Assemblies of Pervasive Services

Svensson Fors, David

2009

Assemblies of Pervasive Services

David Svensson Fors

Doctoral Dissertation, 2009

Department of Computer Science
Lund University

ISBN 978-91-976939-1-2
ISSN 1404-1219
Dissertation 31, 2009
LU-CS-DISS:2009-1

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, 2009

Abstract

Pervasive computing is a vision about computers blending into the background, being there to assist us when we need them, but not requiring constant attention. The vision covers scenarios in the home, at work, and out in the street, and builds on the ongoing development towards an increasing number of embedded computers with network connectivity.

The thesis presents the *assembly* as a lightweight mechanism for combination of devices and services in pervasive computing environments. The assembly is intended to be modifiable by an end user, and to facilitate ad-hoc combinations of services as well as adaptations to changes in services. It does so by separating *configuration* and *coordination*, specified in an assembly descriptor, from *computation*, specified in the services. It supports end-user understanding by using service descriptions that can be inspected and interacted with directly through rendered user interfaces. This gives more flexibility than approaches based on domain-level standardization. An assembly can give rise to services of its own, referred to as *synthesized services*. The synthesized services can be used by other assemblies in turn, for dealing with complex systems in a hierarchical way.

Assemblies and services are elements of the open architecture developed in the project PalCom, and are supported by its communication and discovery protocols. The protocols target resource-constrained devices and situations with varying network connectivity, as required in several of the scenarios studied in PalCom, and they presume no central infrastructure. A central mechanism is the Pacemaker Protocol, which lets devices become aware of each other, using a heartbeat frequency that can be controlled at the application level.

A language for assembly descriptors has been defined, tools have been developed, and frameworks and middleware have been implemented for the developer of PalCom services. These have been used by PalCom partners when building prototypes for scenarios, studied in cooperation with prospective end users in the fields of emergency response, landscape architecture, neonatology, and physical-functional and cognitive rehabilitation.

Sammanfattning

Avhandlingen presenterar resultat från det europeiska forskningsprojektet PalCom inom *pervasive computing*. Detta område handlar om hur små inbyggda, uppkopplade datorer blir allt fler i arbets- och vardagsliv och hur man ska dra nytta av och hantera det.

I PalCom är *assemblyn* ett centralt begrepp. En assembly används för att kombinera tjänster som erbjuds av ett antal enheter. Ett exempel som vi arbetat med i projektet kallas för *GeoTagger*. I detta scenario är en landskapsarkitekt ute i fält för att dokumentera inför planeringen av ett nytt vindkraftverk. Med sig i ryggsäcken har han en kamera, en handdator, en GPS och en kompass. Landskapsarkitekten vill märka varje bild han tar med aktuell koordinat och kompassriktning, vilket underlättar när bilderna senare ska sorteras. Var för sig erbjuder enheterna den funktionalitet han behöver: kameran kan ta bilder, GPS:en ger koordinater och kompassen ger riktningar. Men inte för någon av dem har tillverkaren förberett för den kombinerade funktionaliteten.

Eftersom enheterna erbjuder PalCom-tjänster är det dock möjligt för landskapsarkitekten att bygga en assembly som kombinerar tjänsterna, utan att behöva programmera om enheterna eller själv skriva program i t ex C eller Java. Han kan experimentera med tjänsterna i en PalCom-browser på handdatorn, och också bygga assemblyn med sammankopplingslogiken inifrån browsern. Assemblyn kör sedan på handdatorn, och bilderna märks med koordinater och riktningar. Om han senare byter ut någon av enheterna, eller kanske uppdaterar kamerans programvara till en nyare version, så är det möjligt att göra anpassningar i assemblyn om det behövs.

I PalCom-projektet har vi tagit fram en mjukvaruarkitektur som beskriver hur enheter ska kunna annonsera sina tjänster i ett nätverk, och kopplas samman i en assembly. Vi har också byggt den grundläggande programvara som behövs för detta, programvara som sedan använts av olika grupper i projektet för att bygga prototyper. Prototyperna har testats i scenarier tillsammans med representanter för de tänkta användarna. Dessa kommer bl a från sjukhuset i Siena, Italien och från brandkåren, polisen och sjukhuset i Århus, Danmark.

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 presented in the thesis has been carried out at the Department of Computer Science, Lund University, and within the PalCom project.[1] The work has also been supported by VINNOVA.[2]

Thanks to Sven Gestegård Robertz, who has implemented the PalCom assembly manager and the Developer's Browser, both of which are very important for the work, and to Thomas Forsström, who has implemented the more sophisticated parts of the PalCom protocols, including routing. Thanks to Torbjörn Ekman and Emma Nilsson-Nyman for help with JastAdd and the PalCom Java compiler for the Pal-VM. Torbjörn Eklund has my gratitude for our cooperation during the first parts of the MUI project. Thanks to our PalCom master's thesis students, who have all given valuable input to our research: Boel Mattsson, Brice Jaglin, Thomas Forsström, David Raimosson, Johan Kristell, and Jörgen Ellberg.

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 has been spread across six countries. It has been great both to be able to build on work done in the project, for example when using the Pal-VM, and that the framework we have built has been used by other PalCom teams. For joint paper writing, I especially thank Aino Vonge Corry, Klaus Marius Hansen, Jeppe Brønsted, and Erik Grönvall.

Thanks to the people at the department for good company and interesting lunch room conversations. 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-

[1] PalCom, IST-002057. Palpable Computing: a new perspective on Ambient Computing. In the Future and Emerging Technologies activity of the Information Society Technologies priority, within the European Commission's Sixth Framework Programme.

[2] VINNOVA projects Migrating User Interfaces, 2002-00935, and Mechanisms for Integrating Computer Support in Health Care, 2005-02498.

Marie Westerberg, Lena Ohlsson, Anna Nilsson, Mikael Antic, Jonas Wisbrant, Tomas Richter, Peter Möller and Lars Nilsson have been very helpful with practical things.

Thanks to my football friends in Stora Harrie IF and Staffanstorps GIF, for great fun both on and off the pitch. My deepest thanks to my family for always supporting me, and to my wife Emma Fors for your love and support, and for your way of looking at small and big things in life.

Contents

Chapter 1

Introduction

The vision of *ubiquitous computing* was introduced in 1991 by Mark Weiser at Xerox Palo Alto Research Center [123]. 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 [98] or computers in furniture [55], but they may also be handheld devices, such as handheld computers 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.

