked devices managed by the UPnP Forum, devices are categorized into different classes. Domain expert committees work out standards for devices and their services. This process has resulted in UPnP standards for printers, scanners, lighting controls, and digital security cameras, among others [23].

For Bluetooth [4], the specification for short-range wireless communication standardized as IEEE 802.15.1, there are specifications for a number of different *profiles*. These profiles specify protocols and procedures that a device must follow in order to be profile-compliant and certified to be interoperable. Some of the profiles are domain-independent, but several are domain-specific, e.g. the profiles for audio/video remote control, for phone book access, and for basic printing [5].

While this standardization work comes from a very real need—making services and devices from different manufacturers interoperable—there are problems inherent in the approach. With domain-specific standards, a quite static situation is bound to arise. In a ubiquitous computing setting, standardization processes cannot possibly keep up with the pace at which new kinds of services arrive. Furthermore, it is impossible to combine new services with old ones, unless they follow the old standards, at the level of individual service operations. For ubiquitous computing, we see a need for more dynamic combinations. This is the motivation behind the work in this project, as will be discussed in the following section.

3 Thesis objectives

The overall goal in this thesis is to find techniques for enabling more flexible, *ad-hoc*, use and combination of services in ubiquitous computing networks. This way, we intend to help making non-preplanned communication and interaction more feasible than today, and to ease the process of integrating new devices and services into existing systems. The thesis reports on our results so far, and points out directions for future work.

We aim at supporting what we call *ad-hoc applications*. These are applica-

tions that are put together for a special situation, using services available in the current environment. The ad-hoc applications range from simple set-ups to more complicated:

Remote control An important basic case is when a user interface for one service is migrated to another device. The user interface is rendered on that device, and the user can interact with the service remotely. This gives easier access to services, and is useful for interacting with services on devices with no or limited input/output capabilities.

Simple assemblies A second fundamental case is when a user chooses to establish connections between a set of services, and saves this set-up for later re-establishment. We refer to the saved set-up as a *simple assembly*. This facilitates repeated use of a set of services.

Scripted assemblies Building on the first two, it is possible to add coordinating logic to a set of services in a so called *assembly script*. This way, the ad-hoc application can become more than the sum of its parts.

Software services Finally, for including more advanced functionality than what can be specified in an assembly script, it is also possible to incorporate services written in a general-purpose programming language, such as Java, into an ad-hoc application. We refer to these services as *software services*. The name reflects that these services are not tied to the hardware of any particular device, but built for performing computations in assemblies.

Putting together the different kinds of ad-hoc applications requires different levels of programming skill. The first two should be possible to set up using simple operations in a graphical tool, while the last one requires knowledge about a general-purpose programming language. Our goal is to make the third one, constructing an assembly script, reasonable to manage for an end user.

Building further on this, it would be useful if the assembly script could also specify a user interface for the whole ad-hoc application, capturing aspects that are not covered by the user interfaces of the individual services themselves. In a similar way, the script could define a new service, a so called *synthesized service*, offering combined functionality to other devices in the network. This, and issues about versioning and automatic updating of assembly scripts, are discussed in Section 9, about future work.

The *assembly* is a fundamental part of our approach. This is an entity that defines an ad-hoc application by coordinating a set of services. The assembly is *external* to all the services, which is important for supporting the demands of ad-hoc composition: this is what makes it possible to adjust the ad-hoc application after changes in service interfaces, and to include new services after the initial construction of the application.

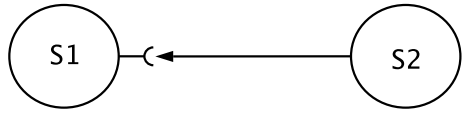


Figure 1: Traditional approach.

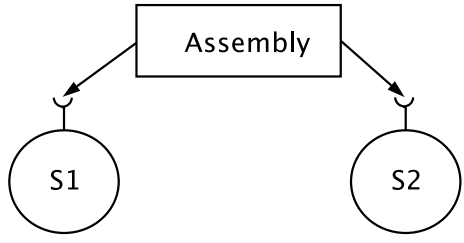


Figure 2: Our approach.

A metaphor with wires and connectors can be used for illustration. The traditional case of interoperation between two services is shown in Figure 1, where the service S_2 uses service S_1. Here, S_1 has a female connector, representing its service interface. S_2 connects directly to it, using a wire with a male connector that has been constructed to fit with S_1. Interoperability with S_1 is built into S_2. In our approach, shown in Figure 2, both services have female connectors. It is the assembly in the middle that connects to both services using male-connector wires. The assembly works as an adaptor, and the advantage is that when either S_1 or S_2 changes, it is possible to change the adaptor, instead of changing the other service. It is not so that both services must conform to a standard that was established when the first one was built.

For this arrangement to be beneficial, it is of course important that it is easier to construct and change the assembly than to rewrite the services (if rewriting the services is possible at that time). In general, the difficulty of changing the interaction depends on the complexity of the included services—adapting to a new standard for streaming video, e.g., is probably a complex task. As indicated above, we aim at enabling end users to work with a large portion of these assemblies. For the most complex cases, we rely on support for incorporating a software service, written in Java or a similar language, into the assembly.

The idea of having an assembly in the middle can be compared to the Mediator design pattern [10, p. 273]. The Mediator pattern gives similar advantages of looser coupling, and the possibility to vary the interaction between objects independently.

Concretely, the objectives of this thesis work is to demonstrate the usefulness of the concept of ad-hoc applications, by developing

- a framework for creating services that can be combined into ad-hoc applications,
- prototype services on top of the framework,
- tools that can be used for manipulating the ad-hoc applications, and
- protocols for discovery and interaction between the services and tools.

Regarding the protocols, we would like to stress that this is not about developing protocols at the domain level. Instead, the protocols we develop are for discovery and interaction at a *generic* level: how to make a service known in the network, how to announce its service interface, and how to format messages. The service interface consists of a description of what messages the service can send and receive. The interpretation of the contents and meaning of the messages is left to the constructor of the assembly.

There are different kinds of qualities that are important in a system such as ours. These will be discussed in more depth in Section 7 on Evaluation. One kind is the *usability* qualities: it must be as intuitive as possible to inspect the descriptions of services, for understanding what the services do, and to combine services into an ad-hoc application. Other qualities are *efficiency* and *responsiveness*: the protocols must permit implementation of services in resource-constrained environments, it must be possible to handle large numbers of devices and services, and the user must become aware of changes quickly. Like in all ubiquitous computing systems, the qualities of *security* and *privacy* are also important. These have not been our focus, though, and we rely on mechanisms in lower layers for keeping data secure and private.

4 Migrating user interfaces and palpable computing

Our work started in a project called MUI, which stands for Migrating User Interfaces. The name reflects our initial focus on the migration of user interfaces between devices, as mentioned above. Since then, our scope has broadened, and we now also target larger compositions of services, and interaction between services at a programmatic level.

MUI was originally started as a project with funding from VINNOVA, the Swedish Agency for Innovation Systems [25]. Our work is now part of the EC-funded IST project PalCom [19]. PalCom introduces *palpable computing*, a new flavour of ambient computing. The project at large seeks

Invisibility *complemented with*	**visibility**
Construction	**de-construction**
Heterogeneity	**coherence**
Change	**stability**
Scalability	**understandability**
Sense-making and negotiation	**user control and deference**

Table 1: The PalCom challenges.

to make ambient computing systems more understandable for humans. This is done by trying to meet a number of challenges, listed in Table 1. These challenges are formulated as pairs of complementary properties. For each challenge, the property on the left is one traditionally sought in ambient computing systems, while the property on the right is added by PalCom for achieving better human understandability. Palpable computing is about balancing these pairs. For the work in this thesis, the most important of the PalCom challenges are balancing invisibility with visibility, and finding ways of allowing construction and deconstruction of systems at appropriate levels.

The two main objectives of the PalCom project are to design an open architecture for palpable computing, and to develop a conceptual framework for it. The latter is needed for understanding the specifics of what *palpability* means, and for being able to talk about it. The PalCom objectives span a wide area, across several disciplines. Consequently, there are people involved from computer science, interaction design, industrial design, ethnography, and sociology. These researchers come from eleven academic and industrial partners, in six European countries. The work is carried out both as software design and development, and as construction of physical prototypes. The software work is resulting, e.g., in a new virtual machine, and in a communication model for palpable computing. The prototypes are used for evaluating and giving input to the open architecture and the conceptual framework. They are tested out in the field, in cooperation with people representing the anticipated end users. In this prototyping work, there are subprojects aiming at support for landscape architecture field work, for personnel at the site of a major incident, for women in their contact with hospitals during pregnancy, for rehabilitation of hand-surgery patients, for training of children needing physical-functional and cognitive rehabilitation, and for treatment of premature children in an incubator.

For the work reported in this thesis, the scenarios in PalCom both serve as inspiration, and give concrete requirements for ad-hoc applications and for the protocols developed.

5 Papers

This section briefly introduces the four co-written papers that constitute the main part of the thesis. The author of the thesis is the primary author of all the papers, and the main implementer of the developed software. The ideas behind the papers have been formed in collaboration with the co-authors.

5.1 Paper I: An Architecture for Migrating User Interfaces

The first paper presents the initial architecture of the MUI system, and the implementation of a framework for building MUI services. The focus is on the first kind of ad-hoc applications mentioned above, *remote control*, where the user interface for a service is migrated to another device, typically with better input/output capabilities, and rendered there. A protocol for discovery is presented, with an XML format for messages, and an example scenario is described where audio equipment is controlled from a handheld device.

It is the user that connects suitable services, either based on information about the MIME types of the data the services can handle [12], or by initiating a user interface migration. In the user-interface case, the client device does not need to be prepared in advance for the exact messages to be exchanged with the service, because they are provided by the service in the user interface description, and the rendering of a user interface is generic. The intention is to make it possible to interact with minimal or no preparation in advance.

5.2 Paper II: Composing ad-hoc applications on ad-hoc networks using MUI

Paper II introduces assemblies that coordinate services, and a dual role for service descriptions as programmatic APIs (in addition to the role as sources for rendered user interfaces). An implemented MUI browser is described. The browser can be used for discovering devices and services, and for interacting with them and combining them. The MUI implementation is evaluated according to the so called *palpable qualities*, i.e., how well it supports PalCom challenges listed in Table 1.

The paper contains two example scenarios that have been implemented in the MUI framework. One is a distributed slideshow scenario, where a presentation session is set up and performed in a smooth way using MUI, and the other is the *SitePack* scenario from PalCom[1], where landscape architects set up an ad-hoc application for tagging camera images with GPS

[1]The scenario referred to in the paper has since been renamed to *GeoTagger*.

coordinates out in the field. The second kind of ad-hoc applications, *simple assemblies* (saving a set of connections for later re-use), becomes useful in the slideshow scenario. The SitePack scenario has been implemented using initial versions of *scripted assemblies*, with coordination by an assembly script, and delegation of complex calculations to a *software service*.

5.3 Paper III: Discovery and communication protocols for palpable computing

Paper III presents discovery and communication protocols developed within the PalCom project. These are much refined and extended, compared to the discovery protocol presented in Paper I. One of the purposes of the protocols is supporting ad-hoc applications. That goal, and other requirements encountered in the PalCom prototyping work, has formed the features of the protocols. One feature is *device awareness*, having devices visible at the top level in the discovery process. *Announcement heartbeats* and *on-demand service discovery* are techniques used for limiting the bandwidth required by the protocols.

The implementation of the PalCom communication components uses these protocols. The components run both on the Pal-VM, the virtual machine developed within PalCom, and on the JVM. They are used in our implementations of services, and in our work on assemblies. The current implementation runs on top of UDP, but the protocols are prepared for other transport protocols, such as Bluetooth.

5.4 Paper IV: Pervasive applications through scripted assemblies of services

The last paper goes into more detail about assembly scripts. It is shown how assemblies separate the interoperation between services from the services themselves, and motivated why we want to use a scripting language, rather than a general-purpose programming language. Different types of intended usages of the scripts are discussed (by end users, by expert users, by tools), and also different assembly representations (XML, concrete syntax, abstract syntax trees, tool-specific representations).

Examples of assembly syntax are shown for the RemoteSlideShow and GeoTagger scenarios. The assembly descriptor specifies sets of devices, services and connections that are included in an ad-hoc application. For the more advanced assemblies, *scripted assemblies*, it also specifies a set of synthesized services, and a script with coordinating logic. The script is written as an event handler, and the actions possible in the script are to send messages to services, and to store values in local variables. The paper discusses how static and dynamic constraints can be checked in a browser, at assem-

bly development time and at run-time.

6 Contributions

The following are what we see as the main contributions of the work in this thesis:

- *An architecture and a framework* for building services that can be combined into ad-hoc applications. This is presented in Paper I, covering *remote control*, and in Paper II, covering *simple assemblies* and initial versions of *scripted assemblies* and *software services*.
- *A browser tool*, that can be used for working with ad-hoc applications. The browser is presented in Paper II.
- *Protocols for discovery and communication*, supporting ad-hoc applications. The protocols are described in Paper III.
- *An assembly script language*, in an initial version, as presented in Paper IV.

7 Evaluation

Looking back at our work in the project so far, we can evaluate both the concept of ad-hoc applications itself, and our implementation of it. In this section, we will present how we have performed this evaluation, and the results we have got.

As mentioned above, our overall goals are to allow for non-preplanned interaction, and for ad-hoc combinations of devices and services. Important qualities for our implementation are usability and efficiency, and we have looked specifically at the palpable qualities.

7.1 Experimental validation

Our main method of evaluation is experimental validation. The implemented framework has been evaluated through continuous use: with simulated devices for trying out scenarios, and in the implemented browser. In Lund, we have built a number of scenarios on top of the framework, including GeoTagger and RemoteSlideShow. For the other groups within PalCom, the code of the implementation is available in the project's shared code repository. The parts that have been used the most outside Lund are the communication components, presented in Paper III. There are also

plans in PalCom to release parts of the code publicly, under an open source license.

The concept of ad-hoc applications is clearly promising. The implementation has worked well, and we are confident that it will be a good foundation when we develop more elaborate tools and scripted assemblies in our continued work. We have been able to combine services that are not constructed according to commonly agreed-upon service interfaces, and use them together. This has been done both with matching based on the type of data handled, and with assemblies written to make use of available service descriptions.

7.2 Usability

From a usability point of view, the functionality of the browser is most important, because this is where services are discovered, interacted with, and combined. The current browser has been implemented to run on a simulated handheld device, with a rather small screen, so it uses a small set of graphical widgets. It is possible to connect services in an intuitive way using a drag-and-drop interface, but the user must switch between different tabs for viewing discovered devices and services, for manipulating connections and assemblies, and for direct interaction with services. That gives decreased usability, but these limitations could be overcome by making a browser version for full-screen mode on a laptop. This is the intended use of a PalCom Eclipse plugin that is currently being developed in Lund (see [8] for information about the Eclipse Integrated Development Environment). There is also ongoing work within PalCom on visualizing connected services graphically. That service graph is best viewed on a larger screen, too.

The Eclipse plugin, and similar browsers, will provide more advanced script-editing facilities. A browser for a mobile phone or a PDA, like the one discussed above, will only offer more basic functionality, such as direct interaction with services, and establishment of *simple assemblies*. This span in handled complexity is also reflected in the groups of users that we foresee. Assemblies are intended to be handled by end users, but there will be more groups of users than just service developers and end users. Within the end-user group, there are naturally some that have more experience with computers, while some would rather not make any configuration by themselves. It is people in the former category that are most likely to write an assembly script using an advanced browser. Therefore, it is also important that written assemblies can easily be shared with others, as will be discussed in Section 9, about future work.

7.3 Efficiency and responsiveness

Efficiency and performance considerations are important both for the services and browsers that run on devices, and for the protocols developed. For the interaction with users, responsiveness is important, and the protocols must tolerate unreliable networks and, preferably, large numbers of devices.

The performance of software running on the devices is most critical for smaller devices, offering limited memory and computing power. The PalCom communication components that implement the protocols run on the Pal-VM, which is intended for small devices. For these devices to provide a PalCom service, an advantage is that they will generally only need to respond to requests in the protocols, often with a fixed description of their services, and handle a small set of commands. The smallest devices need not support, e.g., execution of assemblies or rendering of user interfaces. In a current project in Lund, an Axis network camera [2] is being equipped with PalCom services, implemented in the C programming language. A part of this work is an investigation of how the protocols can be supported on a small device.

The protocols initially target networks of limited physical range, in the vicinity of a single person. Therefore, they do not need to scale up to, say, thousands of devices. As discussed in Paper III, it is difficult to make fair quantitative measurements for comparing to other protocols. We have used the mechanisms of *announcement heartbeats* and *on-demand service discovery* for limiting the network traffic. In order to cope with unreliable networks and transient devices, frequently joining and leaving networks, the protocols are based on asynchronous communication.

7.4 Palpable qualities

The palpable qualities are related to the usability qualities. We evaluate our implementation of ad-hoc applications as follows, in relation to the palpable challenges listed in Table 1:

- The challenge of *invisibility, complemented with visibility*, is about finding a balance of what should be made visible for the end user, and at what times. The discovery of connections is one factor supporting visibility. Another factor is the *device awareness*, that devices are visible at the top level during discovery, which is important, because the physical devices are significant for the users in pervasive computing settings. As a third factor, the involvement of users in the process of constructing ad-hoc applications in itself facilitates increased visibility and understanding. Finally, assemblies are announced as services themselves, which means that they can be composed hierarchically.

When such an assembly does not work as expected, it is possible to open it up level by level, looking for the reason of the failure.

- For the PalCom challenge of *construction and deconstruction*, the assembly concept is central. By connecting a number of services in an assembly, the end user can construct new applications. Existing assemblies can also naturally be deconstructed, and their parts inspected. It is an important goal that saving a set of services as an assembly is as easy as possible for the end user. In the current browser, that goal is met quite well for simple assemblies, and we focus on this in our development of scripted assemblies. The hierarchical composition of assemblies, mentioned above, is very relevant also for the construction and deconstruction challenge. It is equally important that users can establish and manipulate connections between pairs of services that both run on devices other than the browser device. This is supported by the small RemoteConnect protocol. For the same reason, connections are made discoverable through the discovery protocol, in addition to devices and services.

- One way of balancing *heterogeneity and coherence* is by putting the standardization of protocols at the right level. We have a common protocol for discovery and basic communication, but no standardized protocols at the domain level. This means that devices and services are discovered and described in a coherent way, but the heterogeneity at the domain level is not restrained.

- The high degree of *change* in pervasive computing systems is moderated in palpable computing by mechanisms for increased *stability*. As one step in this direction, we let the user be involved when a service interface has changed, and assemblies have to be updated. This challenge will also be important to keep in mind in our future work on more dynamic bindings to services, as discussed below in Section 9.

- The challenge of *scalability and understandability* becomes an issue for the presentation of discovered services. In an environment with many available services, the user must not have to deal with all services at once. For making this easier, we have support for grouping services on devices. There are mechanisms in the browser for narrowing down the number of possible end points while a connection is being established, and hierarchical assemblies also support this challenge.

- When it comes to the challenge of *sense-making and negotiation, complemented with user control and deference*, we tend to focus on the latter part. As an example, services can be explored directly by the user before building an assembly, so problems can be identified.

8 Related work

There is an abundance of previous and existing systems and technologies for communicating services, running on devices in wired and wireless networks. In this section we will relate to a number of these, from the perspective of our ad-hoc applications.

8.1 Jini and UPnP

Jini [26] and UPnP [24] are important examples of network service technologies that have taken a route of domain-level standardization. This does not necessarily imply that ad-hoc applications cannot be built on top of those, but we have chosen not to. For the Jini case, the main reason is the tight connection to the Java language, with Java objects moved across the network. In practice, that puts demands on all participating devices of running a JVM, which is too limiting for smaller devices. There is a notion in Jini of "surrogate devices", devices providing a JVM for the benefit of less powerful ones, but that makes the smaller devices dependent on their surrogate. Jini is also tied to Remote Method Invocation (RMI, [22]), a scheme for making method calls on objects across the network. The problem with RMI is that a method call is synchronous, forcing the caller to block until a response has been returned from the callee. There is also an event mechanism in Jini, but for simple messages, RMI is used. With the unreliable nature of networks in pervasive computing settings, we have instead opted for a scheme with only asynchronous messaging between services.