During the past decades there has been rapid progress towards the vision of ubiquitous computing, and the vision has guided a lot of research. From a technological perspective, this progress builds on 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 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 be-

ing in the background, everywhere around us. In addition, the concept of *ambient intelligence* has been introduced. That denotes a vision of ubiquitous computers acting more autonomously, making intelligent decisions and thereby providing enhanced user experiences [2].

1.1 Non-preplanned interaction and ad-hoc combinations

Some of the central issues in ubiquitous computing have to do with *interoperability* between devices and services: that they can work together in a way that fits with the expectations of the user. In this thesis, we formulate the following two properties of ubiquitous computing systems, capturing what we see as essential for the vision to succeed:

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 services into new applications, in order to provide functionality that is not given by any of the individual services themselves. It should not be necessary for the services to be designed together in order to combine them.

Providing support for *non-preplanned interaction* and *ad-hoc combinations* has been the overall goal and motivation of the work presented in this thesis. The two properties are particularly valuable for ubiquitous computing systems, compared to systems in a traditional setting. This has to do with the scale of the systems. As noted in, e.g., [33], ubiquitous computing systems can be expected to offer many more services than what is 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. Ubiquitous computing systems are, at least partly, *localized*: they are limited to a geographic location where particular devices are available. As people move, they will encounter new services. Therefore, 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 directly, instead of having to wait for a dedicated service being programmed with the wanted combined functionality.

The rest of this introduction is organized as follows. In the next section, traditional approaches to interoperability will be discussed. After that come a discussion of the method of work used in the thesis, and a brief presentation of the PalCom project, in which we have worked. The architecture we

have developed, and its implementation, will be briefly summarized, followed by a discussion of previous work in the area. The introduction will be concluded with a summary of the main thesis contributions, an outline of the rest of the thesis, and a list of included papers.

1.2 Traditional approaches: standardization

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 [121], 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, when writing 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 from a client program, 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 [56].

In UPnP [114], 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 [113]. The standards specify device and service descriptions, which all follow the same XML Schema. Still, we have noticed that different syntactical conventions are used for different device types: e.g., different characters are used for separating items in lists of values. In practice, this means that interaction with UPnP devices always has to be done after studying the written specification, and cannot be done at a more generic, meta level.

For Bluetooth [12], 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 [11].

Zeroconf [99], the technology for zero-configuration networking developed by Apple, has chosen a different path. For Zeroconf, there are no standard

committees, but a very informal process where anyone can add a service protocol to the list of registered protocols [28]. There are currently hundreds of protocols on the list. Those protocols do, however, have nothing or very little in common, because Zeroconf does not specify anything regarding the format of communication with services. Therefore, the situation for Zeroconf can be seen as similar to the other ones, just with a larger number of standards.

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, as has also been noted, e.g., in [81]. 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.

1.3 Method of work

As indicated above, 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 work has been carried out in an iterative fashion, with an emphasis on experimentation and demonstration of the techniques in implemented systems. With this focus, we have developed communication protocols, middleware, tools and programming frameworks, and demonstrated them in prototype scenarios.

The initial part of the work was done in a project called MUI, with funding from VINNOVA, the Swedish Agency for Innovation Systems [118]. The name MUI stands for Migrating User Interfaces, which reflects our initial focus on the migration of user interfaces between devices. MUI started with a master's thesis project [35], where a prototype with a video cassette recorder (VCR) was built and demonstrated in a user scenario. A user-interface description could be migrated from the VCR to a handheld computer via Bluetooth, enabling the handheld computer to be used as a remote control for the VCR. MUI was a generic framework, and the paper [109] presented MUI's discovery protocol, and XML-based languages for service and user-interface descriptions.

Since then, the scope has broadened to also cover ad-hoc combinations of devices and services, and the main part of the work has been done within the PalCom project, as described in the following section.

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

Table 1.1: *The palpable challenges.*

1.4 PalCom and palpable computing

PalCom is a project in the area of ubiquitous computing. It was funded by the European Commission for four years, 2004–2007, within the Information Society Technologies priority in the Sixth Framework Programme [23]. PalCom introduces *palpable computing*, a new flavour of ambient computing. Through this notion, the project seeks to make ambient computing systems more understandable for humans.

The essence of palpable computing can be illustrated by the *palpable challenges*, as listed in Table 1.1. 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 de-construction of systems at appropriate levels.

The two main objectives of the PalCom project have been 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.

1.4.1 Broad competencies in the project

The PalCom objectives span a wide area, across several disciplines. Consequently, there have been people involved from computer science, interaction design, industrial design, ethnography, and sociology. These researchers, more than 100 people in total, come from eleven academic and industrial partners in six European countries.

The main part of the work presented in the thesis has been done within the PalCom project. Our work has been on the technically-oriented side, on development of the open architecture, with a focus on non-preplanned interaction and ad-hoc combinations, and on work with its reference implemen-

tation. The problems targeted by the thesis do not cover the whole concept of palpable computing, and on the technical side there have been several other results in PalCom. Important examples are the Pal-VM virtual machine and the concept of H-Graphs, as discussed below in sections 1.6.10 and 1.6.11. Section 1.6 presents the technology developed in PalCom, from the perspective of this thesis.

1.4.2 Explorative work

The work in PalCom has been highly explorative. It has been carried out both as software design and development, and as construction of physical prototypes. The prototypes have been used for evaluating and giving input to the open architecture and the conceptual framework. They have been tested out in the field, in cooperation with people representing the anticipated end users. In these prototyping activities, there have been subprojects working on 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. Section 1.5 will present three of the scenarios in more detail.

1.4.3 Traveling Architects

One of the activities in PalCom, intended to help the geographically distributed groups from different backgrounds working together, was the so called Traveling Architects. As part of this activity, the author was involved as a member of a team of software architects that visited the different sites where prototypes were developed. Ideas and architectural input from the prototypes were transferred to the PalCom open architecture, and guidelines from architecture work was spread to prototype developers. Experiences from the Traveling Architects activity have been reported in the paper [24].

1.5 Scenarios

In this section, we will give brief presentations of three of the scenarios in the PalCom project: GeoTagger, Active Surfaces and the Tall Ships' Race. For the work reported in the thesis, the scenarios both serve as inspiration, and give concrete requirements for middleware and protocols developed. They illustrate what kinds of situations the PalCom technology is intended for.

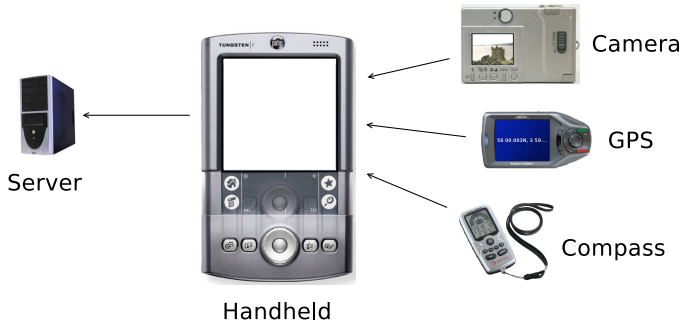


Figure 1.1: *Devices in the GeoTagger scenario.*

1.5.1 GeoTagger

The GeoTagger scenario takes place out in the field, where a landscape architect works with *visual assessment*: the environment of a planned building project is documented, with the purpose of analyzing how it would impact the scenery from different points in the surroundings. In this activity, we have worked with landscape architects in Scotland, who assess the impact of planned wind mills. The landscape architect takes a large number photos, and would benefit from an automatic way of tagging each photo with the geographical location and direction of the camera, so it is easier to sort the photos when coming back to the office.

Out in the field, the landscape architect has a camera, a GPS, a compass and a handheld computer. These are off-the-shelf devices, and none of them have been prepared from the beginning for the functionality of tagging pictures with coordinates and direction. What the landscape architect wants is to combine the devices, so each time a photo is taken with the camera, it is automatically tagged, and uploaded to a server back at the office. The set-up is illustrated in Figure 1.1.

Combining the devices in this way is possible, because each device has PalCom services, and the services can be combined into a PalCom *assembly* (see Section 1.6 for further explanation about how services and assemblies work). The landscape architect creates the assembly in a browser application on a handheld computer, which connects to the other devices in a wireless network. He incorporates a third-party component that performs the actual tagging, by writing the coordinate and direction as meta-data in the JPEG data of the photo. When activated, the assembly saves the current coordinate and direction, each time they are delivered from the GPS and the compass. When the landscape architect takes a picture with the camera, the assembly fetches the picture, tags it, and sends it over GPRS to a storage service at the office.

Figure 1.2: *The Active Surfaces scenario.*

1.5.2 Active Surfaces

Active Surfaces is a PalCom prototype that has been developed in cooperation with therapists and doctors at the 'Le Scotte' hospital in Siena, Italy. It is used for physical-functional and cognitive rehabilitation of disabled children in a swimming pool setting. Exercises are performed in the water, which is interesting from a therapeutic perspective, because many of the children feel safer in the water and can move more freely. It is also interesting from a digital-communications perspective: infrared (IR) communication is used between the devices, partly because radio communication does not work well in water.

The prototype consists of a set of floating tiles (Figure 1.2) that can be connected to each other to form an IR network. The tiles support multiple games, by having a simple composable physical appearance and multipurpose programmable hardware. On each of the tiles' four sides magnets are placed, to make the tiles snap together when they are in close vicinity. On the top of the tile is a replaceable plastic cover, also held in place by magnets. The image on the cover depends on the game. On each side of the tiles light emitting diodes (LEDs) provide visual feedback to the user. Inside the tile, there is a UNC20, an ARM7-based embedded system running uClinux [111].

In one of the games, the child practices reflexes, speed and coordination by placing his or her tile next to one that lights up, as quickly as possible. Other games are different kinds of puzzles, where tiles should be placed in a pattern, forming a solution. The games are configured by the therapist, both before and during the exercise with the child. She needs to be able to adapt the game, depending on how the exercise goes. The configuration is done using a special tile, called the *assembler tile*, that has a button and a display. When preparing a puzzle exercise, the therapist arranges the tiles in the solution pattern, places the assembler tile next to the other tiles, and

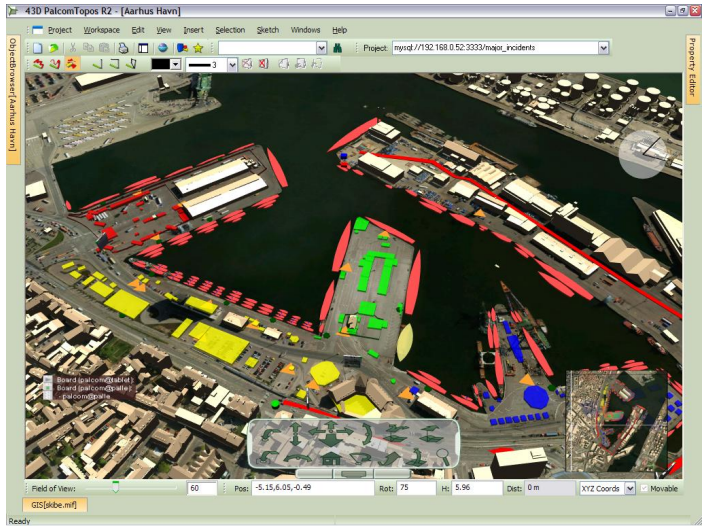


Figure 1.3: *A screenshot from the Topos application used in the Tall Ships' Race scenario. The Overview Assembly prototype was developed at the University of Aarhus.*

presses the button. Information about the correct solution is distributed to all the tiles. When the child plays around with the tiles, LEDs light up when two tiles are placed correctly, giving feedback in the exercise. When the complete puzzle is solved, all LEDs start blinking.

The functionality of the games is not pre-programmed into the tiles. Instead, the therapist configures the tiles for a new game by loading a new assembly onto them. The assembly connects to services on the tiles, that expose the hardware functionality to other devices on the network. In addition to the advantage of better flexibility, the possibility to load new assemblies dynamically makes it possible to use PalCom mechanisms for inspecting the tiles, and finding out what happens if something breaks down or behaves unexpectedly [14].

1.5.3 Tall Ships' Race

The GeoTagger and Active Surfaces scenarios both contain assemblies used by one or a few people at a time. An example of a bigger, more complex PalCom scenario is the use of the PalCom Overview Assembly prototype at the Tall Ships' Race in Aarhus, Denmark, 5th–8th of July 2007 [83]. Tall Ships' Race was a big event, where over 100 sailing ships took part in a competition, and about 200 000 people gathered in Aarhus harbour each of the four days.