UPnP is more similar to our architecture. It is specified as a set of protocols, not dictating a specific implementation on the devices, and the messages are in XML, like we have also chosen. One difference is that UPnP focuses on IP networks, while the PalCom protocols are intended also for other lower-layer protocols, like Bluetooth. Another difference is that UPnP uses SOAP over TCP as its standard form of simple communication. With SOAP over TCP, the sequence of messages is like for RMI, with a blocking caller. In spite of these differences, it would probably be possible to build ad-hoc applications on top of UPnP. When doing that, the format of service descriptions would have to be extended, allowing services to describe themselves in a way suitable for connecting them through assemblies.

8.2 Web browsing technologies

Another obvious set of technologies to relate to is those of the Web. With its explosive growth during the past fifteen years, the Web has an enormous user base. There are Web browsers available in devices of many different sizes and form factors. This means that for realizing *remote control* ad-hoc

applications, it seems like a natural choice to interact with locally available devices through a Web interface. This was the approach of Cooltown, an early pervasive computing project that put Web servers into things, for bringing the Web to the physical world [16]. A fundamental limitation of the Web technologies, though, is that they are based on pull mechanisms. From a user interaction perspective, that burden has been somewhat relieved with the introduction of the Ajax and related technologies [11], which let parts of a Web page be updated without reloading the whole page. Still, the messages over the network follow a request-reply scheme, initiated by the client. This is different from how our services communicate. After a service description has been transferred in our system, there is a true peer-to-peer situation between two communicating services, with messages flowing in both directions.

8.3 Web services

It is also interesting to compare Web service technologies to simple and scripted assemblies. Web services is a set of technologies enabling communication between applications residing on Web servers, and not only between a browser and a server. There is a language, WSDL (Web Services Description Language [6]), for describing Web services. In correspondence to our assemblies, there is also a language for describing *choreographies* between Web services (WS-CDL, Web Services Choreography Description Language [15]). A choreography is similar to an assembly in that it coordinates a set of services, and that it is external to all of them. Being external to all of the services could give the advantages of separating the interoperation from the services, like we want for allowing adjustments to changes in services. But, that is not the focus of Web services. Instead, the choreography is more like a contract that is signed by a number of cooperating service providers, beforehand. There is also no notion of physical devices in Web services, which is important in pervasive computing (*device awareness*, as discussed in Section 7.4).

8.4 The Semantic Web

In relation to Web services, the effort to build a Semantic Web should also be mentioned [3]. Tim Berners-Lee et al. recognized that for making it possible for a computer to make use of Web servers, the structure of a Web service must be made known to the computer in a format different from that which humans read. At the same time, they saw a risk in standardizing at the level of Jini or UPnP, considering that to be too much "at a structural or syntactic level". The Semantic Web approach, instead, builds on the creation of a global *ontology*, formed by connecting many smaller ontologies. An ontology is a collection of information that defines classifi-

cations and relationships among terms. When constructing a service, the programmer describes the service, using the OWL language (Web Ontology Language [7]). The OWL description can then be used by computer programs, together with the global ontology, for finding out what the service does. This, again, is different from the approach we have followed. Our main objection is that the global ontology suffers from similar problems as the domain-level standardization. The classifications of things into the ontology will always lag behind. Instead, for the situations we target, we want the user to be able to make ad-hoc modifications in the assembly.

8.5 Approaches that avoid domain-specific standards

The problems of domain-specific standards have been recognized before by pervasive computing researchers, and solutions have been proposed that are more closely related to our approach. The most similar, of the ones that we have seen, is Obje from Xerox PARC[2] [18]. Obje shares with MUI the idea of having a very generic protocol at the foundation, and then letting the user be involved in sorting out the specifics of how services should interact. The terms used for this are *recombinant computing* [9] and *serendipitous integration* (the ability to integrate resources in an ad-hoc fashion). Features in common with MUI are the user-in-the-loop interaction and the generic interfaces. A difference is the use of mobile code, where a proxy object is moved to a device, for "teaching" it how to interact with another one. We do not make use of mobile code, for reasons discussed above, but put interoperation logic in the assembly instead. This reflects the partly different focus in Obje, where end users only perform very simple operations by establishing connections, while we also support a programmatic perspective in the assemblies.

Ponnekanti et al. at Stanford University have spotted the same problem, but implemented an approach that is more different from ours [20]. Their approach to avoiding standardized service interfaces is based on dynamic downloading of stubs and adapters for services from directories available in the network. Adapters, or automatically generated chains of found adapters, are presented to the user as suggestions for how to combine a set of services. The authors stress the separation between identifying semantically compatible services, and the mechanics of how to combine them. They point out how Jini and UPnP have mixed these aspects, thereby requiring a single standard service interface for each domain. The main difference from our approach is that the pre-programmed stubs and adapters must be available in a central directory at combination time, instead of having an assembly that can be modified by the end user. Not to rely on this kind of central directory is one of the requirements that we put on our protocols, as will be discussed in Paper III.

[2]The project has also been referred to as Speakeasy.

8.6 Task-oriented approaches

For enabling a user to combine and use a number of discovered services, there are also *task-oriented* approaches, where the user formulates what he wants to do as a task description, and not as a combination of a specific set of services. One example is InterPlay [17], which is a middleware for integrating devices in a networked home. That domain is more narrow than the one targeted by MUI and PalCom, but still characterized by an increasing number of heterogeneous devices. The InterPlay designers let the user express his wanted functionality as a *pseudo sentence*, following a restricted form of English. The subject, verb and target device of the pseudo sentence are taken from descriptions available in that particular home, and the middleware handles the orchestration of devices accordingly. There is also a notion of *task sessions*, where the middleware handles the execution state of a task across several devices. Some features of the InterPlay system do not carry over directly to our situation. One is the presence of central directories of available devices and available content, that are used for building the pseudo sentence. Another one is the use of device attributes for automatically selecting the best device for a given task. As discussed above, we see the creation of such an attribute hierarchy as an obstacle in a more general setting.

9 Conclusions and future work

This introduction has given an overview of the research reported in the four papers that follow. The research has been carried out within the PalCom project, a project within the area of pervasive computing. The underlying theme is to allow for non-preplanned interaction and ad-hoc combinations of services. We argue that interoperability between services should be based not on domain-specific standardization of interfaces, but on interfaces defined at a more generic level, and on the possibility to manipulate the interoperation independently of the services themselves. This way, it is possible to combine services that were not created together.

We call our approach *ad-hoc applications*. The ad-hoc applications are supported by four mechanisms. *Remote control* means that a browser on one device renders a user interface for a service on another device, allowing direct interaction. *Simple assemblies* are sets of connections between services that are saved for later reestablishment. *Scripted assemblies* extend simple assemblies with a script, containing logic that controls the interoperation. Finally, *software services* are services written in a general-purpose programming language, that are included in an assembly for performing more complex tasks.

For remote control, the migration of user interfaces gives some of the fea-

tures that we want. The rendering of user interfaces is generic, so there is no need for preparation in advance. For assemblies, a key point is that the assembly is external to the services themselves. This gives looser coupling, and the possibility to adjust the assembly at a later time, without reprogramming the services.

A framework has been implemented for building services that can be combined into ad-hoc applications. We have built a browser application, and a number of example services. We have also designed and implemented the PalCom discovery and communication protocols, that support ad-hoc applications. Services announce descriptions of themselves, specifying a set of commands that can be used for interacting with the service.

The PalCom protocols have been designed to support transient devices, devices that come and go in the network as people move around and as network connectivity varies. An important feature in the discovery protocol is *device awareness*, i.e., physical devices can be discovered as such. There are also features for limiting the traffic in the network: *announcement heartbeats* is a scheme for announcing device information that combines announcements and heartbeats, and *on-demand service discovery* means a split between transfer of light-weight device/service information, and transfer of potentially bulky service descriptions.

In our continued work, an important part will be to design the assembly script language at the right level of sophistication, and with the right functionality. One feature that we discuss is to extend the support for *synthesized services*: letting the assembly itself specify a service description and function as a service. This service will typically provide combined functionality from the services that are included in the assembly. Another goal is to find good mechanisms for bindings, which means specifying in the assembly that one or more services or devices should be automatically substituted, according to some rule, in case they are not available. We will also experiment with support for sharing created assemblies with other users. This involves sending of assemblies between devices, and also migration of software components, which can be instantiated as software services. This development will trigger needs for versioning of assemblies and software components, and for update mechanisms.

Also in the future, our work will be driven by insights from the scenarios in the PalCom project, and we will go deeper into some of these for finding further requirements on ad-hoc applications. We will continue to develop more advanced tools, with support for editing of assembly scripts.

To conclude this introduction, we feel that our work so far has provided a good foundation for the realization of ad-hoc applications.

References

[1] E. Aarts, R. Harwig, and M. Schuurmans. Ambient Intelligence. In B. Denning, editor, *The Invisible Future*, pages 235–250. McGraw-Hill, 2001.

[2] Axis Communications. Network cameras. `http://www.axis.com/products/video/camera/index.htm`.

[3] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–44, May 2001.

[4] Bluetooth.com. The Official Bluetooth®Wireless Info Site. `http://www.bluetooth.com/`.

[5] Bluetooth.org. Specification - Qualification and Testing. `http://www.bluetooth.org/spec/`.

[6] Erik Christensen et al. *Web Services Description Language (WSDL) 1.1*. W3C, March 2001. `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`.

[7] Mike Dean et al. *OWL Web Ontology Language Reference*. W3C, February 2004. `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`.

[8] Eclipse.org. Eclipse.org home. `http://www.eclipse.org/`.

[9] W. Keith Edwards, Mark W. Newman, and Jana Z. Sedivy. The Case for Recombinant Computing. Technical report, Xerox Palo Alto Research Center, April 2001.

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

[11] Jesse James Garrett. Ajax: A New Approach to Web Applications. Technical report, Adaptive Path, February 2005.

[12] Internet Engineering Task Force. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, 1996. `http://www.ietf.org/rfc/rfc2045.txt`.

[13] Masaki Ito et al. Smart Furniture: Improvising Ubiquitous Hot-spot Environment. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03)*, pages 248–253. IEEE, May 2003.

[14] Jini.org. Jini Community Decision Process (JDP) Status. `http://www.jini.org/standards/status.html`.

[15] Nickolas Kavantzas et al. *Web Services Choreography Description Language Version 1.0*. W3C, November 2005. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/.

[16] T. Kindberg et al. People, Places, Things: Web Presence for the Real World. In *Proc. 3rd IEEE Workshop Mobile Computing Systems and Applications (WMCSA 00)*, pages 19–28, 2000.

[17] Alan Messer et al. InterPlay: a middleware for seamless device integration and task orchestration in a networked home. In *Proceedings of PerCom'06, the Fourth Annual IEEE International Conference on Pervasive Computing and Communications*, 2006.

[18] Obje Interoperability Framework, 2003. http://www.parc.com/research/projects/obje/Obje_Whitepaper.pdf.

[19] PalCom. Palpable Computing: A new perspective on Ambient Computing. http://www.ist-palcom.org/palcom-info.pdf.

[20] Shankar R. Ponnekanti and Armando Fox. Application-service interoperation without standardized service interfaces. In *Proceedings of PerCom 2003, the First IEEE International Conference on Pervasive Computing and Communications*, pages 30–37, 2003.

[21] Thad E. Starner. Wearable Computers: No Longer Science Fiction. *IEEE Pervasive Computing*, 1(1):86–88, January–March 2002.

[22] Sun. *Java Remote Method Invocation Specification*, 2003.

[23] UPnP™ Forum. UPnP™ Standards. http://www.upnp.org/standardizeddcps/.

[24] UPnP™ Forum. UPnP™ Device Architecture 1.0. Technical report, December 2003. Version 1.0.1.

[25] Vinnova.se. VINNOVA - Swedish Agency for Innovation Systems. http://www.vinnova.se.

[26] Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, pages 76–82, July 1999.

[27] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, February 1991.

Paper I

An Architecture for Migrating User Interfaces

David Svensson and Boris Magnusson
Dept. of Computer Science, Lund University, Sweden

{david|boris}@cs.lth.se

ABSTRACT

The MUI project looks at flexible ways of creating user-initiated connections between services in wireless networks. A central idea is to migrate user interfaces from controlled devices to devices with better input/output capabilities. The paper shows the different parts of the MUI architecture, and motivates design choices. An initial implementation and a framework for building MUI services are described.

1 Introduction

MUI (Migrating User Interfaces) is an architecture for services in wireless networks, where the user can connect and combine services in a simple, yet flexible, way. User interfaces can be migrated between devices, so the user can control several services from one device. MUI is developed as an ongoing research project at the department of computer science, Lund university. This paper presents the MUI architecture and the thoughts behind it. It focuses on the current state of the architecture, and on the implementation of the system.

MUI is initially designed to operate in networks of limited physical range, typically within a room, or even in networks formed between devices carried by a single person. This can be a very dynamic environment, where services enter and leave networks quite frequently, as people move around. This, in turn, puts requirements on the architecture: it must be smooth and simple to discover new services and connect to them. The diversity of equipment also requires the devices to interact with minimal or no preparation in advance. We think that the need for non-pre-planned communication is best fulfilled by defining interfaces at a very general level. Attempting to standardize the protocol for each interesting combination of services is, as we see it, not feasible—that process would be too complicated and time-consuming, as the number of services in the networks continues to grow. Instead, in the MUI architecture, a service should just present general-level information about the data it can provide and consume. It is up to the user to connect it to suitable services, with matching data types.

User interfaces are important in this context. To allow for the flexible connection of services, a service should be able to show an interface to the user, where the communication with another service can be controlled in more detail. It should be possible to show the interface on a device with suitable input/output resources, such as display and keyboard. Therefore, migrating user interfaces between services is a central concept in the architecture, and user interfaces have got special attention during our initial work. We view a user interface as a service among other services, but it has a special protocol associated with it, which will be presented in section 8.

The devices in the networks will often be very small, with limited memory and processing power. This gives further requirements: the architecture should not rely on facilities such as IP network connectivity or graphical displays on all devices, but allow a light-weight implementation. Our envisioned underlying technology for network communication is Bluetooth [2], even if the initial implementation with simulated devices uses IP, as will be discussed in section 9.

The paper continues with an example scenario, and a discussion about some previous work within this problem domain. Sections 4 through 8 look at different parts of the MUI architecture. In section 4, the fundamental

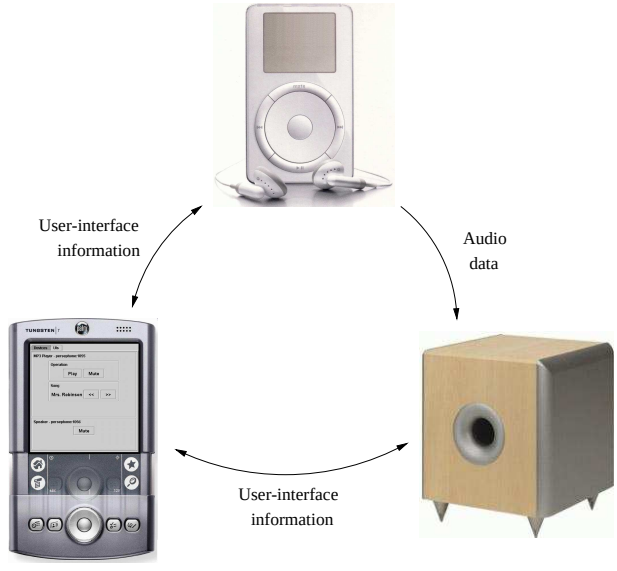
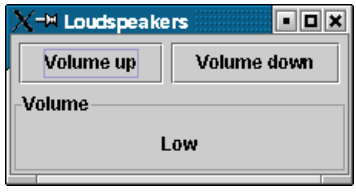


Figure 1: An example scenario with a handheld computer, an MP3 player, and loudspeakers.

structure of services and connections is presented. Section 5 deals with the discovery protocol, and section 6 with MUITP, the binary transport protocol for connections. Two protocols with XML messages, for connecting services with RemoteConnect and for representing user interfaces, are explained in sections 7 and 8. A presentation of the implementation, conclusions, and future work round off the paper.

2 A Scenario

Figure 1 illustrates an example scenario where the architecture is at work: with a handheld computer in his hand, and a portable MP3 player in his pocket, a user enters a room where a set of loudspeakers are in the corner. The MP3 player and the loudspeakers show up as services in a browser application on the handheld. The user can see that they match, and connects them by joining them in the browser. Now, the MP3 player sends its music to the loudspeaker. The volume is a little low, though, so the user chooses to control the loudspeakers by clicking their service in the browser. A user interface is moved to the handheld and shown. It may look as in figure 2 (a). The user presses "Volume up", the volume is adjusted, and he can enjoy the music. At the same time, the user interface on the handheld is updated

(a) 

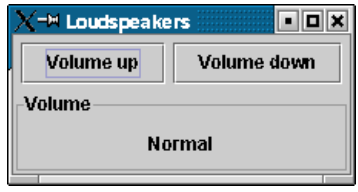
(b) 

Figure 2: A migrated user interface for loudspeakers, before (a) and after (b) adjusting the volume.

to that of figure 2 (b). If he wants to control also the MP3 player from the handheld, a user interface can be obtained for it in the same way.

Tables 1 and 2 show XML documents that are transferred in this scenario. The document in table 1, which describes the user interface, is what the loudspeakers send when connected to from the handheld. Table 2 shows the document that is sent when the volume has been changed, so the user interface can be updated on the handheld.

3 Previous Work

Jini from Sun [1] has targeted many of the same requirements as MUI, and could be used in the scenario above. It is an architecture where services register their presence at a lookup service, so they can be found and used by clients. The lookup service distributes proxy objects, which are downloaded to clients and used for communicating with the service. The client knows about the programmatic interface of the proxy, but does not need to know about its implementation in order to use it.

One problem we see in Jini is that the proxy interfaces are quite specific. Sun and partners are standardizing interfaces for printers, scanners, storage devices, etc. This means that clients have to be written for using a specific kind of service, and as new kinds services are invented, new clients have to be written. We hope to relax this in MUI, with general-level interfaces.

Another aspect to Jini is that it is heavily tied to Java. The proxy objects are Java objects, typically communicating with the service using RMI (Remote Method Invocation, see [13]). When considering small, embedded devices, this makes Jini services bulky. We see the need for allowing implementations in other languages, such as Smalltalk or C.

There is a specification for user interfaces in Jini, in the ServiceUI API [7]. The user interfaces are associated with a service, and are written to use the proxy object interface of that service. A problem, as we see it, is that the user interfaces themselves have specific programmatic interfaces and

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE UI SYSTEM "mui-ui.dtd">
<UI text="Loudspeakers">
  <Button text="Volume up" command="volumeUp"/>
  <Button text="Volume down" command="volumeDown"/>
  <Panel text="Volume">
    <Label text="Low"/>
  </Panel>
</UI>
```

Table 1: An XML document for the loudspeaker user interface. There are two buttons with commands attached, and a panel with an inner label.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE UIUpdate SYSTEM "mui-ui.dtd">
<UIUpdate element="/1/3/1" text="Normal"/>
```

Table 2: A UI update for the user interface of table 1. The syntax for the `element` attribute is a restricted form of the XPointer child sequence syntax (see [17]). In this example, the attribute refers to the label of the document.

different semantics. As new services are standardized, they are expected to come with new user-interface types, so clients will still have to be written against a specific service.

Web Services [16], the technology for application-to-application communication on the Web, is also interesting to look at in relation to MUI. There is a language, WSDL, for describing a Web service, by listing messages sent and received, and specifying message exchange patterns. The description has to be known to both the service requester and the service provider, together with some knowledge about the semantics of the service. The semantics can be encoded as different kinds of metadata, but, as noted in [18], current description technologies are not sufficient for describing the complete semantics of complex services.

The Speakeasy project at the Palo Alto Research Center [4, 5] introduces the term *recombinant computing* for an architecture where the user can combine functionality from several services into one. Like MUI, they have an approach of generic interfaces for letting the user combine services in new ways. Other key concepts in their framework are mobile code and *user-in-the-loop interaction*. Mobile code means proxy objects, like in Jini, that are downloaded to clients and executed there. This requires some platform-independent code, and they have used Java in their implementation. For discovery and user-interface they seem to have used Jini. It is unclear to

us how light-weight Speakeasy implementations can be. User-in-the-loop interaction means that the user should always be in control when connecting services, leaving it up to him to make sure the connection makes sense. The latter point, the authors claim, helps keeping the interfaces small and generic.

In a previous project [6], we implemented a remote-control scenario for two devices with real hardware, where Bluetooth was used for the wireless communication. There was a VCR, which could be discovered from a Palm handheld. A remote-control user interface was downloaded to the handheld, and the VCR could be controlled by clicking buttons in the interface. With the experiences from that project, we are now working with simulated devices communicating over IP—this makes it easier to shape the architecture for multi-device scenarios.

One thing we found in the previous project considered the representation of the user interfaces. There, they were contained in small Java applications, J2ME MIDlets [12], that were downloaded and installed on the client device. We did not use Jini, but our own protocol for discovering and migrating the interfaces. We felt that using Java applications for the interfaces was a bit heavy-weight, so we are now working with an XML representation instead. The experiences from the previous project will be further discussed in section 8.

4 Services and Connections

The architecture of MUI is based on connections between services. A service runs on some device in the network. The service may represent the whole device, such as an MP3-player service representing an MP3 player, but there may also be several services on a device. The latter would typically be the case for larger devices, such as laptops. This *device agnosticism* is also present in Jini: everything is services, both hardware and software [3].

MUI services implement the interface `Service`, shown in figure 3. Services can have subservices in a tree structure, which is useful for grouping services into logical units. When establishing a connection, one of the two parties must be a client. `Client` is a subinterface of `Service`, which means that clients are services themselves. The rationale behind this is the following: all clients offer a particular kind of service, namely the ability to connect to other services. The methods `connectToService` and `disconnectFromService` reflect this. Clients should also support initiation of connections over the network, using the RemoteConnect protocol (see section 7).

A service holds information about itself in a `ServiceInfo`. The information consists of the name of the service, a URL for connecting to the

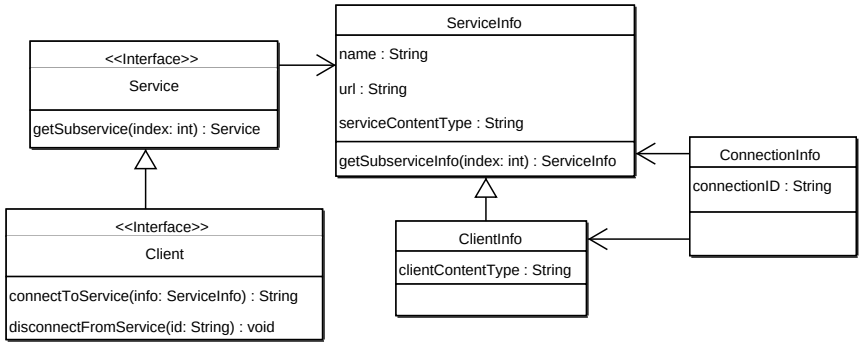


Figure 3: Classes and interfaces for services, clients, and connections. Not all aspects of the types are shown, only the most central.

service, and the content type offered by the service. The content type is the MIME type of the data transferred over connections to the service. `ServiceInfos` can be nested, reflecting subservice trees. Information about clients, in `ClientInfos`, hold an additional item: the content type accepted from services. This is used for determining whether a client can connect to a certain service.

When a connection is established, a `ConnectionInfo` is put together. It contains information about the two parties, and an ID identifying the connection. The ID can be used for disconnecting later.

MUI connections are like socket connections: bidirectional connections that stay up until either party closes them or until a network error occurs. As mentioned above, the data is of a certain MIME type. It is transferred in chunks, called *documents*, and it is normal that several, or many, documents are transferred over the same connection. The MUITP protocol has been defined for the binary transport, as will be discussed in section 6.

5 Discovery

For clients to be able to use services, information about the services must reach the clients in some way. This is handled by a discovery protocol. The information transferred by the MUI discovery protocol is in `ServiceInfos` and `ConnectionInfos`. The `ServiceInfos` can be displayed to the user of an application on a device, letting him browse available services and establish connections between matching client-service pairs. `ConnectionInfos` are for managing established connections—disconnecting them, e.g. The messages are in an XML format, for which DTDs have

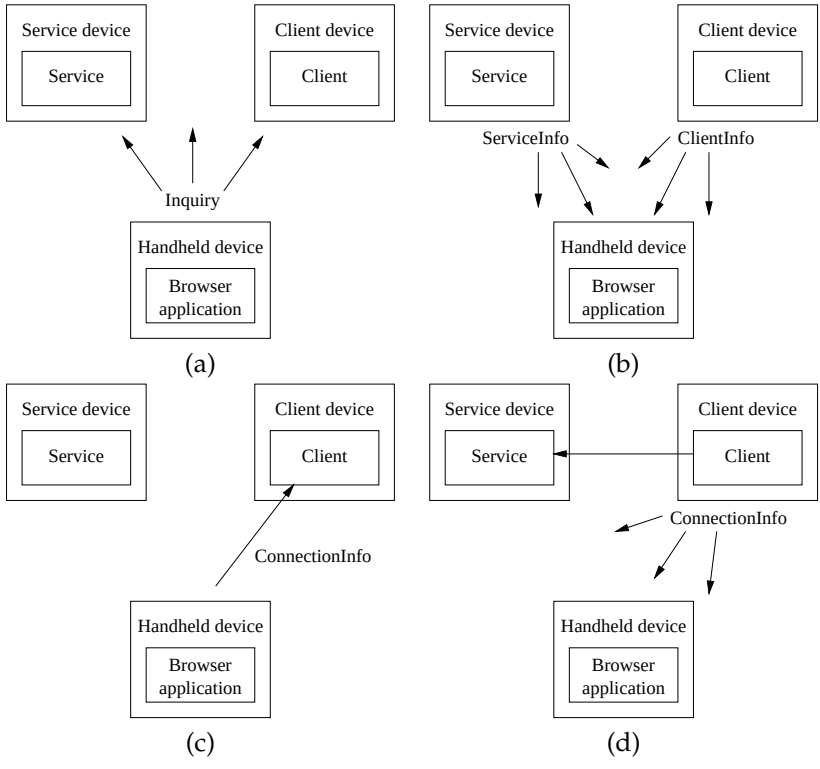


Figure 4: Discovering and connecting services.

been defined (see appendices A and B).

The MUI discovery protocol relies on having some broadcast mechanism. The existing implementation uses IP multicast, as discussed in section 9. The protocol has not yet been optimized to minimize the network traffic.

Figure 4 illustrates the process of discovering and connecting services. There are three devices: a service device, a client device, and a handheld device with a browser application. At first, the handheld broadcasts an inquiry (a). The service and client devices respond by broadcasting their information (b). In the browser application, the user can now see that the service and the client match, meaning that they can handle the same type of data. He chooses to connect them. A `ConnectionInfo` is assembled, and sent to the client using the RemoteConnect protocol (c). The client uses the service URL in the `ConnectionInfo` for connecting to the service, and starts receiving data. It also broadcasts information about the new connection (d).

The risk of *partial failure* is an issue for discovery. This is one thing that

makes distributed systems more complicated than non-distributed: there is always a risk that one node goes down in an unclean way, without having the time to inform the other nodes [3]. This could, e.g., be due to a software crash, power loss, or physical damage. From the other nodes, it can be hard to detect this, because it may look like a node is simply slow in response. There must be a way to handle devices that suddenly leave the network without sending out notifications, so that the lists on all devices can be updated. In Jini, *leasing* is used to remedy these problems: all resources that are used by other nodes are leased for a limited time. If the lease is not renewed regularly, the resource will be removed. This keeps old resources from filling up memory on devices, and gives a form of self-healing to the system. In MUI, we plan to implement some leasing mechanism, or to use a simpler scheme where devices periodically send out broadcasts, signalling that they are still alive.

6 MUITP

In order to transfer documents over a connection, a transport protocol is needed. We considered HTTP, but its request-response nature makes it inadequate for MUI, where any party should be able to initiate a transfer at any time[1]. Instead, MUITP was defined (MUI transport protocol).

The protocol works as follows. When a connection is established, both parties start reading from their input streams. At first, small headers are sent, where the non-client supplies the content type. The client leaves this header empty. After that follows a sequence of zero or more documents in both directions, with an arbitrary delay before and in between. The data of each document is preceded by a header, containing the length of the data. Any party may terminate the connection at any time, by simply closing it.

The definition of a new transport protocol leads to a new scheme for URLs. For simulated devices communicating over IP, a URL may look like

```
muitp://130.235.16.32:6427/
```

This URL can be used for connecting to a service listening on port 6427 of host 130.235.16.32. For a discussion about Bluetooth as the underlying protocol, and an example of a URL for Bluetooth, see section 9.

An important difference, compared to HTTP, is that in MUI the URL refers to the connection, not to a document. This is more appropriate in the MUI case, because the number of documents and identities of individual documents may be unknown to the client.

[1] This requirement initially comes from UI connections, where commands and UI updates are sent asynchronously from both parties (see section 8).

On top of MUITP, a protocol is needed for the flow of documents over the connection—which party should start to send, will there be responses, and so on. The current approach in MUI is to have an extremely simple protocol for general connections, with a little more sophisticated protocols for two special cases: RemoteConnect and user-interface connections. In the simple case, MUI connections are uni-directional; only the non-client sends documents. A uni-directional connection can be combined with a user-interface connection, letting the user control the data flow through the user interface. This is enough in many applications, where a service is the provider of some data, but perhaps the protocol will have to be changed, or split into several subprotocols, as more experience is gained from building and evaluating prototypes.

7 RemoteConnect

An important feature of MUI is the ability to establish a connection between two services, and to close it, from a third device on the network. This is handled by RemoteConnect, a protocol on top of MUITP. The messages of the protocol are listed in the DTD shown in appendix C. For these XML messages, a special MIME type `application/x-mui-remote-connect` has been defined.

The URLs of MUI clients are RemoteConnect URLs. When an application wants to establish a connection, it connects to this URL and sends a `ConnectRequest`. When the requested connection is up, the client sends back an `OKResponse` (or an `ErrorResponse` if something goes wrong). Disconnecting a connection with `DisconnectRequest` works similarly.

8 User Interfaces

The second kind of service with a special protocol—besides the RemoteConnect service—is the *user-interface* service. This service lets user interfaces be migrated to clients, so that users can control services from a device with suitable input/output capabilities. The interfaces are described in an XML format, of which the loudspeaker document in table 1 is an example and whose DTD is shown in appendix D. This format has been given the special MIME type `application/x-mui-ui+xml`. There are XML elements for different widgets, with attributes for specifying a certain command to be sent when the widget is chosen (clicked). The current widget types supported are buttons, labels, and panels. Natural widgets to add next are text input fields, and perhaps check boxes and radio buttons. Adding images would also be nice, but that requires a mechanism for referring to external image resources.

When a client connects to a user-interface service, the XML description will be sent back. It is interpreted by the client, and the user interface is shown with help from a user-interface library, such as MIDP [14]. Commands are sent to the service when the user performs an action in the user interface. The service reacts to these in a domain-specific way: the set of commands is specified entirely be the service. It does not need to be standardized, because the service provides the user-interface description itself.

Web forms is a natural comparison for MUI user interfaces. In contrast to these, we wanted to make the communication two-way. The service can send a UI update at any time, which will result in a change in the interface on the client, as was exemplified in section 2. The commands and the UI updates give both pull and push functionality, and more dynamic interfaces than for Web forms. Using HTTP to accomplish this would require polling, with repeated requests to see if something has changed. In relation to web forms, the set of supported widgets is also relevant. With the extensions discussed above, we will support roughly the same set of widgets as for web forms.

In the previous project [6], we implemented the user interfaces in Java, and moved a small Java application to the client instead of XML data. The application was run on the client, displaying the interface. That gave the full power of Java, and the ability to create very dynamic interfaces, but we also felt that it was a quite heavy process to transfer, install, and start an application for each interface. We think XML will suffice for many interfaces, and it has the advantage of being independent of the client platform—it can be rendered differently depending on screen-size, e.g. Still, we consider adding an applet-like mechanism to the architecture, for situations where the power of Java is needed.

9 Implementation and Framework

An implementation of MUI has been written for the standard edition of Java (J2SE). The network communication is over an IP network, with IP multicast as broadcast mechanism in the discovery protocol. This implementation allows simulated devices, entirely in software, that can be used for testing the architecture.

As stated before, the intention is to use MUI in wireless networks. There are several options for the wireless communication, but Bluetooth [2] has been a target from the beginning. The standard for using Bluetooth from Java, JSR-82 [10], was constructed mainly for the small-device edition of Java (J2ME, [12]). In the future, J2ME is indeed a natural choice for MUI, as many devices in the networks will have limited memory and processing power. Consequently, the design of the network classes has been made with J2ME in mind.

One goal of the implementation is to provide a framework for construction of MUI services. The classes and interfaces presented in section 4 are included, but there are also abstract classes that implement more of the behaviour expected for typical services and clients. Creating a service should be simple, if it does not have very specific needs.

We have tried to build the framework with loose coupling between components, using events and listeners for their communication. The idea is that when implementing a new device, it should be possible to pick just the components needed. Functionality that has been implemented includes the following:

- There is support for discovery. These classes rely on IP multicast, but it should be possible to change their implementation to use Bluetooth discovery, without changing their interface too much.
- RemoteConnect is implemented in the abstract client class, so by default all clients speak RemoteConnect. A class `RemoteConnectClient` represents the client part of the RemoteConnect protocol, and can be used by applications for establishing remote connections.
- Migratable user interfaces have support in the form of a number of UI component classes, which make up the interfaces, a `MigratableUI` service, which provides interfaces to clients, and a `UIClient`, which can receive the interfaces and manage the client side of the communication. The UI components can be rendered as XML or as Swing components. On smaller devices, MIDP [14] would be a more suitable user-interface library than Swing.
- The network classes have been designed to fit into the Generic Connection Framework, the more light-weight network API that is used in J2ME [11]. A protocol handler has been written for MUITP. When establishing a connection to a URL, the scheme part of the URL will be used by Java library classes to select the right protocol handler (both for J2SE and J2ME). Adjusting the network connections to Bluetooth will mainly mean to rewrite the protocol handler. The contents of a MUI URL will also have to change, from IP address and port to a Bluetooth 48-bit device address and a server channel identifier[2], something like

  ```
  muitp://0050C000321B:5
  ```

- There is support for parsing and generating XML messages (used for discovery, RemoteConnect and user interfaces). The DOM API in the

[2]Perhaps, it would be the best to make both IP and Bluetooth possible. Then, the URL would have to be extended with some protocol specifier after `muitp`.

Java standard classes is used. In J2ME, there is no built-in support for XML, so we will have to use a third-party parser, or write our own. See [9] for a discussion about parsers and performance considerations with XML in J2ME.

The MUI implementation is provided as a single JAR file, which can be run on any platform with J2SE and an IP network. It includes some sample simulated devices and services: there is a handheld device with a browser application for discovering and connecting services, a slide-show service, which can be connected to a screen, and a poetry service, which produces text that can be shown on a poetry client device. The implementation of the sample services has helped form the framework.

10 Conclusions

This paper has presented the current state of the MUI architecture and implementation, and the ideas behind it. We believe that the basic architecture is simple and flexible enough to be useful in the context of a wireless network where many devices provide services that can be used by clients over user-initiated connections.

Even if the implementation is in Java, the architecture is not Java-specific. The protocols and XML formats from sections 4 to 8 could be implemented by devices with runtime environments for programs written in other languages, such as Smalltalk or C. This could be an advantage where the memory and processing-power resources are scarce.

The use of XML as data format seems reasonable. The documents following our DTDs are quite compact, and we get XML's advantages of a human-readable format with many available parser implementations. There are XML parsers for J2ME that should be small enough (see [9]). XML is well established as a data format in many domains, and standardized as a W3C recommendation [15].

Thanks to the framework, the sample services were quite simple to write. It seems to be a good idea to provide the basis for a standard service, that can be used when developing custom services.

11 Future Work

The current work on MUI involves fixing smaller things in the implementation, and extending the testing of the classes, using the JUnit framework [8]. We will add a few more widget types, as discussed in section 8. Then, we plan to add support in the architecture for user-controlled composition of *virtual services*: several connected services combined into one, for

smoother repeated use. It will be interesting to see if the user interface for the virtual service can be generated from user interfaces of individual services.

We will investigate further the options for migrating to real wireless hardware, such as Bluetooth. We also want to make it possible for services to provide applet-style executable code to be run on clients, as a complement to the XML UI descriptions. Prototypes for more different kinds of services will be built and investigated. Finally, we will work more on the error handling, especially that concerning the partial failure issues discussed in section 5.

References

[1] Ken Arnold et al. *The Jini Specification*. Addisson-Wesley, 1999.

[2] Bluetooth.org. Bluetooth.org - The Official Bluetooth Membership Site. `http://www.bluetooth.org/`.

[3] W. Keith Edwards. *Core Jini*. Prentice Hall, 1999.

[4] W. Keith Edwards, Mark W. Newman, Jana Sedivy, and Trevor Smith. Challenge: Recombinant Computing and the Speakeasy Approach. In *Proceedings of ACM MOBICOM'02*, September 2002.

[5] W. Keith Edwards, Mark W. Newman, and Jana Z. Sedivy. The Case for Recombinant Computing. Technical report, Xerox Palo Alto Research Center, April 2001.

[6] Torbjörn Eklund and David Svensson. Mui: Controlling Equipment via Migrating User Interfaces. Master's thesis, Lund University, January 2003.

[7] Jini.org. The ServiceUI API specification. `http://www.jini.org/standards/ServiceUI/ServiceUISpec.html`.

[8] JUnit.org. JUnit, Testing Resources for Extreme Programming. `http://www.junit.org/`.

[9] Jonathan Knudsen. Parsing XML in J2ME. Technical report, Sun Developer Network, March 2002. `http://developers.sun.com/techtopics/mobility/midp/articles/parsingxml/`.

[10] Motorola. *Java API for Bluetooth Wireless Technology (JSR-82), Specification version 1.0a*, April 2002.

[11] C. Enrique Ortiz. The Generic Connection Framework. Technical report, Sun Developer Network, August 2003. http://developers.sun.com/techtopics/mobility/midp/articles/genericframework/.

[12] Sun. Java 2 Platform, Micro Edition (J2ME). http://java.sun.com/j2me/.

[13] Sun. Java Remote Method Invocation (Java RMI). http://java.sun.com/products/jdk/rmi/.

[14] Sun. Mobile Information Device Profile. http://java.sun.com/products/midp/.

[15] W3C. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, http://www.w3.org/TR/2004/REC-xml-20040204/.

[16] W3C. Web Services. http://www.w3.org/2002/ws/.

[17] W3C. XPointer xpointer() Scheme. http://www.w3.org/TR/xptr-xpointer/, December 2002.

[18] W3C. Web Services Architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, February 2004.