The Overview Assembly prototype is illustrated in Figure 1.3. It is a tool for emergency response personnel, such as police and fire brigade officers, medics and paramedics, at the site of a major event or a major incident. Tall Ships' Race is an example of a major event. A major incident could be, e.g., a train crash. The tool helps to give overview and coordination between personnel of different professions working at the site.

The prototype consists of PalCom assemblies that collect and display information about what takes place at the site. At the Tall Ships' Race, there was live tracking of ships and key personnel, live streaming of still pictures from mobile phones, and live streaming of video data from cameras mounted in strategic locations. The data from the assemblies was presented using the application Topos from 43D [1], that visualized the data in a 3D model of the area.

1.6 Architecture and implementation

The scenarios in PalCom, the three described in the previous section and others, pinpoint different aspects of the PalCom challenges, and of non-preplanned interaction and ad-hoc combinations. Requirements have been drawn from the scenarios. A set of communication protocols, middleware components and tools have been developed, as part of PalCom, in order to demonstrate support for those requirements, and a reference implementation of the PalCom open architecture has been made in the project. This section will give a brief overview of concepts in the implementation, and their place in the architecture.

1.6.1 Devices

The *device* is a central concept in PalCom. Hardware devices provide an execution platform for the services, and for middleware components. Furthermore, since the architecture is made for ubiquitous computing systems, exposing some aspect of the hardware functionality is often the main purpose of a PalCom service. Such examples are the photo service of the camera in the GeoTagger scenario, and a service exposing LED functionality on an Active Surfaces tile. Therefore, devices have an explicit place in the architecture, not only as an execution platform for the services, but also in their own right.

1.6.2 Services

The *service* is the fundamental interaction point and building block in PalCom systems. Figure 1.4 shows the symbol of a service on a device, using

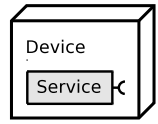


Figure 1.4: *The symbol of a service on a device.*

the notation adopted in this thesis. The small hook on the service box indicates that the service can be connected to.

A service announces itself on the network in a *service description*, which is transferred in an XML representation. The service description describes the functionality of the service: it lists what *commands* the service can send or receive, and their parameters. Users can interact directly with services through user interfaces that are rendered from the service descriptions. The service description makes the service *self-describing*, and users can *inspect* the service descriptions when interacting directly with the services. Service descriptions are also useful for the second main use of PalCom services, namely to combine them into assemblies, as discussed below.

The most important aspect of a service is the interface that it exposes on the network; that it can be announced and interacted with as a PalCom service. How the service is implemented is orthogonal to that: the service can be written in any programming language, as long as it behaves as a service on the network. For example, PalCom services have been implemented in Java using the Service Framework in the PalCom reference implementation, and also in C as Linux daemons on an Axis network camera, as demonstrated in the master's thesis project [72]. In that project, the Axis camera offered PalCom photo services, and was used in a version of the GeoTagger scenario.

1.6.3 Discovery

In order to announce and use services, PalCom devices follow communication and discovery protocols that have been developed and defined in the project. When discovering devices and services, it is important that the user is not overwhelmed by the sheer amount of discovered entities. The devices and services shown to the user have to be those that are relevant for him or her to use. The approach taken by PalCom is to limit discovery to the local network: in particular, devices and services are not automatically discovered across the entire Internet. This fits with the picture that you interact with physical devices that you can see and touch, and with the requirements drawn from scenarios. Tunnels, as discussed below, are used for extending PalCom communication beyond the local network.

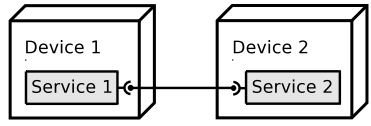


Figure 1.5: *A connection between two services.*

1.6.4 Connections

In addition to devices and services, a third element in the architecture is also made visible to the user as a result of discovery: *connections*. The connections are paths of communication between PalCom services, or between a service and an assembly. Figure 1.5 shows the graphical notation for a connection between two services. Connections are announced explicitly during discovery, in order to make the communication situation in a PalCom system visible to the user, and as support for the execution of assemblies.

1.6.5 Browsers

PalCom *browsers* are tools used for discovering and interacting with PalCom services. The browsers are applications running on the user's device, performing discovery and displaying nearby devices and services. In the browser, the user can interact with services through user interfaces that are generated from the service descriptions. These user interfaces can be generated without the browser having to be prepared for the exact contents of the description—for what particular commands that service has. It suffices that the browser can interpret the general format of service descriptions and commands. This generic, domain-independent rendering of a service

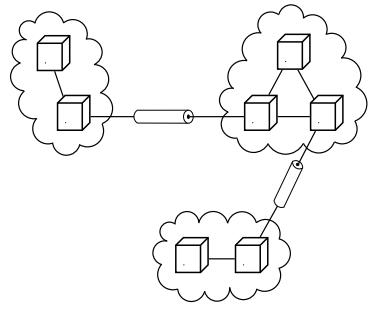


Figure 1.6: *Three local PalCom networks connected by tunnels.*

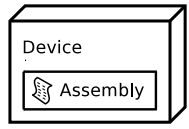


Figure 1.7: *The symbol for an assembly.*

description is part of what makes *non-preplanned interaction* possible. As tools for interaction with PalCom systems, browsers also have other functionality, most importantly as assembly editors, and as assembly managers that execute assemblies (see Section 1.6.7).

1.6.6 Tunnels

The way we have chosen for extending PalCom networks beyond the sometimes too limited local network is PalCom *tunnels*. Tunnels, as designed and implemented in the master's thesis project [38], are manually-administered, secure connections between local PalCom networks that may be geographically distant. Devices and services can be discovered through the tunnel, and thereby the two PalCom networks are combined into one. The model, when working in a scenario where devices are geographically distributed, is that small local networks are tied together by tunnels, as illustrated in Figure 1.6. In the figure, the local networks are shown as clouds, and the tunnels as tubes. A few devices communicate in each network. For the required scalability of PalCom systems and communication protocols, this means that we do not target the discovery mechanism towards networks with huge numbers of devices, such as the Internet, but smaller networks with more limited numbers of devices.

1.6.7 Assemblies

The *assembly* is the element in the PalCom architecture that enables combinations of services. Assemblies can be defined by users or developers, and may contain an *assembly script* which adds to the combined behaviour of the services. Figure 1.7 shows the symbol for an assembly.

The assembly is intended to be used with services available in a particular, local situation. When creating a new assembly, or adapting one that has been created for similar services on other devices, the user can try the services and the assembly out in the interactive assembly editor in the browser. This possibility to explore available services and try things out is important for being able to create useful ad-hoc combinations.