A mui-info.dtd

This DTD defines the format for information about MUI services, clients, and connections:

```
<!ENTITY % serviceinfoelement
         "(ServiceInfo | ClientInfo)">
<!ENTITY % boolean
         "(true | false)">

<!ELEMENT ServiceInfo ((%serviceinfoelement;)*)>
<!ATTLIST ServiceInfo
  name CDATA #REQUIRED
  serviceContentType CDATA #REQUIRED
  url CDATA #REQUIRED>

<!ELEMENT ClientInfo ((%serviceinfoelement;)*)>
<!ATTLIST ClientInfo
  name CDATA #REQUIRED
  clientContentType CDATA #REQUIRED
```

```
  remoteConnectURL CDATA #REQUIRED>

<!ELEMENT ConnectionInfo (ServiceInfo, ClientInfo)>
<!ATTLIST ConnectionInfo
  connectionID CDATA #IMPLIED>

<!ELEMENT ServiceInfoEvent (%serviceinfoelement;)>
<!ATTLIST ServiceInfoEvent
  active %boolean; #REQUIRED>

<!ELEMENT ConnectionInfoEvent (ConnectionInfo)>
<!ATTLIST ConnectionInfoEvent
  active %boolean; #REQUIRED>
```

B mui-discovery.dtd

The DTD for discovery contains a single element, except those included from `mui-info.dtd`:

```
<!-- Include declarations from info DTD -->
<!ENTITY % infodecl SYSTEM "mui-info.dtd">
%infodecl;

<!ELEMENT Inquiry EMPTY>
```

C mui-remote-connect.dtd

The messages of the RemoteConnect protocol are defined in the DTD file `mui-remote-connect.dtd`:

```
<!-- Include declarations from info DTD -->
<!ENTITY % infodecl SYSTEM "mui-info.dtd">
%infodecl;

<!ELEMENT ConnectRequest (ConnectionInfo)>

<!ELEMENT DisconnectRequest (ConnectionInfo)>

<!ELEMENT OKResponse EMPTY>
<!ATTLIST OKResponse
  connectionID CDATA #REQUIRED>
```

```
<!ELEMENT ErrorResponse EMPTY>
<!ATTLIST ErrorResponse
  message CDATA #REQUIRED>
```

D mui-ui.dtd

The UI DTD defines elements for widgets, commands and UI updates:

```
<!ENTITY % uielements  "(Panel | Button | Label)*">

<!-- UI elements -->
<!ELEMENT UI %uielements;>
<!ATTLIST UI
  text CDATA #REQUIRED>

<!ELEMENT Panel %uielements;>
<!ATTLIST Panel
  text CDATA #REQUIRED>

<!ELEMENT Button EMPTY>
<!ATTLIST Button
  text CDATA #REQUIRED
  command CDATA #REQUIRED>

<!ELEMENT Label EMPTY>
<!ATTLIST Label
  text CDATA #REQUIRED>

<!-- Elements for communication with the user -->
<!ELEMENT Command EMPTY>
<!ATTLIST Command
  name CDATA #REQUIRED>

<!ELEMENT UIUpdate EMPTY>
<!ATTLIST UIUpdate
  element CDATA #REQUIRED>
  text CDATA #REQUIRED>
```

Paper II

Composing ad-hoc applications on ad-hoc networks using MUI

David Svensson, Boris Magnusson, and Görel Hedin
Dept. of Computer Science, Lund University, Sweden

{david|boris|gorel}@cs.lth.se

ABSTRACT

The MUI framework supports composition of ad-hoc applications from services available on ad-hoc networked devices. MUI is an open-ended framework, relying on migrating user interfaces and standardized data formats for connecting services, allowing existing devices to be connected to new devices without needing any pre-defined knowledge of their services. We illustrate the benefits of the approach with scenarios involving devices like cameras and laptops that are connected through wireless networks.

1 Introduction

More and more devices in our daily environment are being equipped with wireless communication capabilities, both at home, at work, and out in the street. Using Wi-Fi, Bluetooth, and similar technologies, they can connect and form local ad-hoc networks, not relying on a central network infrastructure. This development brings us closer to the vision of ubiquitous computing [10], where computation blends into the environment, supporting people without requiring constant attention. Services can become available when needed. An example can be when a user, carrying his handheld computer, comes into the vicinity of a particular device, such as his home TV or a ticket vending machine at the train station. Services from these devices can be brought to the handheld computer at that moment. In order to make adequate use of services in this context, special preparation of the handheld must not be needed each time. Instead, services should ideally just emerge on the handheld, ready for immediate use. It should also be possible to combine previously unknown services into new applications. For a more general introduction to the challenges and goals in the field of ubiquitous computing, see for example [3]. In particular we focus on the demand for forming ad-hoc applications, i.e., the possibility to combine devices and services with no, or very general, prior knowledge of each other.

In order to support such ad-hoc applications we have developed the MUI framework (Migrating User Interfaces). MUI allows (1) user interfaces for services to be migrated to other devices, e.g. the handheld in the example above, making it possible to interact with the services remotely, and still in a direct fashion. Services can also (2) be connected to each other via typed data connections. Such connections can be set up remotely, from a third device. For example, using a handheld to connect an MP3 player to a loud speaking system.

For more complex service-to-service interactions, the user interface descriptions can (3) play a dual role of programmatic interfaces, or proxies, for the services. These proxies can be utilized by programs or scripts that glue services together in (4) assemblies.

MUI was originally started as a project with funding from VINNOVA[1], but is now also part of the EU IST project PalCom [6], which, at large, seeks to make ambient computing systems more understandable by humans. This is done by trying to meet a number of challenges, of which perhaps the most important are balancing invisibility with visibility, and finding ways of allowing construction and deconstruction of systems at appropriate levels.

This paper is structured as follows: Section 2 puts the work in context of

[1] VINNOVA - Swedish Agency for Innovation Systems, `http://www.vinnova.se`

previous work in the field. Section 3 presents a scenario that illustrates how MUI can be put to work. Section 4 gives a more in-depth discussion of the framework. Section 5 discusses the overall goals and challenges of PalCom in more detail, and evaluates the MUI framework from this perspective, providing directions for future work. Section 6 concludes the paper.

2 Previous Work

There are several earlier systems proposing solutions to the general problem of how to combine distributed services in a flexible manner. In this section we will discuss some of them and contrast them with the suggested technology in MUI.

Jini [9] is an early attempt to support combination of distributed services. The focus of Jini is programmatic, i.e. it is about programs that communicate. A central mechanism in Jini is a look-up service that aids client programs to find available services. Proxies for services are defined as Java code and in practice also the service provider is a Java program. In contrast, MUI has a user focus, i.e., it is a user that finds and combines services, at least initially. MUI uses a lightweight description of services rather than Java code which enables MUI service providers (and service customers) to be implemented in any language. This is particularly important when small service providers (such as sensors and actuators) are considered. The MUI service descriptions can be used both to directly drive user interfaces, and also as programmatic interfaces. In the latter case, glue code at the service customer will bridge from the customer to the provided service, rather than relying on standardized Java APIs that are defined and must be known prior to connecting to the service.

Speakeasy [3] and MUI share an overall idea of recombinant computing and agree on (1) keeping the user in the loop in deciding when and how components should interact with each other, and (2) using a small set of generic interfaces. Here, Speakeasy uses the terms *serendipitous integration* (the ability to integrate resources in an ad-hoc fashion), and *appropriation* (using resources in unexpected ways). Speakeasy does, however, use mobile Java code to encapsulate communication details, where MUI uses more lightweight descriptions in a textual (XML) format. For data communication, such as audio or video, the Speakeasy solution puts the burden of having a JVM also in dedicated devices such as MP3 players and speakers. The use of downloaded Java code also raises security issues as has been observed when using applets. For UI information, the use of Java to describe these means that customizing the user interface for different output devices is problematic. In contrast, the textual descriptions used in MUI allow the output devices to control the rendering. Furthermore, the MUI solution gives an architectural advantage in that the same interface description can

be used both to drive a UI and to drive a programmatic API.

The focus in the Speakeasy project and MUI are partly different. The focus in Speakeasy has been on providing user interface mechanisms that enable an end user without programming expertise. This is an important aspect of MUI as well, but in addition we have a focus on building ad-hoc composite applications, assemblies, using the control part of a remote device as an API. Assemblies in MUI can offer new services which can be used in other assemblies in their turn, thus providing a hierarchical composition mechanism.

Barton et. al. [2] have chosen to build on existing HTTP technology, enhanced with a "Producer" mechanism to register services with a HTTP-server and XForms to communicate between such services and sensors (which here is used for any source of information). XForms share, with MUI, the approach to use XML-inspired textual descriptions for communication, thus avoiding dependence on Java. Being based on existing HTTP it is, however, limited by the capabilities of that technology such as a communication model based on pull and no direct support for push, as well as other restrictions.

Our early work with MUI has been presented in the master's thesis [4] and in the paper [8]. In the master's thesis project, a prototype with a VCR was built, where a user interface description could be migrated from the VCR to a handheld computer via Bluetooth: the handheld computer became a remote control for the VCR. The paper [8] presented MUI's discovery protocol, and XML-based languages for service and UI descriptions. At that stage, the focus was on migration of user interfaces. Since then we have started to work also with assemblies, and with using the interface descriptions as programmatic APIs.

3 Scenario: Distributed slideshow

As an example of a scenario where MUI can be applied, consider a slight variant of the traditional presentation session scenario, where slide shows are projected onto a large white screen. In the traditional scenario, the slide shows run on a laptop connected to the projector. When it is time for the next speaker, he either switches to his slide show, which has been copied in advance to that laptop, or he plugs in his own laptop. In our variant of this scenario, we make use of MUI to provide more flexibility. Rather than physically connecting a laptop to the projector, we use a computerized projector that the laptops can communicate with via the wireless network. Furthermore, a mobile phone can be used as a remote controller for the slide show on the laptop. This scenario is more flexible in several ways: First, the slide shows can be run on the different speakers' own laptops, giving an obvious advantage in terms of less preparation in advance. Sec-

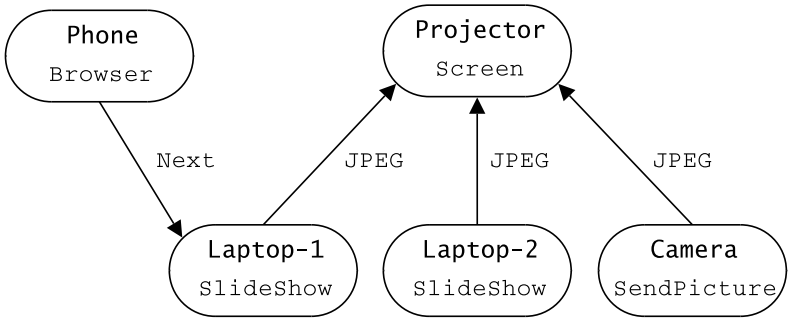


Figure 1: Distributed slideshow scenario

ond, the laptops can be left anywhere in the room, and the speaker can also be located anywhere in the room, not necessarily beside the laptop. Third, more than one slideshow can be shown at the same time, with images interleaved. This can be useful in group discussions, where one person might want to jump in with a few slides in the middle of a presentation.

Figure 1 shows a set up for this scenario. The devices in this scenario: projectors, laptops, etc., are *MUI-fied*, i.e., they run the MUI system. This is easily accomplished for a laptop. The projector, on the other hand, needs to be equipped with an embedded computer with wireless capabilities. Today, this situation is easily emulated by using a standard projector and physically connecting it with a dedicated computer.

The projector has a MUI service, Screen, that can receive JPEG images and project them onto the physical screen. A laptop has a MUI service, SlideShow, which has a user interface for controlling a slide show (with buttons Play, Stop, Next, etc.), and which can send out slide show JPEG images on network connections. The mobile phone has a MUI browser, that can discover nearby devices and their services. Through the browser, the user can ask the Screen to connect itself to the SlideShow of a specific laptop, causing the images sent out from that laptop to appear on the screen. In the browser, the user can also ask for the SlideShow user interface which causes this to migrate from the laptop and pop up on the display of the phone. Then, he/she can use the phone to change slides during the presentation. The laptop also has a MUI browser, so, if desired, the user can issue the user interface commands (Play, Stop, Next, ...) and/or set up the service connections directly from the laptop as well. If several people have their slide shows connected to the projector, the latest slide is shown on the screen whenever one of them changes to a new slide.

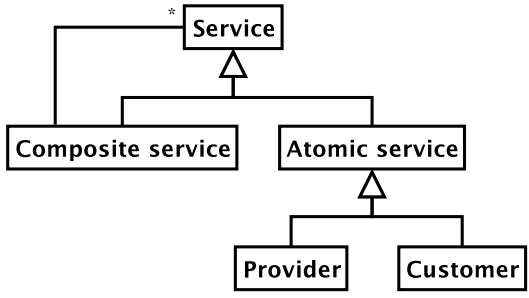


Figure 2: Service hierarchy

3.1 Extending the scenario: adding a camera

The MUI system is open-ended, allowing new devices with new services to easily be added and connected. Suppose the presentation is at a conference for bottle cap collectors, and a person in the audience would like to show a particular rare bottle cap. With a camera with a MUI service Camera that can send JPEG images, she can simply take a picture of the bottle cap, and send it to the projector to show the image.

3.2 Ad-hoc composition

In order to support composition of ad-hoc applications, MUI relies on standardized connection types. This is in contrast to systems that rely on standardized service types, like Jini [9] I.e., in MUI it is possible to connect the laptop to the projector because they send and receive JPEG images. The service Screen does not need any prior knowledge of the service SlideShow, or vice versa. This allows a service to be used in new, perhaps unforeseen, ways. The SlideShow can be connected to any other service that can receive JPEG images as well, e.g., printers, file storage devices, etc.

4 The MUI framework

MUI is based on services. Services are what runs on the devices, and what offer functionality to users and to other services. The services describe themselves in XML service descriptions, which are distributed to other devices on the network by means of a discovery protocol [8]. More complex services can be formed as composite services with subservices (see Figure 2), but it is the basic, atomic, services that are ultimately connected via the ad-hoc network. These have a certain type, and can be either providers or customers. We will describe below the different roles these two play in

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE CustomerInfo SYSTEM "mui-info.dtd">
<CustomerInfo name="Screen"
              customerContentType="image/jpeg"
              urn="mui://10.0.0.3/screen">
  <Subservice urn="mui://10.0.0.3/ui"/>
</CustomerInfo>
```

Figure 3: Service description for the screen, which is a customer for JPEG images. The URN identifies the service. There is one subservice (a user interface), whose service description is referenced by its URN.

connections. Figure 3 shows a small example of an XML service description, for the Screen in the slideshow scenario. The type of an atomic service determines the kind of connections that can be established to it. There are two main kinds of connections: (1) *control connections*, allowing the service to be controlled by another device, either programmatically or via a generated user interface; and (2) *data connections*, for transfer of typed data. The Speakeasy infrastructure [3] is in many ways similar to ours: services have meta-data descriptions, and connections can be either for transmission of data, or for control. In both systems, there is also a browser from which the user can view and set up connections.

4.1 Control connections

The protocol implemented by a control connection is described as a service description of the type *control*. These descriptions can be rendered as a user interface in order to allow the user to inspect the functionality of the service, and to interact with it directly, which is a key aspect of MUI. An example is the user controlling the laptop SlideShow service via the mobile phone: an XML description for a simplified version of this interface is shown in Figure 4.

When a control customer is connected to a control provider, the service description is migrated to the customer, and the user interface can be rendered on the receiving device. The XML description specifies mappings from actions in the user interface to what commands should actually be sent over the network to the service, and the service can also send out messages which lead to updates in the user interface. So, after the user interface has been migrated, the roles of the two sides are really symmetric—we have a peer-to-peer arrangement, where, e.g., both pull and push are possible. It is up to the service programmer, who also writes the service description, to decide upon the details of this protocol. A brief example of this kind of two-way communication will be discussed in Section 4.5.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE ControlStructure SYSTEM "mui-control.dtd">
<ControlStructure text="Slideshow">
  <InCommand id="prev" text="Previous slide"/>
  <InCommand id="next" text="Next slide"/>
</ControlStructure>
```

Figure 4: A control interface describing a simplified slideshow service. There are two commands for moving between slides.

Representing a user interface as a description has the advantage that the different browsers on different devices, having different display capabilities, can use different ways to present the user interface. This is an area that has attracted some attention in itself, see for example [7], and there are a number of XML-based user-interface markup languages, e.g. UIML [1].

An alternative use of the service descriptions is as proxies that can be used for controlling the service programmatically from another device. This allows composite services to be built, relying on distributed subservices, where a script on one device can coordinate the subservices. In PalCom, this script is referred to as an assembly [5]. We will briefly discuss an example of this in Section 4.5.

4.2 Data connections

Besides control connections, there are data connections for transfer of typed data. These are formed when a customer of a given type is connected to a provider of the same type. For example, the Screen service is a customer for type JPEG, and can be connected to providers of type JPEG, e.g., those in SlideShow and SendPicture. Currently, we use MIME types to distinguish different types.

The data connections can also be called streaming connections, because data flows from provider to customer, one message at a time. This suits multimedia formats, such as streaming audio and video, but can also be used in less resource-demanding applications, such as when one JPEG image is sent every time a speaker switches to the next slide in the slideshow example.

4.3 Remote connection of services

An important aspect of MUI is the possibility to connect two services from a third device. This was exemplified above, where the user connected the laptop's SlideShow to the projector's Screen, using the browser on the mo-

bile phone. In order to allow this, there is a simple protocol which devices can use for instructing a device to connect one of its services to a service on another device: the mobile phone instructed the projector to connect its Screen to the laptop's SlideShow. Similarly, it is possible to disconnect two services that are currently connected. This functionality builds on a property of the discovery protocol: devices announce information not only about the devices themselves, and about their services, but also about established connections. This support for connections gives more visibility for the user. He can see not only what devices and services there are, but he can also view and control the connections.

4.4 The MUI browser

A MUI browser has been implemented, on which nearby MUI devices and services can be inspected and controlled, and from which new connections can be established. Using the browser for remote control allows devices to be networked that do not themselves have any or very limited user interaction capabilities, e.g., sensors and actuators. Browsers can be expected to run on more resource-rich devices, such as PDAs and mobile phones. The current implementation is in Java, but having an underlying JVM is not essential—the browser could be written in any language.

Existing connections can be viewed, and possibly disconnected, as mentioned above. The browser utilizes the hierarchical structure of services for making the connection process easier and more natural. E.g., the user can choose to connect the SlideShow service directly to the Screen, without opening up to see what subservices they have. In this case, there will be exactly one matching provider-customer pair, and this pair—the JPEG provider of SlideShow and the JPEG customer of Screen—will be connected. If there had been more than one matching pair, the user would have been asked to select the one he intended. This can be seen as a simple way of supporting visibility at an appropriate level.

4.5 Example of usage: the SitePack

The SitePack is one of the scenarios studied in the PalCom project (see [6, 5] for more background information). In this scenario, landscape architects out in the field make use of PalCom technology for combining devices in different set-ups, suitable for the situation at hand. One example is during the documenting phase, when photos taken at a site need to be tagged with location and other information, so they can be put together later at the office. For this purpose, the landscape architects use three devices from the SitePack: a digital camera, a GPS, and a handheld computer. When a picture is taken, the current GPS location should automatically be saved with the picture. This is realized as an ad-hoc application, with an assembly

running on the handheld computer, that coordinates the camera and the GPS. The special logic needed for this particular case is in the assembly. It is important to note that the camera and the GPS are not prepared in advance for this scenario, except being PalCom-compliant at a general level.

We have implemented a simple version of this scenario using the MUI framework. The camera and the GPS expose their functionality as MUI services, both as data (GPS coordinates) and as user-interface descriptions. The user-interface descriptions are used as programmatic control interfaces by the assembly script, running at the handheld (the assembly is currently "hard-coded" in Java, but is to be written in a simpler script language later). The control interfaces are migrated to the handheld computer when the assembly is activated. As a picture is taken by the camera, the assembly gets notified through a message over the camera's control interface. In response to this, it asks for the latest picture from the camera, using an operation in the control interface. When the assembly gets the picture, the picture is tagged with the latest coordinate received from the GPS (using a special coordinate stuffer service, running on the handheld), and is sent to a back-end server for storage.

Important aspects of the implementation are that it makes use of the two-way communication that is possible with control interfaces, where both the camera service and the assembly initiate communication at different stages, and that it is an example of a user-interface description functioning as a programmatic proxy. As mentioned above, it should also be noted that only the handheld has been especially prepared for this scenario: the GPS and the camera expose their normal interfaces. The preparation of the handheld consists of construction or installation of the special coordinate stuffer service, and of writing the assembly script. The coordinate stuffer, which manipulates JPEG image meta-data, is an example of a service which is best implemented in a full-blown programming language, such as Java, and which therefore has to be written by someone with that knowledge. It offers a service description as other services. The script, on the other hand, should be possible to write by end users. This is where the actual adaptation to the scenario is done.

5 Evaluation and Future work

MUI involves the user in the establishment of connections between services. This gives him visibility and control over how services form ad-hoc applications. But, at the same time, this must not become a burden for him. It has to be possible also to automate the process. E.g., when he comes home, carrying his MP3 player, he might want it to automatically connect to his set of loudspeakers. Therefore, we are working on support for saving a set of connections in assemblies, which can be stored, e.g. on the MP3

player, and which can actively establish their connections. This is a simpler form of assembly than the SitePack assembly above. In our continued work, we will combine these types of assemblies into one type, so that a simple set of connections can be further customized with script logic.

From a PalCom perspective, it is interesting to look at how well MUI supports the so called *palpable qualities*, i.e. how well it meets the PalCom challenges mentioned above [6]. Our focus has been mainly on visibility/invisibility, and on construction/deconstruction. Scalability, complemented with understandability, is another important PalCom challenge. We will relate to these three in turn, and after that there will be a short discussion of security aspects, which are of course also important in the MUI context.

5.1 Visibility and invisibility

Ubiquitous computing brings a degree of invisibility to computing systems, in that they blend into the environment. PalCom highlights the need for balancing this with an appropriate degree of visibility, so that the systems remain understandable. Regarding visibility, we find it important that the user can be involved in the process of setting up connections between services. It is of course often desirable with an automatic process for this, but when the user is involved it is easier for her to understand the system. In many cases, it will also be necessary, because a program cannot be expected to understand the interfaces of the previously unknown services that will pop up in these ad-hoc networks. Another point when it comes to visibility is the merit of letting the discovery protocol distribute information about established connections, and not only about the services. This can give the user a view of the current communication.

5.2 Construction and deconstruction

For construction/deconstruction, the notion of assembly is important in PalCom. MUI combines this with limited pre-defined knowledge of service interfaces. When a new service is encountered, it should be as easy as possible for the end user to make use of it in assemblies. We think this should be approached at different levels: At one level, it should be possible to save a set of connections as an assembly for later activation. At another level, programmable scripts should support the need for more complex logic. In both cases, the deconstruction aspect is crucial—it must be possible to open up an assembly and inspect its parts, especially when something goes wrong.

The current implementation of MUI contains first versions of support for both levels of assemblies: it is possible to establish a number of connections and save them in a list for later re-activation, and the SitePack implemen-

tation, described in Section 4.5, demonstrates a more complex assembly. Future work will involve refinement of both types, e.g. with the introduction of a script language for the more complex assemblies, and unification of the two into one concept, so they can be handled similarly.

5.3 Scalability and understandability

A third PalCom challenge that is certainly relevant for MUI is the need for supporting scalability, complemented with understandability. When using the MUI browser, the user must not be overwhelmed with the sheer amount of available services. One step in the right direction here, which we have implemented and which is also related to the visibility/invisibility challenge, is the possibility to group services as composite services. Another useful mechanism in our implementation is the use of type information for narrowing down the number of possible end-points during the establishment of a connection.

Scoping mechanisms on the network will also be needed, for making sure that the services discovered are really reachable in the current context. Similar concerns must be handled for the discovery of established connections. Ideally, only relevant connections should be shown, and for connections there is also an additional problem area of visualization.

5.4 Security

In relation to scoping, there is the general question of security. It is important that unauthorized users or devices are not able to use your services, or spy on your connections, or modify them. To some degree, we rely on mechanisms in the lower networking layers here. In Bluetooth, e.g., two devices have to be paired before they can use each other's services. Pairing occurs once, and does not need to be done more in the future. But, there are several open issues. E.g., it has to be possible to use public services, out in the street, without having to pair each time, and still without your connections being visible to everyone. There should be some more advanced scoping mechanism also for this.

In many cases, social conventions provide sufficient security. In the slideshow scenario, all participants that have the pin codes for pairing with the devices, are also trusted with not using the technical possibilities for disturbing a presentation. Social structures could also be used for scoping. One example is the Speakeasy converspaces [3], where members of a converspace can invite others to share a set of components. This way, it will be possible to trust users, on the basis of trust in those who invited them. In order to use such social conventions and trust, a complement can be logging of events that can be used to find out who did what after the fact.

6 Conclusions

MUI answers some of the basic challenges in ubiquitous computing. It enables ad-hoc interaction among devices without prior knowledge of each other. They need to share a common, generic set of protocols for discovery and communicating service descriptions, but nothing that is special for a particular service. A user can very intuitively connect service descriptions to a browser and remotely control devices. It is also intuitive in a browser to connect the typed data channels between different devices and thus have them share information like audio or JPEG pictures. Here the data types are standardized, not the services. With this basic functionality, MUI supports many of the scenarios envisioned as ubiquitous computing.

Sets of connections can be stored as assemblies, saving the user from establishing the connection over again in case it is a situation that will occur frequently. In more complex situations, an assembly can be instrumented with a script that ties together service descriptions from remote devices (now interpreted as APIs rather than UIs). In this way complex interaction between devices can be constructed in an hierarchical fashion, thus supporting composition and decomposition.

In case an application needs algorithmic support that goes beyond what a scripting language can offer, the MUI model enables program components to be incorporated in an assembly, if only they implement the discovery protocol and offer a service description of their capabilities.

The requirement for a device to take part in a MUI system is to observe the generic set of protocols. MUI is thus an open framework that can be implemented in any language. It might be particularly interesting to implement "small" devices such as sensors or actuators in a low-level language and in such cases the effort to implement the MUI protocols should be small.

The MUI model is supporting a user-centric perspective where the user decides on when and how devices and services should interact with each other, but at the same time offers a programmatic perspective for automating tasks.

References

[1] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. UIML: An Appliance-Independent XML User Interface Language. *WWW8 / Computer Networks*, 31(11-16):1695–1708, 1999.

[2] John J. Barton, Tim Kindberg, Hui Dai, Nissanka B. Priyantha, and Fahd Al-Bin-Ali. Sensor-enhanced Mobile Web Clients: an XForms Approach. In *Proceedings of ACM-WWW 2003*, pages 80–89, Budapest, Hungary, May 2003.

[3] W. Keith Edwards, Mark W. Newman, Jana Sedivy, and Trevor Smith. Challenge: Recombinant Computing and the Speakeasy Approach. In *Proceedings of ACM MOBICOM'02*, September 2002.

[4] Torbjörn Eklund and David Svensson. Mui: Controlling Equipment via Migrating User Interfaces. Master's thesis, Lund University, January 2003.

[5] Mads Ingstrup and Klaus Marius Hansen. Palpable assemblies: Dynamic service composition for ubiquitous computing. In *Proceedings of SEKE 2005, The Seventeenth International Conference on Software Engineering and Knowledge Engineering*, 2005.

[6] PalCom website. Palpable computing—a new perspective on ambient computing. `http://www.ist-palcom.org/`. IST-002057.

[7] Peter Rigole, Chris Vandervelpen, Kris Luyten, Yves Vandewoude, Karin Coninx, and Yolande Berbers. A Component-Based Infrastructure for Pervasive User Interaction. In *International Workshop on Software Techniques for Embedded and Pervasive Systems STEPS'2005*, Munich, Germany, May 2005.

[8] David Svensson and Boris Magnusson. An Architecture for Migrating User Interfaces. In Koskimies, Lilius, Porres, and Østerbye, editors, *NWPER'2004, 11th Nordic Workshop on Programming and Software Development Tools and Techniques*, pages 31–44, Turku, Finland, August 2004.

[9] Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, pages 76–82, July 1999.

[10] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, February 1991.

Paper III

Discovery and communication protocols for palpable computing

David Svensson, Boris Magnusson, and Görel Hedin
Dept. of Computer Science, Lund University, Sweden

{david|boris|gorel}@cs.lth.se

ABSTRACT

This paper reports on the specification and implementation of discovery and communication protocols for *palpable computing*, which is a flavour of pervasive computing, explored in the PalCom project. The protocols have been designed to support the construction of *ad-hoc applications*, i.e. applications put together by end users for utilizing available services in a particular situation.

Based on the work on prototype scenarios in the PalCom project, we have drawn a number of requirements on the protocols. These include *device awareness*, the need for making physical devices explicit in the discovery protocol, *on-demand service discovery*, which means that service descriptions should be transferred only on demand and handled incrementally, and the need for *discovering connections*, in addition to discovering devices and services. Bandwidth and responsiveness considerations have lead us to propose a new device discovery scheme, called *announcement heartbeats*.

1 Introduction

PalCom is an EC-funded integrated project within the area of pervasive computing, which introduces the notion of *palpable computing* [16]. Palpable computing systems are capable of being noticed and mentally apprehended, to a larger degree than pervasive systems in general. As one of the keys to better human understanding of pervasive systems, PalCom seeks to enable both construction and de-construction of systems by end users. In our work within PalCom, we have focused on this challenge, and in particular on finding ways of supporting so called *ad-hoc applications*, which are applications that are put together in a certain situation, from services available in that particular context [22].

This paper reports on the discovery and communication protocols we have developed to support such ad-hoc applications. The requirement for supporting ad-hoc networks has heavily influenced the design of the protocols. We have implemented the protocols in Java and they run on a normal JVM as well as on the Pal-VM [15], a new virtual machine for embedded devices that is being developed in the PalCom project. The protocols are independent of technology and have at the time of writing been implemented for IP networks on top of UDP, and an implementation for short-range wireless networks using Bluetooth [3] is forthcoming.

Asynchronous communication

The unreliable nature of wireless ad-hoc networks calls for a model based on asynchronous communication. A synchronous model, such as one based on Remote Method Invocation (RMI, [20]), is not suitable. In RMI, the caller of a method is blocked until the method has completed, and a return value has been received. With dropouts in the communication, this means that the caller may, theoretically, get blocked forever. Instead, we use asynchronous message passing both in the discovery protocol, and for communication between services.

Discovery protocols

There are many existing protocols for discovery of devices and services in pervasive computing systems. The basic purpose of such protocols is that it should not be necessary to know the exact address of a service (such as IP address and port). Instead, it should be possible to discover and use the service when one connects to a network. Some discovery protocols are developed in academia, such as DEAPspace and GSD [14, 4], some by software vendors as part of operating systems, such as Jini, UPnP, and Apple's Bonjour [25, 23, 1], and some by various consortia, such as Salutation [18],

IETF and Bluetooth SIG. Zhu et al. have compared a number of these existing discovery protocols along different dimensions, and placed them in a taxonomy [26]. This taxonomy, and how our protocols relate, will be discussed in Section 10.

Paper organization

The rest of this paper is organized as follows. Section 2 discusses our notion of ad-hoc applications, and Section 3 the requirements on the protocols that we have derived from this kind of applications. In this section we motivate why we have chosen to develop new protocols and message formats. A more detailed description of our implementation platform is presented in Section 4, with presentations of the protocols in Sections 5 through 8. In Section 9, we evaluate the protocols in relation to our requirements. Section 10 discusses related work, and Section 11 concludes the paper.

2 Ad-hoc applications

The general setting for a palpable application is that it involves a number of devices, each offering one or more services on the network that the user wants to combine ad-hoc, i.e. in new, unforeseen ways. We want to make this possible also if the services are not designed to work together, i.e they do not share a common domain-level protocol.

A service has either a *provider* role or a *customer* role, and connections can only be established between two services with different roles. Being a provider means that the service offers some functionality, described in a service description, that can be used by connected customers. There are two basic kinds of provided services: *data services* and *control services*. A data service provides streamed data. For example, a photo service on a digital camera device can provide digital images to the customers. A control service provides commands for remote control of the device. For example, a control service on the digital camera can provide a command for taking a photo.

In creating an ad-hoc application, the user typically interacts with the services via a PalCom *browser application*. The browser allows the user to establish connections between services, e.g., connect the camera's photo service to a customer service on a database backend. The browser also allows the user to directly interact with a control service, rendering the commands as a graphical user interface. For example, the command for taking a photo can be rendered as a button in the browser. The user interface displayer in the browser thus serves as a customer of control services.

Ad-hoc applications can be made more powerful through the introduction

of an *assembly*, an entity that keeps a configuration of services, and coordinates them [21]. The interaction with a service can be automated by an assembly script which then serves as the customer of control services. For example, to take a photo every 10 seconds, an assembly can coordinate incoming events from a timer service with events sent to the control service of the camera. Through the use of the PalCom browser and assemblies, the user can construct and deconstruct ad-hoc applications: put them together and try them out interactively, save them and run them as assemblies, and later deconstruct and change them as needed.

The protocols discussed in this paper have been designed with the overall goals of ad-hoc applications in mind. We have developed protocols for discovery, and for communication between services. The discovery protocol is used by a service, or by a browser application, for obtaining information about devices and services in the vicinity. Then, the communication protocol can be used for interacting with the services, either manually, or automatically through an assembly.

3 Requirements

Within the PalCom project, a number of prototypes have been constructed, exploring scenarios in collaboration with end users. Examples include IT support for professional landscape architects, rehabilitation games for disabled children, major emergency incidents scenarios, and several others. We have drawn on this work to provide requirements for the developed protocols.

The requirements we put on discovery and communication protocols are both functional and non-functional. Our primary goal is to support ad-hoc applications, as mentioned above. We need to do that in a way which keeps the user's experienced performance high, and which does not demand too much bandwidth from the network. This section will discuss our most important considerations:

Standardization at a general level In order to enable ad-hoc combinations, the protocols must not rely on domain-specific standards.

Support for transient devices It is normal in pervasive systems that devices join and leave networks frequently. When a device leaves the network range, or goes down unexpectedly, this must become known to the other devices within a reasonable time.

No central directory Our discovery protocol should not be based on a central directory, where services are looked up, because in the ad-hoc networks we target, such a central node cannot always be expected to be available.

Device awareness As the identity of the physical devices are important for the end users, it is a requirement that these are announced in the discovery protocol.

On-demand service discovery Descriptions of services are anticipated to often become rather extensive, so the whole descriptions should not be announced with every device announcement. Instead, they should be available on-demand for interested customers.

Discovery of connections In order for the user to be able to get a view of the communication between nearby services, and manipulate it, it must be possible to discover connections between services.

Limited broadcasting Messages that are broadcasted to all devices in the network must not be used too often in the discovery protocol, because they require more bandwidth.

Advertisements, not queries It must not be necessary to specify the properties of a service in order to discover it.

The first two of these requirements are for communication in general, while the other six are specific to the discovery protocol. The requirements will be discussed in more detail in the following.

3.1 Standardization at a general level

PalCom devices should be possible to combine in ways that are not preplanned, in ad-hoc ways, which means that the discovery protocol must be understood and supported by all PalCom devices. This is standardization at a general level, not on the domain level, as discussed in [22]. Furthermore, PalCom should be an *open framework*. This means it must be possible to implement PalCom devices not only on top of the existing implemented framework (see section 4), but also using other technologies. This could be for supporting "small devices", such as sensors or actuators, or for adding support for PalCom to existing devices. This communication on the discovery level must thus be defined and standardized as a protocol with regards to message formats and representation. In particular, we do not see it as feasible to base the protocol on mobile code, as in Jini [25]. That puts too strict demands on the run-time systems of devices.

For small devices, it is important that the overhead inflicted by the discovery protocol is small. Such a device would be one that simply offers a service. It will never initiate a discovery procedure, because it is itself not interested in replies. It can answer requests with a simple, often fixed answer, and it can ignore the broadcasted replies from the other devices.

Regarding standardization of protocols, there are different levels of PalCom communication:

- *The discovery protocol* enables devices to find other devices, services and connections. This is a standard protocol, supported by all devices.
- *Service descriptions*, obtained through discovery, describe the *service protocol* observed by a service. A service protocol defines the messages and format used by a particular service. There is a standard format for messages that can be used, but for services with special demands on performance, encryption, etc, a non-standard format can also be used, for utilizing available technology in an optimal way. This will be discussed more in Section 7. The content of the messages is service-specific, and not standardized—it is the provider of the service who decides what messages it can handle. The descriptions themselves follow a standard format, but will probably have to be extended in the future to support non-standard service protocols.

3.2 Support for transient devices

It is in the very nature of pervasive computing systems, that many devices will frequently join and leave networks. Devices will follow people, forming ad-hoc networks with the devices currently in the vicinity. These networks are generally wireless, with varying signal strengths. This means that the protocols must be prepared to handle dropouts in the communication, and that replies to messages cannot always be expected. In general, this calls for a model based on asynchronous communication. For the discovery protocol, it is important that new devices are discovered fairly quickly, and that devices leaving a network do not remain "discovered" for too long, in the caches of other devices.

A related aspect to consider when designing the discovery protocol is how we get to know that a device is not available any more. This is a tough problem, because when you are dead, or have left the network range, it is too late to tell the world, and death can come unexpectedly. The problem is called *partial failures*, one of the issues that makes distributed systems more difficult to handle [10]. Some kind of *heartbeat* is one way of solving this, and we propose a solution to this fundamental problem with a combination where the heartbeat also works for discovering devices. We call this scheme *announcement heartbeats*. It is the most "eager" device that controls the frequency of the heartbeat. The technique seems to be minimal in some aspects. As mentioned above, it is also desirable to minimize the messages broadcasted in a network; for network load reasons, both in terms of the number of such messages and the size of them. In our protocol, broadcasting is only needed for these combined heartbeat and device-discovery messages. Compared with techniques within fault tolerance, the *announcement heartbeats* scheme can be seen as a combination of the "I Am Alive" and "Are You Alive" patterns in [19]. The heartbeats sent out by the most

eager device are I-am-alive messages, spreading information about that device. At the same time, they work as Are-you-alive messages, triggering responses from the other devices.

An alternative to using a heartbeat solution is sometimes *leasing*. In general, leasing means that a service keeps a resource for another, as long as the lease is renewed, or until it times out. In Jini, leasing is used for service announcement. Services list themselves at lookup services, by leasing a place in their directories, and clients go to the lookup services for finding available services. A problem with this scheme is that when a service disappears abruptly, the lookup service will not remove the entry for the service until its lease times out. In the mean-time, clients will be informed as if the service was still available, and it is *the service* that decides the duration of the lease. This does not fit well with the PalCom scenarios, where very up-to-date information is required in many cases. As an example, consider a doctor that uses a BioMonitor for monitoring a patient at the site of an accident. The BioMonitor service may have granted a rather long lease, perhaps based on its battery power, but it may still be the case that it does not respond (maybe it has been moved out of range). Here, it must instead be the doctor that sets the time limit. In a way, our protocol can be seen as a leasing protocol where the user of a service sets the time limit.

3.3 No central directory

One important design parameter for a discovery protocol is whether there is a central directory or not. The discovery protocols for Jini and Salutation make use of this: services register themselves in a central directory, and clients go to the directory for looking up services. The opposite alternative is to let each device or service announce its presence on the network directly by means of broadcasts,[1] and for clients to go directly to the service for interacting with it. This approach is taken, e.g., by UPnP [23]. There are also hybrid approaches: one example is VSD [11], where more powerful devices can volunteer to serve as directories, when it is needed.

Based on our work with end-user composition in PalCom, and on requirements drawn from the PalCom prototypes, we have chosen an approach without a central directory. In many cases there are only a few devices forming an ad-hoc network, and, as pointed out in [14], it is often not obvious what device should host the directory in such situations. As we see it, directories are better used for limiting bandwidth usage in large networks, as discussed in Section 9.1.

[1] Here, we use the term *broadcast* in a broad sense, referring to a message that goes out to a number of anonymous receivers. On an IP network, broadcast may be implemented using IP multicast.

3.4 Device awareness

The most obvious entity that should be possible to discover through a discovery protocol is the service. Services are what is ultimately used over the network. Starting from this, several protocol designers have chosen to let services be the only entity that can be discovered. One example is Jini, whose creators speak about *device agnosticism*. This means that also devices are treated as services by Jini, and this is presented as one of the key contributions of the technology. Hardware and software can be handled in a unified way by clients [6].

For pervasive computing, we argue that this unification is not useful. In this context, the integration of discovery protocols with people is a very serious challenge [26]: pervasive services are localized (as opposed to, e.g., Web services), and people can be expected to care more about exactly which of several services is used. For people, the physical devices *are* important. Therefore, we let devices be discovered at the top level, and services below that, for each device. This is backed by the observation that in many application prototypes studied in PalCom, the actual device that hosts a service plays an important role for the end user: the GeoTagger and the SiteSticks for landscape architects, biomonitor sensors for emergency personnel [12], enhanced incubators at the hospital, the Stone device supporting pregnant women [24], etc. In these scenarios there are multiple devices with identical services around, and device identity matters for the user—in particular when something goes wrong. Even in the often-mentioned example with finding a "Print" service for printing a document,[2] the physical location of the printer (the device that provides the service) *is* relevant. This is the motivation for having devices as discoverable elements. We should also mention that this decision is not unique. UPnP focuses on devices, and lets services be discovered inside device descriptions [23].

3.5 On-demand service discovery

Of the messages transferred by the discovery protocol, it is the service descriptions that we expect to become most extensive. For complex services, the descriptions will contain specifications for many messages. Therefore, these descriptions are not sent out with every device announcement. Instead, they are obtained through unicast requests from those devices that are interested. This may be the case, e.g., when a user has opened up a service list for one device in a browser.

[2] The "Print" example is a common example in discovery protocol literature.

3.6 Discovery of connections

Besides devices and services, our protocol also supports discovery of connections. Like has been discussed in [22], the visibility experienced by the user of a browser is increased by visualizing also connections, and the possibility to manipulate established connections depends on having connections discoverable. This functionality is also a basis for our work on assemblies.