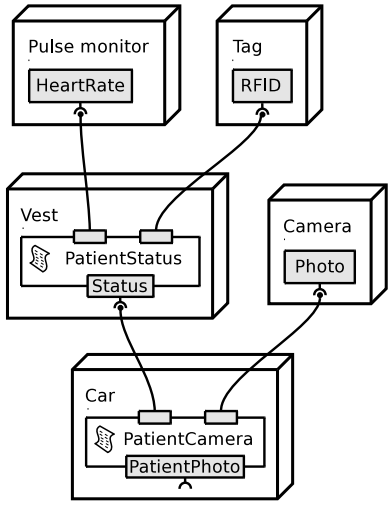


Figure 1.8: *A hierarchy of PalCom assemblies in a major incident scenario.*

The basic contents of an assembly is the following:

- A list of **devices** included in the assembly
- A list of **services** on those devices
- A list of **connections** between services, and between services and the assembly

A simple assembly, containing only these parts, facilitates repeated use of a set of services. When the assembly is executed, it monitors the network and establishes the listed connections, unless they are already established.

Building on simple assemblies, it is possible to add coordinating logic to the assembly in an assembly script. The assembly script contains one or more *event handlers*, specifying how commands coming to the assembly should be handled, and forwarded to other services. In the script, it is also possible to keep state in variables. An example of this is in the GeoTagger assembly, where the latest coordinate from the GPS is saved, every time it arrives.

Synthesized services

It is sometimes useful to let the assembly provide one or more services, which are referred to as *synthesized services*. Those are services that offer some combined functionality of the included services to other devices in

the network, and they can capture aspects that are not covered by the individual services themselves. In Figure 1.8, which illustrates assemblies used in a major incident scenario, the assembly *PatientStatus* has a synthesized service *Status*, as indicated by a service symbol with a hook on the border of the assembly symbol. PatientStatus combines the services *HeartRate* and *RFID* on the pulse monitor and tag devices.

Like other services, the synthesized services can be interacted with directly in the browser, where user interfaces can be rendered for them. It is also possible to use synthesized services from other assemblies, building a hierarchy of assemblies. Such a hierarchy is illustrated in Figure 1.8. The assembly PatientStatus uses HeartRate and RFID, and provides the synthesized service Status. The assembly *PatientCamera* uses Status and *Photo*, and itself provides a synthesized service *PatientPhoto*. PatientCamera combines patient status information with photos, for use by emergency response personnel at the incident site.

Unbound services

It is possible to incorporate services written in a general-purpose programming language, such as Java, into an assembly, for including more advanced functionality than what can be specified directly in the assembly script. We refer to such services as *unbound services*. The name reflects that they are not tied to the hardware of any particular device, but built for performing computations in assemblies. An example of an unbound service is the service that tags JPEG images with GPS coordinates in the GeoTagger assembly.

One of the ideas behind unbound services is to keep the assembly script language simple. The script language should be possible to work with for an end user. Therefore, we do not want to add features to the language that allow computations as complex as, e.g., dealing with JPEG image metadata. Such computations are left to unbound services.

Bindings

The assembly lists a set of connections between services and assemblies. For handling dynamic cases, where an assembly functions with a varying set of available services, there is support in the assembly description language for different *bindings*. The first type of binding defines whether the presence of a service is *mandatory* or *optional* for the assembly to function as intended. The second type of binding is *alternatives*, which means that one of a list of services should be used, in a prioritized order. Alternative bindings are used, e.g., in scenarios with varying network connectivity, where services can act as back-ups for others. Optional bindings can be used in scenarios where all currently available services in a set of services are rele-

vant to use—they can all be defined as optional—or simply where a service is not necessary for the core functionality of the assembly.

Assembly managers

At runtime, an assembly executes in an *assembly manager*. The assembly manager can be integrated in a browser, or it can run by itself on a device on the network. The assembly manager's task is to handle the execution of assemblies, by monitoring the network and establishing connections that are possible but not already established. The manager interprets assembly scripts, and executes event handlers when messages arrive at the assembly. It runs on one device, but if assembly managers are available on other devices, the assembly can be moved to another assembly manager in order to optimize use of network bandwidth, as discussed in Chapter 4.

1.6.8 Versioning

The possibility for users to make adjustments to assemblies, adapting them to local needs, in itself means that there can be many very similar assemblies, solving similar problems. As a means of making it possible to keep up the order in this situation, a versioning scheme has been designed for assemblies and services. An assembly lists particular versions of its included services, and it is tested with those before it is released. A fundamental property of the versioning scheme is that if the assembly is changed to refer to a newer version of one of its services, this means a new version of the entire assembly is created.

Assemblies can be migrated to assembly managers on other devices. After that, updates to the assembly can happen independently on the different devices. Version numbers are created in a way that makes it possible to re-create the version tree of an assembly when receiving one that might already exist in several versions on the receiving device. This versioning scheme is handled at the level of the PalCom communication protocols.

1.6.9 Communication protocols

For making it possible for PalCom devices, services, assemblies and tools to discover each other and communicate, there has to be common protocols governing the basic communication. Such protocols have been defined in the project. At the bottom level, the so called *Wire Protocol* defines how messages are packaged on top of different underlying network technologies. The architecture supports routing across network technologies such as UDP, Bluetooth or infrared, and the Wire Protocol defines basic message formats that are independent of the technology used. It also defines how

routing information is put into the messages. On top of the Wire Protocol, the *Discovery Protocol* specifies how PalCom devices and services can announce and discover each other and their descriptions. This protocol is based on periodic announcements, with caching of received information for reducing the network traffic. The third protocol is the *Service Interaction Protocol*, which specifies how services communicate once they have been discovered: how commands with parameters are packaged when sent over the network, etc.

Regarding the protocols, it is important to note that these are not protocols defined at the domain level. Instead, the protocols are at a *generic* level: how to make a service known in the network, how to announce its service description, and how to format messages. The service description lists what commands the service can send and receive. The interpretation of the contents and meaning of the commands is left to the constructor of the services and assemblies.

1.6.10 Middleware and framework

As support for developers of PalCom devices and services, the reference implementation of the PalCom open architecture contains middleware components, referred to as *middleware managers*, that perform various tasks that are common for all services on a device. There are managers for announcement and discovery, for execution of assemblies, for loading of unbound services, as well as for the lower-level communication protocols. For the service developer, there is also a *Service Framework* that provides the basics of implementing a PalCom service and interfacing the middleware managers.

In the PalCom open architecture, and in the reference implementation and the framework, there is also support for the concept of *H-Graphs*, which is a technique for allowing devices and services to be inspected by organizing their data in externally accessible tree-like graphs [86]. The author has not been directly involved in the development of the H-Graphs concept nor its implementation, and H-Graphs are not covered in this thesis.

1.6.11 Hardware platforms

PalCom software, as implemented in the reference implementation, runs on two different virtual machines, which in turn run on several hardware platforms. The virtual machines are the *Pal-VM*, a virtual machine developed in the PalCom project, and the JVM, Sun's Java Virtual Machine [70]. The Pal-VM is targeted for resource-constrained devices, and has support for interfacing between classes written in different languages, such as Java and Smalltalk [86]. The Pal-VM has been ported to desktop versions for

Mac OS X, Windows and Linux, and also to UNC20, the ARM7-based embedded system running in the Active Surfaces tiles.

The implementation of the Pal-VM is one of the big efforts in PalCom. The work presented in this thesis has consisted of building software running on the Pal-VM—as a user of the Pal-VM and the special Java compiler developed for it in the project—but not of work on the Pal-VM or compiler implementations themselves.

1.6.12 An open source implementation

The source code for the reference implementation of the PalCom open architecture, and for a number of example devices and services, has been published as open source under a BSD license. The code, and pre-built binaries, are available for download from the PalCom web site [88].

1.7 Scalability and security

Scalability is important for ubiquitous computing systems. In the work on the PalCom communication protocols, efficiency and performance have been taken into account, in addition to the functional requirements drawn from scenarios. 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, a large number 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.

As mentioned above, 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 Chapter 5, we have used a very light-weight heartbeat mechanism, combined with caching, 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.

Like in all ubiquitous computing systems, the qualities of security and privacy are also important in PalCom. These have not been our focus, though,

and we rely on mechanisms in lower layers for keeping data secure and private. In ubiquitous computing literature, the physical boundaries of systems are mentioned as helpful for keeping security in ubiquitous systems [62]. Social models for security have also been suggested [57].

1.8 Previous work

The area of ubiquitous computing started with the work of Mark Weiser and others at Xerox PARC in the late 1980s [122]. The ideas were based on the observation that successful technologies tend to disappear into the background. One example, mentioned in Weiser's 1991 paper [123], is electric motors. There are more motors in a modern car than the driver can meaningfully keep track of—he does not need to—while a hundred years ago in a factory there was typically a single engine that drove dozens or hundreds of different machines. Weiser speaks of *calm computing*, meaning computing systems that disappear from the attention of the user when they work as intended.

At Xerox PARC, the ideas were evaluated and elaborated by developing a ubiquitous computing infrastructure, that was used in projects and in day-to-day work. The technology developed included a large, wall-sized display called the LiveBoard, the book-sized computer ParcPad, and the palm-sized computer ParcTab. Weiser envisioned hundreds of computers per room, most of them, of course, on the smallest scale.

In later projects, the ubiquitous computing research community has continued to work towards the vision of a potential third generation of computing systems, with many computers per user, after the mainframe (many users per computer) and the personal computer (one computer per user) [2]. There are many challenges, both on the technical and the sociological levels. Tiny sensors that cooperate in networks need to function with minimal energy consumption. Redundancy may be needed for achieving precision in sensor values. More powerful devices may need to handle demanding computations on behalf of others [37]. User interaction becomes quite different than with screen-mouse-keyboard. E.g., implicit commands based on the user's physical actions in a certain context have been suggested [3]. Taking ubiquitous computing from the research laboratory to large-scale use in real environments leads to issues about robustness, software integration, privacy and service payment [25].

1.8.1 Ubiquitous computing architectures

Like PalCom, several research projects have built architectures for ubiquitous computing, and evaluated them in labs or in contact with end users. This section will give a brief walkthrough of some of those projects.

Interactive Workspaces

Interactive Workspaces at Stanford [57] focuses on specially equipped meeting rooms, iRooms, and how ubiquitous computing technology can be used in that setting. The rooms have large, interactive touch screens that can be used cooperatively by meeting participants. The software infrastructure consists of the *meta-operating system* iROS, that has support for moving data between applications on screens and on handheld devices, for remote control of devices and applications by any user in the room, and for coordination between applications. The iROS subsystem ICrafter [90] generates user interfaces for services, and the Event Heap subsystem is used for coordination. The Event Heap is based on a *tuple space* model, for achieving loose coupling (as used in the Linda language [16]). One general commonality between Interactive Workspaces and PalCom is the focus on letting the user adjust the environment, as opposed to having the environment react autonomously and intelligently to user behaviour.

One.world

The project One.world [43, 44] aims to give system support for building pervasive applications. The main overall goals are (1) to *embrace contextual change*, i.e., to make applications aware that location and execution context change as people move around, (2) to *encourage ad-hoc composition* by making devices and applications possible to just plug together, and (3) to *facilitate sharing* of information between applications and devices. The One.world system has been tested in a number of experimental applications, among them a digital biology laboratory. Corner stones in the architecture are an *environment* mechanism, which makes it possible to group data and running applications, and migration of code and data using the environments. Also One.world uses a tuple space model, for loose coupling and data sharing. PalCom has similar goals of ad-hoc composition, but an important difference, compared to PalCom, is that PalCom does not rely on code migration.

Gaia

Gaia [94] is another meta-operating system for active spaces, i.e., for environments such as offices and meeting rooms. The purpose of Gaia is to provide abstractions for the heterogeneous devices in such environments, and make the environments programmable. Gaia is organized at three levels, where the lowest level corresponds to a traditional OS, the middle level is the application level which provides frameworks and tools to build applications, and the upper level is the *active space behaviour-level*, where the active space can be programmed by orchestrating interactions among app-

lications (dynamic application composition). For the orchestration there is a mechanism called *application bridge*, which can be compared to PalCom's assemblies. Gaia has been evaluated through implemented applications such as a speech engine, a slide show manager and a location service, and through implemented bridges between the applications.