3.7 Limited broadcasting

Sending out a broadcast message requires more bandwidth from the network than a unicast message, in general. Therefore, it is desirable to keep the number of broadcasts down, and this is a design goal for our discovery protocol. We should mention that the scope of the protocol is for local networks. Large and/or heterogeneous networks are discussed under Section 9.1.

Having a discovery protocol without a central directory, as discussed in Section 3.3, does not mean that all information has to be provided using broadcasts. Instead, many protocols use broadcasts in an initial phase, and then unicast communication for obtaining detailed information about services. As an example, Jini uses a broadcast protocol for announcing its lookup servers (directories), and then clients contact the servers using unicast. One benefit with such an approach is that a unicast call demands less bandwidth from the network, as mentioned above. We have opted for a similar solution: Broadcasts are used for sending out overview information about devices. After that, a pull protocol with unicast messages is used for distributing information about services and connections. These, potentially extensive, pieces of information go only to those devices that are explicitly interested.

The cost of a broadcast depends on the type of the network. On larger IP networks, there are normally multicast routers, that permit communication using *multicast* instead of pure broadcast. This is more efficient than pure broadcasting to all nodes on a subnet, because the message only reaches those nodes that have registered at a particular multicast address. In our PalCom implementation with UDP, we make use of multicast communication.

3.8 Advertisements, not queries

In much of the literature about device and service discovery (see, e.g., [17, 26, 9]), a query-based approach is taken as the starting point. This means that, initially, a user or client makes a query on the network, specifying

the name or properties of the service, or services, that he wants to use. Again, the "Print" service is the most common example. The query-based approach has advantages. It is possible to let the (relatively compact) query be broadcasted out on the network, and for only those services that match the query to reply with their (perhaps extensive) service descriptions, in direct unicasts to the client. This keeps the load on the network down.

Still, we do not want to base our protocol on queries. For one thing, the language of the queries is a complication. In order to be able to formulate good queries, you would need some kind of ontology for the properties of all types of services. This leads to the type of domain-level standardization that we want to avoid. Secondly, relying on queries means that the connection to the physical devices in your vicinity, as advocated in Section 3.4, is easily lost, because the focus is on services, not devices. Instead, we prepare for building querying mechanisms on top of our announcement-based approach, where services can be filtered depending on the contents of their service descriptions. Building on top of the discovery protocol, rather than basing the discovery protocol on queries, also means that the query calculations can be made other places than on the (possibly small) device that provides the service, which is good, and how the query is implemented can be realized in many different ways without changing the protocol.

4 Implementation

Before going into the details of the protocols, we would like to briefly outline our implementation, and the platforms on which it runs. Figure 1 shows selected parts of the PalCom protocol stack, with the communication layer zoomed in. The communication layer is where communication and discovery protocols are implemented. As mentioned above, Pal-VM [15] is the virtual machine (VM) developed within PalCom. This is a VM which supports programs written in Smalltalk, Java, and BETA. The communication components run on top of the Pal-VM, and they also run on an ordinary Java Virtual Machine (JVM), shown beside the Pal-VM in Figure 1. The layer just above the VMs, with `pal-base` and `pal-jbase`, contains base libraries in versions specific for each VM. In the layers above that, components written in Java run on both VMs, with the same source code. This is the case for the components in `process` and `communication`. The main reason for targeting two different VMs is that there exist more libraries for the JVM, supporting graphics etc., that can be used in browser applications and in prototypes, while the Pal-VM is being developed further.

The process layer contains a PalCom thread library, modelled after SimIO-Process in the Lund Simula system [13]. The threads are built on coroutines in the base libraries, which are supported directly by the Pal-VM and

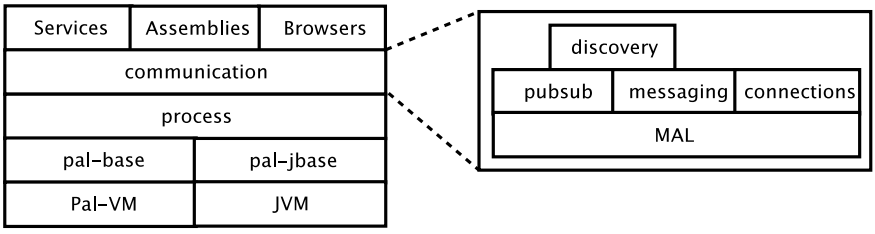


Figure 1: The PalCom stack. Components in the communication layer are zoomed-in on the right.

implemented using Java threads for the JVM. There is a PalCom thread scheduler, which relies on a Unix *select* mechanism for timeouts and non-blocking I/O. There are classes for semaphores, monitors, and event communication between threads. Currently, cooperative scheduling is supported. Plans for future improvements include preemptive scheduling and hierarchical schedulers.

For the actual network communication, the communication components rely on support for different networking technologies in the Pal-VM. On top of this, the components offer APIs, uniform message formats, and addressing. The first kinds of networking supported in the Pal-VM are UDP, unicast and multicast, which suit the asynchronous communication needed by many PalCom applications. UDP is used in the first implementation of the communication component. An earlier design study was based on Bluetooth and PalCom communication is now also developed for Bluetooth.

The box on the right in Figure 1 shows the components in the communication layer. MAL is the Media Abstraction Layer, which abstracts the details of different networking technologies. On top of that are components for publish/subscribe, point-to-point messaging, and point-to-point connections. These will be discussed in the following sections.

5 Basic communication

Our presentation of the communication protocols will start with the basic facilities in the PalCom stack for sending point-to-point and publish/subscribe messages. The discovery protocol and the communication between services is built on top of these two mechanisms: a point-to-point message is intended for one specific receiver, while a publish/subscribe message goes to all receivers that have subscribed to the *topic* on which the message is published. The topics are strings agreed upon in advance by a number

of communicating parties. As an example, our discovery component is implemented using publish/subscribe, with a pre-defined "discovery" topic.

5.1 Point-to-point

The basic type of communication offered by the PalCom communication component is asynchronous message passing: messages are sent without waiting for a reply. In the current Java implementation, with UDP, there is no guarantee that the message will reach its receiver. Adding different levels of such guarantees is possible, and we plan to implement that. This could be done through the use of other transport protocols, such as TCP, or through acknowledgments and re-sends on top of UDP. Our intention is to do the latter, rather than using TCP, because TCP's connected nature is not suitable for the ad-hoc networks and the message-based communication that we target.

PalCom defines a general wire format for messages. This format is used for packaging of point-to-point messages between services, except for those services that use their own, private message formats, as discussed above (those special services implement their communication against the network layer directly). The message format has been selected so that it permits hierarchically structured messages, with messages inside messages at any number of levels. That is used, e.g., for the packaging of publish/subscribe messages, and for packaging of commands sent over connections. Another important point is that the messages should be human-readable, as far as possible. That makes inspection and debugging easier, and helps meeting the PalCom visibility challenge [16]. Finally, care has been taken to keep the messages compact, and make parsing simple. Regarding the hierarchical structure, and the human readability, an obvious choice would have been to base the format on XML. We have not done that for the basic message format, though, because we want to permit binary data in the messages. However, the message formats for discovery and service descriptions, sent as data inside basic messages, are XML-based (see Sections 6 and 7).

The messages are self-addressed: the address of both the sender and the receiver is in the message. In order to obtain the complete sender address from a received message, information from lower layers is used. In the UDP case, the IP address and port of the sender is taken from the UDP packet. Together with the sender selector from the message header (see below), the sender address can be formed.

Figure 2 shows an example of a message. The data in the message are the 8 bytes of the word *palpable*. The message header contains four fields with printable characters, separated by semi-colon: `8` is the length of the message, `d` says that it is a data message, and `1` and `2` are *selectors* for the sender and the receiver of the message. Selectors are assigned by devices for iden-

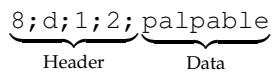


Figure 2: An example message.

tifying local end-points for communication. Note that in this example all bytes are printable characters, including the message data, so the wire format can be shown this way. This is practical for debugging.

Connections

In many of the PalCom scenarios, there are situations where two services exchange a sequence of messages. In order to support this common case, the communication component provides a connection abstraction, which builds upon the message sending described above. To the programmer, a connection is presented as a connection object, in a symmetrical way to both parties. The connection object fills in sender and receiver addresses in all messages sent, and handles shut-down when one party closes the connection. There is a special type of messages, `c`, for connection-control messages.

5.2 Publish/subscribe

Communication in a more loosely-coupled fashion, where the sender and the receiver are not necessarily aware of each other's identity, is also used in PalCom applications. The communication components support the publish/subscribe style of communication [8], where services publish events on different topics, which are delivered to subscribers that have registered interest in matching topics. The discovery component is one example of a component that uses publish/subscribe.

Publish/subscribe messages are packaged as multi-part messages with two parts: one for the topic, and one for the actual message. The data sent out for a publish/subscribe message is shown in Figure 3, where the data in the actual message is the string `palpable` (8 bytes), and the topic string is `words` (5 bytes). The type of the message is `+` for a multi-part message, and the data of the multi-part message contains two messages after each other, which are structured as the example in Figure 2. No selectors are used, because the message is a broadcast message.

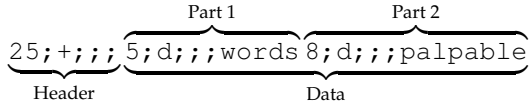


Figure 3: A publish/subscribe message packaged as a multi-part message.

6 The discovery protocol

Taking the requirements of Section 3 into account, we have designed a discovery protocol. Table 1 lists the discovery messages, grouped after messages about devices, services, and connections, with an informal description for each message. An announcement protocol is used for device information, and a pull protocol for the other two. As discussed above, the reason for this is that bandwidth-demanding broadcasts should be limited to the relatively compact information about devices.

The discovery protocol uses the publish/subscribe communication for broadcasts, and the discovery messages are in an XML format. In this section we will give an overview of how the PalCom discovery protocol works, what kind of messages are sent, and what kind of information they contain.

6.1 Discovering devices

There are several situations when a PalCom device needs to know what other PalCom devices are available. Clear cases are when a device is booted, and when it has been moved so the set of reachable devices has changed. Many devices need to keep an up-to-date list of reachable devices and thus need to know when other devices come and go.

The device that needs to know what other devices are available broadcasts the message *DeviceInfoRequest(DeviceInfo)* with the interpretation *"I am the device described by this DeviceInfo—who are you?"* The request, which is an XML message, contains the information about the requesting device, so other devices get that at the same time. Receiving this message, all other devices broadcast a reply *DeviceActive(DeviceInfo)* with the interpretation *"I am the device described by this DeviceInfo, and I am active"*. After this sequence, all the devices have sent one message each, and received one message from each of the other devices. They all thus have had the possibility to update their list of available devices, add new devices, and remove devices that no longer answer, after a short time. The requirements about limited broadcasting, discussed in Section 3.7, are fulfilled.

DeviceInfos are structured according to line 1 of the abstract grammar[3]

[3]The syntax of abstract grammars in this chapter follows that of abstract grammars used by the JastAdd tool [7], which we use in our implementation.

Message	*Broadcast/ unicast*	*Description*
DeviceInfoRequest(Device-Info)	B (request)	*"I am the device described by this DeviceInfo—who are you?"*
DeviceActive(DeviceInfo)	B (reply)	*"I am the device described by this DeviceInfo, and I am active"*
DeviceInactive(DeviceInfo)	B	*"I, the device with this DeviceInfo, am about to shut down."*
ServiceListRequest	U (request)	*"Please send me lists of your services"*
ServiceList*	U (reply)	*"Here are my services"* (may contain several lists)
ServiceDescription-Request(serviceURN)	U (request)	*"Please send me a description of the service identified by* serviceURN*"*
ServiceDescription	U (reply)	*"Here is my service protocol."*
ConnectionInfoRequest	U (request)	*"Please send me information about the connections you have initiated"*
ConnectionInfo*	U (reply)	*"Here are my connections"* (may contain several Infos)

Table 1: Discovery protocol messages.

```
1  DeviceInfo ::= <URN> <Name> <URL>;
2  ServiceList ::= <ParentURN> ServiceInfo*;
3  ServiceInfo ::= <Name> <Type> <Role> <URN> <URL>;
4  ConnectionInfo ::= <ProviderURN> <CustomerURN>;
```

Figure 4: Abstract grammar for information about devices, services, and connections.

shown in Figure 4. They contain very basic information about a device. `URN` is a globally unique identifier of the device, `Name` is a name of the device that can be displayed to the user, and `URL` is an address that can be used for sending requests to the device. Both the URN and the URL are needed, because the URL may change when a device reboots, or perhaps when it moves, and the URN must be stable, so the device can be referred to over a long time. The XML messages for DeviceInfos follow this structure directly (a DeviceInfo element with three attributes).

One crucial design decision is how often the discovery procedure is initiated. If it is done too seldom, a user will notice a delay between when things should become available and when they are actually shown as available. If it is done too often, the resulting messages can overflow basically any network. The design chosen here is to initiate the discovery procedure on demand—that is when needed. The device most eager to know about the others initiates the discovery procedure, and all the other devices will be updated as well as a side effect, because they listen in. We call this scheme *announcement heartbeats*.

The PalCom discovery protocol does not impose a specific strategy for how the discovery mechanisms are used, but this is up to the implementers of PalCom devices. The discovery procedure will typically be initiated by devices running for example a PalCom browser, and thus providing a user interface, which is showing the available devices. A possible strategy for that situation is to send out requests relatively seldom, say if the available list is older than 10 seconds. This will guarantee that the view presented is fairly up to date. If there is another device sending out requests more frequently, the less demanding device will never actually send out its requests. On top of this mechanism, the browser could offer an update button, so the user can refresh the view more frequently when needed. This strategy has been used in our implementation.

A device that is about to be orderly shut down can inform other devices that so is the case. It does so by broadcasting a message *DeviceInactive(DeviceInfo)* with the meaning: *"I, the device with this DeviceInfo, am about to shut down."* Devices receiving this message should remove the sending device from its list of available devices, and disconnect all connections it has with services on that device. Devices can thus be removed from listings on other

devices: either explicitly, after receiving this DeviceInactive notification, or indirectly when a device fails to respond to a DeviceInfoRequest.

6.2 Listing services

When a device has discovered another device through the device discovery procedure above, it knows the address of the device, and the device name can, for example, be shown in a browser application. The device can also request information about the services on another device.

Services on a device can be arranged in a logical tree, as shown in Figure 5. This is useful for grouping on devices that have many services. In order to know what services are available on another device, a device sends a single unicast message to it: a *ServiceListRequest* with the meaning *"Please send me lists of your services"*. Receiving this message, the addressed device will answer over unicast with (a number of) *ServiceLists*, which contain information about the addresses and names of the services. A ServiceList is structured according to lines 2–3 of the abstract grammar in Figure 4. `ParentURN` is the URN of the device or service which is the parent of the services in the list. This can be used for building the tree of Figure 5 incrementally, as ServiceLists come in. Each ServiceInfo contains basic information about one service: name, content type (MIME type), role, URN, and URL. The role can be provider or customer: a customer is a service that uses another service (the provider), as explained above in the introduction. URN and URL correspond to the similar fields in DeviceInfo. Note that the services have a name and an address each. They can thus be addressed as separate entities, although they of course reside on a device.

There are different kinds of situations, where services are offered. A "small device" can reply with a fixed, pre-prepared representation of its services, making this part of the protocol easy to support. Browsers, and other devices that can activate assemblies, need to represent a dynamic list of services, including the services offered by its active assemblies. In the same way, providing execution of *software services* means that the services offered by these must be included in the resulting service list. Software services are instantiated from components written in a general-purpose programming language, such as Java, but are not tied to the hardware of a particular device. Software services can be included in an assembly, for performing more complex calculations than what can be expressed directly in an assembly script.

Before a device can start to use a service on another device, it must know what protocol to use for communication with that particular service. These service protocols are described in *service descriptions*. A device that wants to know the details of a particular service unicasts a *ServiceDescriptionRequest(serviceURN)* to the device of that service (it knows the URN of the service from a ServiceList reply). This means *"Please send me a description*

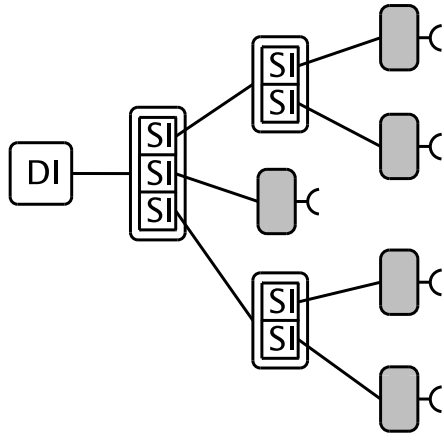


Figure 5: The logical tree of information about services on one device. DI is the DeviceInfo. Under the DeviceInfo, there is a ServiceList, shown as a box with ServiceInfos, SI, inside. Each ServiceInfo has another ServiceList with subservices, or a ServiceDescription that describes the service (grey box with a hook).

of the service identified by serviceURN". The service replies to the requester with a *ServiceDescription* with the meaning *"Here is my service protocol."* Section 7 further describes the structure of ServiceDescriptions.

Service listing is a two-step process, because ServiceDescriptions are envisioned to be large, so it should only be necessary to retrieve those that you are interested in. This is what we refer to as *on-demand service discovery*. The protocol for obtaining ServiceDescriptions is currently a pull protocol. For obtaining up-to-date information about services, when services are added to or removed from a device, customers must periodically send ServiceListRequests and ServiceDescriptionRequests to it. This is a situation we are not fully satisfied and we are considering alternative techniques here.

6.3 Listing connections

It is possible to ask another device what connections it has initiated (thus where it has a service that plays the role of a customer). A device does so by sending the other device the message *ConnectionInfoRequest* with the meaning: *"Please send me information about the connections you have initiated"*. The receiving device will answer with a number of *ConnectionInfos*. Each ConnectionInfo contains the URN of the provider and the customer in a connection (line 4 of Figure 4). This is enough information to re-establish

the connections at the later stage, as needed by the RemoteConnect protocol, described in Section 8. For periodically updating its list of connections, a device polls the other device repeatedly.

7 Service interaction

After having discovered a service, it is possible to request a ServiceDescription for it. There can be several reasons for this. A user might want to directly interact with the service and want his browser to render a user interface as a remote control for it. A second case is that a user may be putting together an assembly, and needs to know what messages the assembly script can accept and send. A third case is that an assembly is about to be activated, and it needs to be verified that the protocol expected by the assembly is actually provided by the service. The content of a ServiceDescription is such that these examples can be handled, provided that the standard description format is used, as discussed in Section 3.1.

It is, however, also clear that there will be applications with demanding needs, exploring the available technology to the limit. In such cases, one might need to use handcrafted optimized service protocols, and it should be possible to do that for PalCom services. It is at this point, however, not decided on exactly how they will be specified in a ServiceDescription. This is a trade-off between flexibility in tailoring a specialized protocol, and conformance with PalCom and the support that can be offered. Here, the further development will be guided by the application prototypes in PalCom.

Requirements on the service descriptions are that they should contain the information needed to connect to the service and make use of it. This involves specifying the messages the service can send and accept, as well as the involved data types.

The structure of a ServiceDescription is illustrated in Figure 6. There are commands, with zero or more parameters, and commands can be grouped hierarchically. It is possible to have groups within groups, at any number of levels. The ServiceDescription is transmitted as one XML element, with groups, commands and parameters as sub-elements. The grammar in Figure 7 shows the structure in a more formal way, and also shows what fields the elements contain. All elements have an `ID` field for identifying them. The notation with colon means that `GroupInfo` inherits from `ControlInfo`. The ServiceDescription has a `ProviderURN` field for linking it to its parent, just as for the ServiceList above (the parent URN is used when building the tree of figure 5, where the ServiceDescriptions are the small grey boxes with hooks). The `Dir` field for commands is the direction of the command. It can be either `in` for *in-going commands*, sent from the customer to the provider, or `out` for *out-going commands*, sent in the other direction. The `Type` field of `ParamInfo` holds the MIME type of the

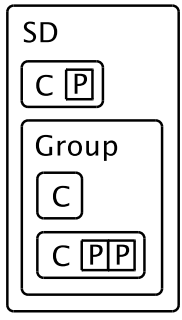


Figure 6: The structure of a ServiceDescription, SD. There are commands, C, with zero or more parameters, P, and the commands can be grouped hierarchically.

```
1  abstract ControlInfo: Info ::= <ID>;
2  GroupInfo: ControlInfo ::= ControlInfo*;
3  ServiceDescription: GroupInfo ::= <ProviderURN>;
4  CommandInfo: ControlInfo ::= <Dir> ParamInfo*;
5  ParamInfo: ControlInfo ::= <Type>;
```

Figure 7: Abstract grammar for service descriptions.

parameter's data.