Aura and PICO

A project with a slightly different focus is Aura at Carnegie Mellon [40]. Aura is more directed towards *proactivity* and *self-tuning*, where the system adapts to the user's actions in an autonomous way. The ultimate goal is a system that distracts the human user as little as possible. Prism, which is the Aura subsystem that handles proactivity and self-tuning, uses task descriptions for keeping track of users' intentions. One of the example applications keeps track of people's location by analyzing the traffic in a wireless LAN. PICO [66] is another project that aims for an autonomous and proactive system. PICO has agents, referred to as *delegents*, that are proactive and cooperate for solving tasks for users and applications. In an example telemedicine scenario, delegents detect that an accident has happened out in the street, and establish a community that contacts an ambulance community. That community in turn connects to a hospital community for giving fast and accurate information about the accident. With their more AI-like approach, both Aura and PICO differ a lot from PalCom.

Accord

The home has been an important target for ubiquitous computing projects. The Accord project [93] presents an architecture supporting the introduction of new ubiquitous devices and services in the home. The architecture has an assembly mechanism. One goal is to be able to introduce ubiquitous computing in a piecemeal fashion, without having to build a new house from scratch. There is a focus on rapid reconfiguration. Smart-Its components [9] are used for giving the physical things a digital representation. The basis of the assembly mechanism is a shadow digital space that acts as a digital representation of the physical environment. The assembly concept seems to be simpler than PalCom's, but with a jigsaw puzzle-style editor for combining components, the model might be too simple.

InterPlay

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 [75], which is a middleware for

integrating devices in a networked home. 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. We see the creation of such an attribute hierarchy as an obstacle in a more general setting.

1.8.2 Interoperability in ubiquitous computing

While the projects mentioned in the previous section give a context for the PalCom project as a whole, and its open architecture, there are also projects that are more directly related to the *ad-hoc combinations* and *non-preplanned interaction* dealt with in this thesis. In [62], Kindberg et al. mention *spontaneous interoperation* as an important challenge for ubiquitous software systems. By this the authors mean that components should be able to communicate easily without too much human administration, also in environments with frequent changes, where new components often enter and others leave. The authors argue that mobile computing, which is seen as a pre-cursor of ubiquitous computing, has not been able to solve the interoperability problem with many heterogeneous and physically integrated devices. The concept of *the semantic Rubicon* is introduced for the border between semantics handled by the system and by humans, and it is argued that clear mechanisms are needed for how to deal with that border. One example is discovery protocols, which are traditionally completely on the system side, making them blind for human territories and conventions.

Obje

Obje at Xerox PARC [32, 34, 81] is a project that targets very similar problems as we do. They seek to enable interoperability between ubiquitous devices or components with only limited *a priori* knowledge of one another, and without relying on domain-specific standards.[1] Obje introduces the term *recombinant computing* for an architecture where the user can combine functionality from several services into one. There should be easy configuration, use and re-use in an ad-hoc manner.

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

A central part of the Obje approach is 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 functionality similar to that of our migrated user interfaces, where the proxy object corresponds to our service description. A second main point in Obje is that there should be a small set of generic, *trans-domain*, interfaces, without domain-specific details. Applications should be written against the generic interfaces, and the interfaces are used for combining components into new combinations. MIME types are used for determining what connections can be made. A third key point is what Obje refers to as *user-in-the-loop interaction*, which means that the user should always be in control when connecting services, leaving it up to him to make sure the connection makes sense. Users are expected to sort out much of the semantics, by determining when and why components should interact. This, the Obje designers claim, helps keeping the trans-domain interfaces small and generic.

There are similarities between Obje and PalCom in the way of involving the user in the process of sorting out how services can be combined. This is in contrast to systems with more AI-like techniques. The idea of rendering a user interface from a service description is also similar. Still, the focus in Obje and PalCom are partly different. The focus in Obje has been on providing mechanisms for an end user without programming expertise. This is an important aspect of PalCom as well, but in addition we have a focus on building assemblies, using the control part of a remote device as an API. PalCom puts the interoperation logic in the assembly, instead of relying on mobile code. Assemblies in PalCom can offer new services, which can be used in other assemblies in their turn, thus providing a hierarchical composition mechanism. There is no concept in Obje corresponding to the assembly.

Mobile code requires some platform-independent code representation, and Obje has used Java in their implementation. For data communication, such as audio or video, the Obje solution puts the requirement 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. In addition, Obje's trans domain interfaces offer only very generic operations, such as reading a chunk of data. In contrast, PalCom's service descriptions are distributed as XML, which are more lightweight and 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. The textual descriptions used in PalCom allow the output devices to control the rendering. Furthermore, the PalCom solution gives an architectural advantage in that the same inter-

face description can be used both to drive a user interface and to drive a programmatic API.

Ponnekanti et al.

Ponnekanti et al. at Stanford have also spotted the problems associated with the use of standardized domain-specific service interfaces [91]. They have implemented an approach that is more different from ours, than is Obje's. Ponnekanti et al. separate the two problems of (1) sorting out if it is semantically meaningful to interact with a service, and (2) the mechanics of interacting with it. They argue that existing service frameworks such as Jini and UPnP mix the two, and that this makes interoperability severely restricted. The presented approach relies on standardized application-independent invocation mechanics, and on human-understandable natural language descriptions in service advertisements. For the semantics, the system first narrows down the possible targets from all the services in the environment to a small set of related services, and then relies on the user to select the desired service. For the mechanics, the system dynamically constructs a suitable proxy by assembling *stubs* and *adapters*, that are downloaded from directories available in the network. Chaining of adapters is used for avoiding a combinatorial explosion. The adapters can be lossless or lossy, depending on if they support the whole interface of the service they represent.

Ponnekanti et al. distinguish between single-standard and multi-standard assumptions about the world. While Jini and UPnP strive for a single standard for each domain, Ponnekanti et al. want to support a multi-standard situation, with several different interfaces for printers, for search engines, etc. The authors have noted that the single-standard assumption is hard to follow, even for well-understood services, and that market-driven consolidation is slow where that mechanism is in play. A crucial difference from PalCom 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. From the PalCom scenarios, we have concluded that we cannot rely on that kind of central directory.

1.8.3 Technical issues

In addition to the ubiquitous computing architecture level, the work in this thesis has also been carried out at other, more technical levels, primarily in the areas of networked services, service composition and communication protocols. For each of these areas there is much related work, which will be presented in more detail in conjunction with the respective discussions in Chapters 2–6.

Among technologies for networked services, Jini and UPnP are based on domain-level standardization of service interfaces, and Jini is tied to the Java language, which has implications for the runtime system on devices. It is not clear to us if technologies for Web Services [120] can be made as lightweight as is needed for ubiquitous computing environments, and several of those approaches are based on the Semantic Web and on ontologies, which we argue lead to problems similar to those of domain-level standardization.