After having received a ServiceDescription about a provider through discovery, a customer can start interacting with it. This is done by first connecting to the provider, using the connection mechanism described in Section 5.1. Then, messages can be exchanged, according to the contents of the ServiceDescription. The customer sends in-going commands to the provider, and vice versa for out-going commands. The commands are sent as XML documents called `Command`, with inner `Param` elements for the parameters. These are structured like `CommandInfo` in Figure 7, but also contain data values for the parameters. This data is sometimes binary, so it is not transmitted inside XML elements. Instead, an XML element `Command` is packaged together with its parameter values in a multi-part message.

8 Remote connect

The need for being able to remotely connect two services was motivated in [22]. This is important for being able to construct ad-hoc applications in a browser, and for assemblies. Table 2 shows what messages are sent in the small protocol used for this. In order to initiate a connection between ser-

Message	*Broadcast/ unicast*	*Description*
RemoteConnect(ConnectionInfo)	U	*"Please establish this connection"*
RemoteDisconnect(ConnectionInfo)	U	*"Please close this connection"*

Table 2: RemoteConnect messages.

vices on two other devices, a device sends a *RemoteConnect(ConnectionInfo)* message to the service that plays the role of customer. This means *"Please establish this connection"*. The ConnectionInfo describes the desired connection, and the customer establishes it. This can be initiated by a direct user action in a browser, or indirectly, when an assembly is activated. *RemoteDisconnect* is the counterpart to RemoteConnect, with similar motivation. A message *RemoteDisconnect(ConnectionInfo)* is sent to the service playing the role of customer. In response to this, the customer closes the connection.

9 Evaluation

Our main way of evaluating the described protocols is by continuously using them in developed prototypes and tools. This has been done in a number of simulated devices and in a browser application [22], using the Java implementation of the communication components described here. In these examples, the implemented components and the protocols have worked well, and we are confident that the protocols will be adequate in our continued development of assemblies. The asynchronous mode of communication in the discovery protocol is handled by building a tree structure, like in Figure 5, incrementally as information comes in.

The implemented communication components are now also being used in an Eclipse plugin [5] for PalCom browsing and assembly development. Another project implements support for the PalCom protocols directly in C, on an Axis network camera running Linux [2]. A part of this work is an investigation of how the protocols can be implemented on a small device, without a virtual machine.

9.1 The discovery protocol

It is difficult to evaluate the discovery protocol quantitatively, by comparing it to existing protocols. The performance of any discovery protocol depends very much on the environment, and on parameters that vary be-

tween protocols. In this section, we will try to analyze and discuss how it will behave with respect to different design goals, and in particular its potential for scaling up to larger networks.

The approach taken in the discovery protocol is based on broadcast without a central directory. Initially we presume a situation with a single network with a limited number of devices. The devices available are those that are visible on that network.

Protocol-inflicted delays It is a design goal that the responsiveness of PalCom is sufficient for interactive work. From a user perspective this can be illustrated with the situation when a user studies the available devices in a PalCom browser. How long will it take before a change in the number of available devices is reflected for the user in the worst case?

There are five cases of how the situation can change, where X is one of the devices:

1. Device X boots.
2. Device X is shut down uncleanly.
3. Device X is shut down cleanly.
4. Device X comes within reach.
5. Device X leaves.

In the discovery protocol, there will be direct notification to all other devices in case 1 and 3. Here, the protocol will thus not inflict any extra delay. In the other three cases the change will be noticed at the next round of the discovery procedure. The time from the change takes place until it is actually reported depends on the time until the most eager device initiates a new discovery procedure. This time is thus determined by the applications, and not by the discovery protocol itself. It can for example be set very short for a while by a device that is in a critical state. This can be lifted all the way up to the end user to control.

Protocol inflicted communication load Another design goal is that the protocol must be efficient from the point of view on putting load on the communication channel. It is a PalCom design choice that there is no central server.

When a new device becomes present (boots or comes within reach) it needs to get information about all other available devices. In an environment with N devices, this requires one initial request and $N-1$ replies, which is what the protocol will use. Since these messages are sent as broadcasts it means that at the same time all other devices are brought up to date as well (including registering the new device).

For updating a situation with N devices we require N messages. If single messages (unicast) had been used for answering the request there would have been needed $N * N$ messages.

There are several areas where more examination and experimentation are needed. The following two areas are potential possibilities for the future evolution of the model:

Large numbers of discoverable elements In large networks, the number of discoverable services can potentially become overwhelming for the user. One possible way to help the user in such a situation is to offer some query and filtering mechanism. The specification of such mechanisms are studied in PalCom (resource and contingency management).

Discovery in complex networks The discovery protocol described in this paper has primarily been developed for the situation in a local network, and for meeting high demands on responsiveness and with less restrictions on broadcasted traffic. In a situation with a wide area network, these design goals change. The demands on responsiveness are relaxed, the available services are less dynamic and the broadcasted traffic must be kept to a minimum. For this situation it might be that the discovery model will need to be extended with some service directory mechanism. One possibility here is to combine such a mechanism with a gateway between a local area network and a wide area network.

10 Related work

As mentioned in the introduction, there are many existing protocols for discovery of, and interaction with, services and devices in various types of networks. The protocols target different levels of communication, and also different network technologies and complexities. Therefore, it is difficult to compare them quantitatively. As an example, DEAPspace [14] focuses on single-hop short-range wireless systems, where low power consumption is crucial, while Jini [25] targets larger and more fixed networks, where devices can be more powerful.

Zhu et al. [26] have made a taxonomy for discovery protocols within the area of pervasive computing. Nine protocols are characterized along ten different dimensions. There is not room here to present the complete categorization, but we can look at how the PalCom protocols relate:

1. *Service and attribute naming.* PalCom uses a template for how services are named, but there are no predefined attributes that can be used for further describing services.

2. *Initial communication method.* The PalCom protocols make use of broadcast initially (multicast on IP networks), not unicast which requires prior knowledge of device addresses.

3. *Discovery and registration.* The discovery is announcement-based, not query-based, with the variant that clients send out announcements, that at the same time trigger announcements from services.

4. *Service directory infrastructure.* The PalCom discovery protocol is non-directory-based.

5. *Service information state.* The PalCom model here is more like *hard state*, where clients poll services for getting sufficiently up-to-date information, than like *soft state*, where services put a lifetime on announcements, which are considered valid until they time out, or are renewed.

6. *Discovery scope.* It will be the network topology that delimits the scope of discovery, not user roles or context. Investigating these mechanisms has not been our focus, though.

7. *Service selection.* Service selection is manual, not automatic. It is the user who chooses what service to use.

8. *Service invocation.* Here, PalCom is at the first or second level of the three levels given by Zhu et al. It is at the first level, where only a service location is provided through the discovery protocol, for the case where a non-standard service protocol is used. In the most common PalCom case, where the standard message formats are used, it is at the second level, defining also the communication mechanisms. PalCom is never at the third level, where domain-specific application operations are defined.

9. *Service usage.* Service usage is explicitly released, and *leasing* is not used. We are investigating timeouts for connections, which can be compared to a lease-based scheme.

10. *Service status inquiry.* Both *service event notification* and *polling* are explicitly supported, considering the peer-to-peer style of communication between a provider and a customer, after a service description has been transferred, and a connection established.

Looking at the categorized protocols in the paper by Zhu et al., UPnP [23] is the one which is most similar to the PalCom protocols, with similar choices in five of the ten dimensions (4, 6, 7, 9, and 10). DEAPspace and Salutation [18] have four dimensions the same. Most interesting is probably where PalCom differs from many protocols:

- All the protocols except DEAPspace (and PalCom) support query-based discovery.
- Only UPnP supports both notification and polling for service status inquiry, like PalCom does.
- The announcement strategy in the PalCom protocols are different from the standard announcement procedure described in the article. In PalCom, the frequency of announcements is controlled by the most eager device, as discussed above.

Having categorized our protocols according to a taxonomy like this, we would also like to point out that the choice of dimensions is of course important. A dimension that could be added to the taxonomy is the description format, where we have chosen a textual (XML) format for human readability, not a binary format. It is also interesting to see whether the protocols use mobile code or not. Jini does so, but UPnP and the PalCom protocols do not. The reason for this, in the PalCom case, is that it puts requirements on the run-time systems of devices.

Being most similar to UPnP, we will now compare in more detail with that. Similarities are that UPnP is XML-based, that it does not make use of mobile code, and that the primary focus in the discovery protocol is on devices, not on services. There is also a correspondence in the split between the UPnP *discovery step*, where only "a few essential specifics" about devices and services are distributed, and the *description step*, where detailed information is retrieved. This is similar to how the PalCom discovery protocol separates between sparse information in DeviceInfos and ServiceLists, and rich information in ServiceDescriptions (the *on-demand service discovery*). Differences, compared to UPnP, are that UPnP is built on IP networks, while PalCom supports also, e.g., Bluetooth, and that UPnP makes use of SOAP over TCP. The latter is a protocol that works much like Remote Method Invocation (RMI), in that the caller waits for a synchronous response from the callee. As discussed in the introduction, that is not suitable for ad-hoc networks. A third difference is that in UPnP, it is the advertising device or service who determines the duration of advertisements, while in PalCom it is the most eager device. Finally, at another level, the UPnP consortium has taken an approach of domain-level standardization. This is something we have avoided, as discussed in Section 3.1.

11 Conclusions and future work

This paper has reported on developed discovery and communication protocols for palpable computing. We are confident that the protocols will work well as a foundation for our continued work on ad-hoc applications.

The discovery protocol distributes information about devices, services, and connections, and the distributed service descriptions can be used for communicating with a service according to the right service protocol. This service communication can be done directly from another service, from a user interface rendered in a browser, or from an assembly that coordinates several services.

The protocols are based on asynchronous communication. A request is typically sent in one message, and as replies come back, they can be handled by incrementally building a data structure of devices and services in the vicinity. This approach has worked well in our implementations.

The development of the protocols has been guided by requirements for ad-hoc combinations of devices and services, and by requirements from prototypes developed within PalCom. One of the requirements is that of *device awareness*: not only services, but also devices, should be visible through discovery. Another is that of *on-demand service discovery*: for efficiency and bandwidth reasons, it must be possible to acquire service information incrementally, and for only those services one is interested in. A new device discovery procedure has been proposed, named *announcement heartbeats*, which is based on broadcasted requests, where all devices on the network listen in on the broadcasted replies. This way, all devices will be updated as a side effect, when one device makes a request. It is the device most eager for fresh information about other devices that controls the frequency of requests, and thus how fast device information spreads in the network.

A protocol stack has been implemented, which runs both on the Pal-VM (PalCom's new virtual machine), and on the JVM. The implementation uses a thread library built on top of coroutines, with a scheduler based on Unix *select*.

Future work on the protocols includes support for large numbers of discoverable elements, and for discovery in complex networks.

Acknowledgments

This work has been conducted within the PalCom project, EU-IST 002057. The protocols have been developed in collaboration with other people within the project, among them Jacob Frølund and others at Aarhus University. Of the implementations mentioned in the paper, the Pal-VM implementation and the Smalltalk base libraries have been developed mostly at Aarhus University. We would also like to thank Torbjörn Ekman, the main architect and developer behind the PalCom Java compiler, for valuable help with many issues related to running on top of the two VMs.

References

[1] Apple. Bonjour. `http://www.apple.com/macosx/features/bonjour/`.

[2] Axis Communications. Network cameras. `http://www.axis.com/products/video/camera/index.htm`.

[3] Bluetooth.com. The Official Bluetooth®Wireless Info Site. `http://www.bluetooth.com/`.

[4] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha. GSD: A Novel Group-based Service Discovery Protocol for MANETs. In *4th IEEE Conference on Mobile and Wireless Communications Networks (MWCN 2002)*, 2002.

[5] Eclipse.org. Eclipse.org home. `http://www.eclipse.org/`.

[6] W. Keith Edwards. *Core Jini*. Prentice Hall, 1999.

[7] T. Ekman, G. Hedin, and E. Magnusson. JastAdd: an open-source Java-based compiler compiler system. `http://jastadd.cs.lth.se`.

[8] P. Th. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[9] Internet Engineering Task Force. *Service Location Protocol, Version 2*, 1999. `http://www.ietf.org/rfc/rfc2608.txt`.

[10] Samuel C. Kendall, Jim Waldo, Ann Wollrath, and Geoff Wyant. A Note on Distributed Computing. Technical Report TR-94-29, Sun Microsystems, November 1994.

[11] M.J. Kim, M. Kumar, and B.A. Shirazi. Service discovery using volunteer nodes in heterogeneous pervasive computing environments. *Pervasive and Mobile Computing*, 2(3):313–343, 2006.

[12] Margit Kristensen, Morten Kyng, and Esben Toftdahl Nielsen. IT support for healthcare professionals acting in major incidents. In *Proceedings of SHI2005, 3rd Scandinavian conference on Health Informatics*, Aalborg University, August 2005.

[13] Boris Magnusson. *Using the simioprocess library on Unix Systems*. Lund Software House AB, August 1997.

[14] M. Nidd. Service Discovery in DEAPspace. *IEEE Personal Comm.*, pages 39–45, August 2001.

[15] Ulrik Pagh Schultz, Erik Corry, and Kasper V. Lund. Virtual Machines for Ambient Computing: A Palpable Computing Perspective. In *ECOOP 2005 Object Technology for Ambient Intelligence Workshop*, Glasgow, U.K., 2005.

[16] PalCom. Palpable Computing: A new perspective on Ambient Computing. `http://www.ist-palcom.org/palcom-info.pdf`.

[17] Golden G. Richard III. *Service and Device Discovery: Protocols and Programming*. McGraw-Hill Professional, 2002.

[18] Salutation Consortium. Salutation Architecture Specification, 1999.

[19] Titos Saridakis. A System of Patterns for Fault Tolerance. In *Proceeding of EuroPLoP 2002, Seventh European Conference on Pattern Languages of Programs*, Irsee, Germany, July 2002.

[20] Sun. *Java Remote Method Invocation Specification*, 2003.

[21] David Svensson, Görel Hedin, and Boris Magnusson. Pervasive applications through scripted assemblies of services. In *Proceedings of SEPS '06, 1st International Workshop on Software Engineering of Pervasive Services*, June 2006. To appear.

[22] David Svensson, Boris Magnusson, and Görel Hedin. Composing ad-hoc applications on ad-hoc networks using MUI. In *Proceedings of Net.ObjectDays 2005, 6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, pages 153–164, Erfurt, Germany, September 2005.

[23] UPnP™ Forum. UPnP™ Device Architecture 1.0. Technical report, December 2003. Version 1.0.1.

[24] Aino Vonge Corry, Tony Gjerlufsen, and Jesper Wolff Olsen. The Stone: Digital support for (un)common issues during pregnancy. In *Proceedings of SHI2005, 3rd Scandinavian conference on Health Informatics*, Aalborg University, August 2005.

[25] Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, pages 76–82, July 1999.

[26] Feng Zhu and Matt W. Mutka. Service discovery in pervasive computing environments. *IEEE Pervasive Computing*, 4(4):81–90, October–December 2005.

Paper IV

Pervasive applications through scripted assemblies of services

David Svensson, Görel Hedin, and Boris Magnusson
Dept. of Computer Science, Lund University, Sweden

{david|gorel|boris}@cs.lth.se

ABSTRACT

This paper proposes a technique for letting end users build pervasive applications by combining services on networked devices. The approach avoids relying on standardized service interfaces which are deemed too limiting, and instead makes use of migratable user interfaces and scripted combinations of services.

1 Introduction

In a world of pervasive computing, people will encounter a wealth of devices that offer software services in (typically wireless) networks. These services will often be tied to the particular devices, enabling control of and interaction with the devices in powerful ways. We argue that in this setting, interoperability is bound to become a major challenge.

A typical need that can be foreseen is the possibility to combine services, utilizing the combined functionality of several devices. Support for this can facilitate repeated use of a set of connected devices, and also provide inherently new functionality, not given by the individual devices themselves.

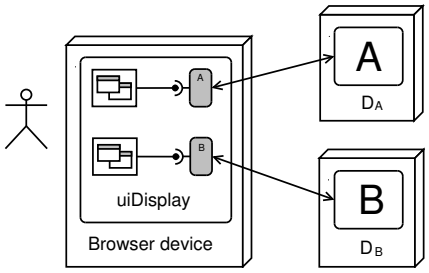
However, the device vendors cannot be expected to foresee all possible combinations of services that can be demanded by future users; combinations possibly including future services and devices. This makes the usual approach, where one service interacts directly with another service through a standardized service-specific interface, too limiting. Instead, we propose that the combination of services should be separated from the services themselves, and that this combination is scripted, rather than programmed, to make it easy to adjust by end users. This will allow individual services to be developed independently of other services, but still be integrated into combined services.

In this paper, we describe a mechanism called *scripted assemblies*, that supports such combination of independent services. We have built an experimental system based on this approach, and tried it out on example scenarios. The ideas build on the MUI system [7], that supports remote control through migratable user interfaces. The work has been carried out within the EC-funded integrated project PalCom [5].

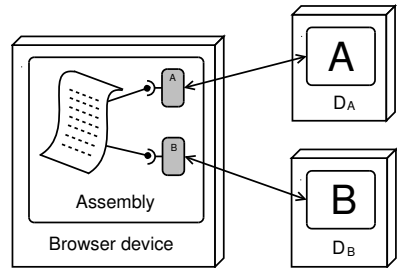
The rest of this paper is organized as follows. In Section 2, we present our basic approach. Sections 3 to 5 go into more detail about non-scripted and scripted assemblies, dealing with issues in the assembly description language. Section 6 relates to other work in this area, and Section 7 lists some things that remain to be investigated and developed. Finally, Section 8 concludes the paper.

2 Basic approach

Figure 1 illustrates our approach to dealing with interoperability. There are two services, A and B, located on two different devices, D_A and D_B. The user wants to use and combine these two services. To accomplish this, the user has a third *browser* device that supports device and service discovery. Typically, the browser device is a handheld like a PDA or a mobile phone, but it could also be a general-purpose computer, e.g., a laptop. Each of



(a) The user interacts with two services, A and B, through user interfaces that are rendered on his browser device. The user interfaces are generated according to service descriptions (shown as small boxes with "hooks").



(b) An assembly on the browser device coordinates A and B, automating the desired combined behavior.

Figure 1: Our approach to interoperability between services.

the services A and B has a service description that can be migrated to the browser device, and rendered as a user interface there, in order to remotely control the service.

In Figure 1(a), the user interacts with A and B through these migrated user interfaces that are rendered on the browser device. This remote control mechanism is provided by a service *uiDisplay* that runs on the browser device. The uiDisplay service receives service descriptions from remote services, creates corresponding user interfaces that are shown on the screen of the browser device, and connects to the services over the network.[1] In the figure, the service descriptions are the gray boxes with a small "hook". After this setup process, there is a two-way, peer-to-peer communication between uiDisplay and each of the remote services A and B. When the user performs an action in one user interface, a command is sent over the connection to the remote service, which can react appropriately. When something happens at the remote service, typically as a result of physical interaction with the device on which the remote service lives, a command is sent in the other direction, typically updating status information in the user interface on the browser device.

The interaction through user interfaces solves parts of the problems with standardization of service interfaces. In this case, the human is in the loop, and can make intelligent interpretations of changes in the service descriptions (which show up in the user interfaces). When a new feature is added to a device, perhaps through an update of its firmware, a change in the service can be directly spotted and utilized. There is nothing in uiDisplay itself that is tied to the specific service.

[1] The migration of user interfaces, and adaptation to different client platforms, is a research area in itself. See, e.g., [6].

The other part of the figure, 1(b), shows how interaction with the services can be automated, realized through an *assembly*. The assembly is a service which performs much the same function as uiDisplay, but instead of rendering service descriptions as user interfaces, it coordinates the services according to an *assembly descriptor* (shown as a scroll). The assembly descriptor contains a specification of a combination of services, residing on different devices. Simple assembly descriptors just specify a number of connections between the services, while more advanced assembly descriptors also contain a script, which coordinates the interaction in a more fine-grained way. The assembly shown in Figure 1(b) is of the latter type. It specifies how different events received from A lead to one or more commands sent to B.

The key point in our approach is that *interoperation is separated from services.* This is what makes it possible to combine groups of services that were not created together, without restraining all of them to use standard service interfaces that were already established when the oldest service was created. In the case of user interface rendering, the user controls the interoperation directly, something which is good for trying things out in order to see how they work. This can be impractical, though, for more complex or long-lasting tasks. For these, the assembly works better. The goal is that assemblies should be possible to create and modify by end users. For this reason, we propose that assemblies should be created using a scripting language, rather than using a general-purpose programming language that would require programming skills. In some cases, the interoperation of two services might, however, require programming. For example, in order to convert between different kinds of real-time data streaming formats. To handle this, we propose that such problems are delegated to separate *software services*. Such software services need to be programmed, but can be used and combined in an assembly by an end user.

3 Assembly representations

Our model of an assembly consists of the following parts:

1. A set of devices.
2. A set of services (on those devices).
3. A set of connections between those services.
4. A set of offered synthesized services, generated by the assembly.
5. Logic and scripts defining and constraining how the assembly should be deployed and executed.

Furthermore, an assembly can be fully bound, forming a composition of particular identified services on particular identified devices, or it can be in various ways partially unbound, e.g., to act as a template. In this paper, we will focus on fully bound assemblies.

It is useful to discuss the assembly from several different perspectives: the end user, the expert user, the tools manipulating the assembly, etc. In our system, we use the following important representations of the assembly:

1. An XML representation that is used for storing the assembly in a file system, and for moving or copying it between different devices.

2. A concrete syntax that is used in documents like this to show the same details as in the XML representation, but in a syntax more readable by humans. In principle, this concrete syntax could also be used by editing tools on laptops for creating or editing assemblies by expert users.

3. A representation as an attributed abstract syntax tree (AST) that is used internally by tools accessing and manipulating the assembly. We use the JastAdd compiler construction system for supporting AST programming [1], allowing the internal tools to add computations on the AST as modular aspects. The XML and concrete syntax can trivially be unparsed from the AST representation, programmed as simple JastAdd aspects.

4. Tool-specific editing representations for displaying parts of the assembly information to end users, often in a visual way. E.g., a PalCom browser device can display the connections between services as lines between boxes, and provide graphical commands for composing or changing an assembly.

In this paper, we will use the concrete syntax when providing examples. In Section 4 we will discuss simple assemblies with devices, services, and connections, but without scripts. In Section 5 we will discuss how scripts can be added to capture the execution logic of an assembly.

4 Simple assemblies

4.1 A remote slide show assembly

Some simple assemblies consist only of a set of connections between services on particular devices, and have no logic of their own. As an example, consider an assembly RemoteSlideShow which composes a video projector, a laptop, and a PDA. Slides are sent from the laptop to the video pro-

```
assembly RemoteSlideShow {
  this = global-service-name;
  devices {
    projector = global-device-name;
    laptop = global-device-name;
    pda = global-device-name;
  }
  services {
    control on laptop = global-service-name;
    images on laptop = global-service-name;
    uiDisplay on pda = global-service-name;
    imageViewer on projector = global-service-name;
  }
  connections {
    control on laptop -> uiDisplay on pda;
    images on laptop -> imageViewer on projector;
  }
}
```

Figure 2: A simple assembly.

jector, and the user controls the actions, next slide, previous slide, etc., from the PDA. The assembly itself resides on the PDA.

To illustrate the use of this assembly, consider the following scenario:

> The user is a university professor who has weekly lectures in room E:1406 at the university. On the first lecture, she creates the assembly RemoteSlideShow by connecting her laptop, her PDA, and the video projector in room E:1406. This is done by a few visual commands on the PDA. On the PDA, she then uses commands to select the desired presentation, and to step through the slides. At the next lecture, she simply activates the existing assembly, which will then discover and connect the devices according to the assembly description. She can then immediately select the appropriate presentation and step through the slides.

4.2 Local device and service names

The RemoteSlideShow assembly is shown in Figure 2 in concrete syntax. The assembly introduces a number of local device names: projector, laptop, pda; and a number of local service names: control, images, uiDisplay, and imageViewer. These local names are used within the assembly, e.g., to define the connections, and inside assembly scripts (discussed later).

Typically, the local names are taken from the logical names used in the device and service descriptions of actual devices. But refactoring to other names inside the assembly (for greater readability), would not affect the

behavior of the assembly. These names are not used for binding the assembly to real devices and services. For such binding, the global names are used (see below).

In the proposed language, the service names are simple rather than structured. However, it could easily be generalized to support structured names. This would be useful since the services on a device are typically structured in a hierarchy, and it would be useful to keep that hierarchy in the local names of the assembly.

4.3 Global names of devices and services

Devices and services are identified by globally unique names. The globally unique names have an internal structure including a globally unique identifier, versioning information, and a logical name (which does not need to be unique). Typically, these names are quite long, and not intended to be very readable to a human. In the example in Figure 2, we simply display them as "global-device-name" and "global-service-name". These unique names are used for making it possible to reconnect an assembly to the same devices and services as used when the assembly was constructed. The assembly also has such a unique name itself (the value of "this"), with the same structure as a service name. The versioning information in the globally unique names is used by tools to make safe upgrades of an assembly when a service or a device has been upgraded. Note, however, that when an assembly is upgraded (rebound device and/or service), this has to be somehow visible to the user, and testing might be needed (unless it can be deduced that testing has already been carried out).

Two different devices, e.g., two projectors, can have (different instances of) the same service on them. To uniquely identify a service instance, i.e., a service on a particular device, the global names of the hosting device and the service are combined.

4.4 Connections

The connections part in the assembly specifies how the services in the assembly are connected to each other, using clauses on the following form:

```
providing-service on device-1 ->
customer-service on device-2
```

Connections can be either data connections (uni-directional), sending messages from provider to customer, or control connections (bi-directional), where messages can go in both directions. For example, in Figure 2, the connection `images on laptop -> imageViewer on projector` is

a data connection where JPEG images are sent from the laptop to the projector. The connection `control on laptop -> uiDisplay on pda` is a control connection. As an example of messages over this connection, the PDA can send a message "next" to the laptop, to go to the next slide. The laptop will then send a status message back to the PDA, showing the name of the currently shown slide.

MIME types are used for specifying the types of connections. For example, the type of the `image-imageViewer` connection is `image/jpeg`, whereas the type of the `control-uiDisplay` connection is `application/x-palcom-control+xml`. These types are not explicitly visible in the assembly, but belong to service descriptions that are available for each device through the discovery protocol. The service description also classifies a service as being either provider or customer.

4.5 Static-semantic constraints on the assembly

There are certain semantic constraints on how assemblies may be constructed. The local names should be declared and used correctly. E.g., two devices named the same way is forbidden, using an undeclared name is forbidden, using a device name where a service name is expected is forbidden, etc. This boils down to normal name and typechecking rules similar to those in simple programming languages.

4.6 Dynamic constraints on the assembly

There are additional semantic constraints that can be checked only dynamically, i.e., when trying to activate the assembly:

- *Device bound.* When activating an assembly, the device declarations will be bound to descriptions of actually discovered devices. Naturally, it may be the case that it is not possible to discover a given device. It might be broken, turned off, not within range, etc. The operation of the assembly may then be limited for the moment.
- *Service bound.* Even if a device is bound, it is not guaranteed that all its services are available. Some services may be down, depending on the state of the device. It might also be the case that when an assembly is changed so that a device declaration is rebound to another device, the new device does not have all the declared services, and this is then flagged as errors or warnings. The tools can then guide the user in trying to rebind to another service on the same device, or possibly to a service on another device.
- *Connection well formed.* If the services of a connection are bound, it is checked that the connection is well formed. I.e., the service declared

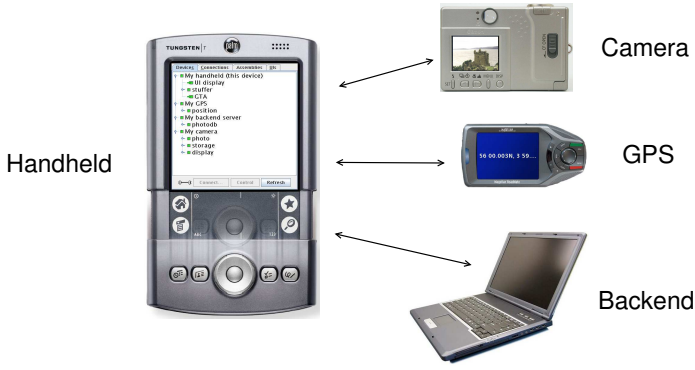


Figure 3: The GeoTagger scenario

as the providing service should indeed be specified as a providing service in its service description, and similarly for the customer service. Furthermore, the MIME types of the provider and the customer should match.

5 Scripted assemblies

In the RemoteSlideShow example, the assembly simply connects existing services directly to each other. A more advanced assembly can itself receive and send messages and perform actions internally. These actions are written in a simple script language that can be used by an end-user. In the present experimental system, the script is edited as text, but in future versions, we plan to provide visual tools for editing the scripts. If the internal logic is more complex than the script language can handle, parts of the logic can be delegated to new software services, programmed in a general-purpose language.

Below, we extend the assembly representation to include a scripting possibility. The basic idea is that the assembly can be connected to other services to receive and send messages. The body of the script is an event handler that receives messages from other services and acts upon them. The possible actions (supported so far) are to send messages to other services and to store values in variables local to the script.

5.1 GeoTagger as a scripted assembly

GeoTagger is one of the end-user scenarios studied in PalCom, see Figure 3. It is an application intended for use by landscape architects. The idea is that photos taken with a camera should be automatically tagged with the

current GPS coordinates and stored in a backend database on a laptop. This application is realized as a scripted assembly running on a handheld PDA. The assembly combines and coordinates services running on the camera, the GPS device, and the laptop.

Figure 4 shows the scripted assembly. In the event handler, clauses are written as

```
when message from service on device {
   actions
}
```

where the actions can access data in the message, send new messages to other services, and perform simple computations (assignments of local variables).

The service coordStuffer on the PDA device is a software service that can receive an image in JPEG format, and a GPS coordinate, and which sends out an image tagged with the GPS coordinate. This is a typical example of a computation that is too complex to express directly in the scripting language, and that is instead implemented as a software service.

As shown in the example, the assembly interacts with other services by receiving and sending messages. Thus, the assembly implicitly plays the role of a service that connects to the other services. The "this" expression used in the assembly script refers to the assembly itself viewed as a service. Received messages that are not listed in the event handler are simply ignored.

5.2 Additional constraints on the assembly

The introduction of the script in the assembly makes it possible and necessary to check additional constraints, statically and dynamically. In the static part, the name and type analysis is extended to the local variables. In the dynamic part, it is checked that the bound services actually have the incoming and outgoing messages used in the script, with the appropriate message structure.

5.3 Loopback mechanism

It might be the case that the assembly is located on the same device as some of the other services. A loopback mechanism is used which allows the assembly to communicate in the same way with these services as with services on other devices, without causing any messages to go out unnecessarily on the network.

```
assembly GeoTagger {
  this = global-service-name;
  devices {
    gps = global-device-name;
    camera = global-device-name;
    backend = global-device-name;
    pda = global-device-name;
  }
  services {
    gps on gps = global-service-name;
    photo on camera = global-service-name;
    storage on camera = global-service-name;
    display on camera = global-service-name;
    coordStuffer on pda = global-service-name;
    photo_db on backend = global-service-name;
  }
  connections {
    gps on gps -> this;
    photo on camera -> this;
    storage on camera -> this;
    display on camera -> this;
    coordStuffer on pda -> this;
    photo_db on backend -> this;
  }
  script {
    variables {
      text/plain latestReadableCoordinate;
      text/nmea-0183 latestStandardCoordinate;
    }
    eventhandler {
      when position from gps on gps {
        latestReadableCoordinate = thisevent.WGS84;
        latestStandardCoordinate = thisevent.NMEA-0183;
      }
    when photo_taken from photo on camera {
        send show(latestReadableCoordinate) to display on camera;
        send sendme_photo() to storage on camera;
      }
    when photo from storage on camera {
       send sendme_stuffed_image(
          latestStandardCoordinate, thisevent.Photo)
        to coordStuffer on pda;
      }
    when stuffed_image from coordStuffer on pda {
       send store_photo(thisevent.Image) to photo on backend;
        send store_photo() to storage on camera;
      }
    }
  }
}
```

Figure 4: A scripted assembly.

The loopback mechanism is used also if the assembly connects two services on the same device: the network is transparent, and messages between services will only go out on the network if the services are on different devices.

Note that it is often the case that services on the same device are tightly bound and communicate with each other directly (not via an assembly). For example, when taking photos with a digital camera, the photos will be stored locally on the camera. This process is a bottleneck and needs to be carried out as efficiently as possible, to allow pictures to be taken at high speed.

Assemblies for connecting services on the same device are useful when the services are more unrelated, i.e., when they could in principle be located on different devices, but just happen to be located on the same device.

5.4 Moving the assembly?

For a scripted assembly, its location can dramatically affect the efficiency. For the GeoTagger, there will be large messages sent that include JPEG images. Suppose the assembly is located on the PDA (a natural choice since an assembly interpreter will need some kind of general-purpose platform to run on). In this case, JPEG images will be sent from the camera to the PDA, coordinates added to the image on the PDA, and the "stuffed" image is sent to both the camera and the laptop backend.

Clearly, if the coordinate stuffer and the assembly were moved to the camera or to the laptop, network traffic would be substantially reduced. It might be possible to move them to the camera if it is sufficiently advanced to serve as a general-purpose software service platform. And moving them to the laptop should be possible, but then the assembly would rely on the laptop which might be heavy for the user to always carry with him.

Note that if the assembly is moved to another device, its script does not need to change. There is nothing in the assembly script that makes it depend on its own platform.

If the assembly and coordinate stuffer are moved to the camera or laptop, it might still be the case that the user would like to control the assembly from the PDA, e.g., to activate it. Future versions of our system will support this by using an uiDisplay service for remote control of assemblies.

6 Related work

The scripted assemblies we propose are related to W3C's Web Services Choreography Description Language [2]. WS-CDL *choreographies* are expressed in an XML language, and govern peer-to-peer interoperation between a number of services. Like our assemblies, the choreographies are

external to all the participating services. One thing that differs is the context. WS-CDL is intended for E-business, taking place between Web services on the Internet. There is no notion of physical devices, which are important in our approach, and in pervasive computing in general. The purpose of WS-CDL is also not on keeping interoperability between services when facing service interface changes. Instead, the choreography is more like a contract that is decided on before a business relationship is started, making it possible for all parties to keep the internals of their services private.

Another closely related project is Obje at PARC [4]. Obje targets the same basic problem, and seeks to enable interoperability without relying on domain-specific standards. A difference is that Obje builds on mobile code. Using mobile code, in the form of a proxy object that is distributed to clients and executed there, services are able to "teach" clients how to communicate. This way, it is possible to let users combine their clients with new services, some of whose features were unknown at the time the clients were written. There is also a possibility to let the proxy object generate a user interface, giving a situation similar to that of Figure 1(a), where the proxy object corresponds to our service description. Another difference, though, is the way of programmatically interfacing the proxy objects. Obje proxy objects are (Java) code, which requires the capability of running Java on clients, and their interfaces are so called *meta interfaces*, offering only very generic operations, such as reading a chunk of data. In contrast, our service descriptions are distributed as XML, which can be handled on almost any device, and they contain domain-specific operations: the operations are invoked by the user through a user interface, or by the assembly script. There is no concept in Obje corresponding to the assembly. Instead, the user directly connects components written in Java.

Cooltown at HP Labs [3] is an early pervasive computing project, whose target is to bring the Web to things in the physical world. By embedding wirelessly accessible web servers into things, it is possible for a user to interact with them in a Web browser on his handheld device. It is also possible to connect one device to another, by sending a URL to one of the devices, identifying a resource on the other. There is nothing domain-specific in the Web protocols involved, so this can be seen as a way of achieving basic parts of the interoperability we look for. But, apart from the client-server model being inherent in the interaction between Web clients and servers, one big difference is that our assemblies can define other aspects of a service interoperation than just pure connections.

Jini and UPnP are important technologies for network services. Jini [11] is tied to the Java programming language, and clients interact with services through proxy objects, distributed to the clients at discovery time. Our objection, also stated by Obje, is that this approach requires the interfaces of the proxy objects to be standardized at the domain level. To partly over-

come this, there is a framework for user interface services built on top of Jini [10]. But, still, the tight connection to the Java language makes it inconvenient to build assemblies on top of Jini. UPnP [9] is not tied to Java, or to another programming language: devices and services are described using XML. But the focus in UPnP is on standardization of device types at the domain level. There are standards for devices such as printers, scanners, lighting controls, and digital security cameras, among others [8]. Therefore, UPnP is not directly usable as a platform for assemblies, either.

7 Future work

In our continued work on scripted assemblies, we will look into a number of issues including synthesized services, binding of services, message types, and service versions. Synthesized services are services that are offered by the assembly itself, allowing control of the combination of services, rather than of each service individually. A simple case of a synthesized service could be to collect the most important parts of the participating services' interfaces into one interface, for convenience. Another case could be to have an interface for changing the activity state of the whole assembly.

So far, we have considered only fully bound assemblies where each service declared in the assembly is bound to a specific service on a specific device. We will look into more elaborate support for service bindings. One example is to investigate cases where an assembly can be functional without all services being present. This may give degraded, but acceptable, functionality of the assembly. In other cases, it may be enough with one out of a set of services for full functionality. There are also possibilities for experimenting with partially bound assemblies, where the identity of a device or service is not filled in, but can be specified later, perhaps when the assembly has been moved into a new context. In relation to this, versioning of assembly descriptors becomes important.

Currently, we demand exact matching of MIME types for connecting services. However, we will investigate the use of subtyping to allow services to be connected where the types match only partially.

8 Conclusions

This paper has presented scripted assemblies as a technique for letting end users combine services, and for letting them control the cooperation between services in a script. The assembly concept allows the interoperation between services to be separated from the services themselves. As a consequence, it is possible to adjust aspects of the interoperation at a later time,

without re-programming the services, and to incorporate services with different, or changed, interfaces, by manipulating the assembly only. We see this as a way of easing interoperability in pervasive computing systems.

In the paper, the current language of assembly descriptors has been presented, exemplified by scenarios from the PalCom project, and possibilities for future development and experimentation have been discussed.

References

[1] T. Ekman, G. Hedin, and E. Magnusson. JastAdd: an open-source Java-based compiler compiler system. `http://jastadd.cs.lth.se`.

[2] Nickolas Kavantzas et al. *Web Services Choreography Description Language Version 1.0*. W3C, November 2005. `http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/`.

[3] T. Kindberg et al. People, Places, Things: Web Presence for the Real World. In *Proc. 3rd IEEE Workshop Mobile Computing Systems and Applications (WMCSA 00)*, pages 19–28, 2000.

[4] Obje Interoperability Framework, 2003. `http://www.parc.com/research/projects/obje/Obje_Whitepaper.pdf`.

[5] PalCom. Palpable Computing: A new perspective on Ambient Computing. `http://www.ist-palcom.org/palcom-info.pdf`.

[6] Peter Rigole, Chris Vandervelpen, Kris Luyten, Yves Vandewoude, Karin Coninx, and Yolande Berbers. A Component-Based Infrastructure for Pervasive User Interaction. In *International Workshop on Software Techniques for Embedded and Pervasive Systems STEPS'2005*, Munich, Germany, May 2005.

[7] David Svensson, Boris Magnusson, and Görel Hedin. Composing ad-hoc applications on ad-hoc networks using MUI. In *Proceedings of Net.ObjectDays 2005, 6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World*, pages 153–164, Erfurt, Germany, September 2005.

[8] UPnP™ Forum. UPnP™ Standards. `http://www.upnp.org/standardizeddcps/`.

[9] UPnP™ Forum. UPnP™ Device Architecture 1.0. Technical report, December 2003. Version 1.0.1.

[10] Bill Venners. *The ServiceUI API Specification, Version 1.1a*, 2005. `http://www.artima.com/jini/serviceui/Spec.html`.

[11] Jim Waldo. The Jini Architecture for Network-Centric Computing. *Communications of the ACM*, pages 76–82, July 1999.