Our approach for service composition is the assembly, which is targeted at the end user, unlike Gaia's application bridges that are created by developers. The assembly is not focused on autonomous agents, like Aura and Amigo [115], and can be used for more sophisticated compositions than Obje's direct connections between services. We also believe the assembly is more powerful than the producer-consumer patterns used by ICrafter.

At the communication protocol level, there are many dimensions along which to compare with other protocols. UPnP and Zeroconf [99] are based on IP, whereas PalCom supports different underlying technologies by means of an abstraction mechanism. Jini, Zeroconf and Web services focus strictly on services and have no notion of physical devices, which is important in PalCom and in pervasive computing. Some protocols, such as Jini and Salutation [96], base their discovery mechanisms on central directories, which we cannot rely on in PalCom. There are also different approaches to the issue of devices frequently joining and leaving networks. UPnP and PalCom use periodic broadcasts, where in PalCom the broadcasting period is adapted to the needs of the currently present devices. Jini uses a mechanism called *leasing*, while Zeroconf does not detect sudden device disappearances at all, until some service tries to communicate with the device. For the communication with services, Jini's proxy mechanism builds on mobile code, while PalCom is specified as a set of protocols and message formats. Jini, UPnP and Web services use variants of *Remote Procedure Call* (RPC) for communication with services. PalCom has asynchronous communication, with commands sent to services without blocking the sender.

1.9 Contributions

The main contributions of this thesis are architectural ideas and a prototype implementation for *non-preplanned interaction* and *ad-hoc combinations* in pervasive computing environments. In particular, we have made

- designs of the notions of **services and assemblies** in the PalCom open architecture,
- definitions of domain-independent **discovery and communication protocols** that support the services and assemblies,

- a design of a **language for assembly descriptors**, that is interpreted by assembly managers, and
- implementations of **frameworks and middleware** that support the developer of PalCom services.

1.10 Thesis outline

The rest of this thesis is organized as follows:

Chapter 2: Devices and services presents the concepts of device and service in the architecture from a user perspective, starting from two scenarios.

Chapter 3: Connections discusses the concept of connections, which are made explicit in the architecture.

Chapter 4: Assemblies presents assemblies, and the role of the assembly in providing flexible combinations of services, starting from user scenarios.

Chapter 5: Communication protocols presents PalCom communication protocols that have been developed as part of the thesis, supporting services and assemblies.

Chapter 6: The language of assemblies gives a presentation of the assembly descriptor language, and how it supports a separation of the aspects of configuration, coordination and computation.

Chapter 7: Browsers presents PalCom browsers that have been implemented.

Chapter 8: Framework and middleware discusses the implemented Service Framework and middleware from a developer's perspective.

Chapter 9: Implemented scenarios describes PalCom scenarios where the reference implementation has been tested, in cooperation with end users and on resource-constrained platforms.

Chapter 10: Evaluation evaluates the proposed architecture and implementation, looking at non-preplanned interaction and ad-hoc combinations, at developers' use of the framework, and at the scalability of the architecture to large networks and small devices.

Chapter 11: Conclusions and future work concludes the thesis and gives directions for future work.

The work has been part of the big PalCom project, which means that most of it has been done in cooperation with others. The detailed notions of services, connections and assemblies, as presented in Chapters 2–4, have been developed mainly by our group in Lund, but in cooperation with the other PalCom partners. Likewise for the communication protocols presented in Chapter 5, and for the assembly descriptor language in Chapter 6. Of the browsers presented in Chapter 7, I developed the first so called Handheld Browser, while the PalCom Developer's Browser was developed by Sven Gestegård Robertz at Lund University, the PalCom Overview Browser at the University of Aarhus, and the NICU Browser at the University of Siena. In the implementation of the framework and middleware, as presented in Chapter 8, I developed the Service Framework and initial versions of a UDP media manager, a routing manager, a communication manager, an announcement manager, a discovery manager and a service manager. This work has been continued by Thomas Forsström at Lund University. Sven Gestegård Robertz has implemented the assembly manager. I implemented the PalcomThreads thread library, and worked on the interface to the underlying virtual machines, but did not work on the Pal-VM implementation, or the compilers for it. The framework for simulated devices was developed as part of this thesis. The work on PalCom scenarios, from where examples are reported in Chapter 9, was carried out mainly at the PalCom partners outside Lund. Of the examples in Chapter 9, the first GeoTagger example with simulated devices (not the one with Google Earth support), and the experiment with Groupcast tiles (Section 9.4.4) were implemented by me. The others were implemented at the University of Aarhus, at the University of Siena, and as master's thesis projects in Lund, as explained in Chapter 9. The performance measurements, reported in the evaluation in Chapter 10, were done as part of the thesis.

1.11 Publications

The thesis is largely based on material from published papers. An early version of the architecture, with a focus on remote control of services through migrating user interfaces, was published in

> David Svensson[2] and Boris Magnusson. An Architecture for Migrating User Interfaces [109]. In *NWPER'2004, 11th Nordic Workshop on Programming and Software Development Tools and Techniques*, pages 31–44, Turku, Finland, August 2004.

[2] I changed my last name to Svensson Fors in 2007.

Simple assemblies, that coordinate services through a set of saved connections, and an initial version of scripted assemblies were presented in

> David Svensson, Boris Magnusson, and Görel Hedin. Composing ad-hoc applications on ad-hoc networks using MUI [110]. 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.

The assembly descriptor language, and an implementation of the GeoTagger scenario using the language, were described in

> David Svensson, Görel Hedin, and Boris Magnusson. Pervasive applications through scripted assemblies of services [108]. *Pervasive Services, IEEE International Conference on*, pages 301–307, July 2007.

Work on a simulator for the Active Surfaces scenario, that built on the implemented Service Framework and middleware, was published in

> Jeppe Brønsted, Erik Grönvall, and David Fors. Palpability Support Demonstrated [14]. In *Embedded and Ubiquitous Computing*, volume 4808/2007 of *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin/Heidelberg, 2007.

In [14], the author of this thesis contributed mainly with work on the middleware and the framework. There are also other publications, which are not considered part of the thesis:

> Aino Vonge Corry, Klaus Marius Hansen, and David Svensson. Traveling Architects – A New Way of Herding Cats [24]. In *Quality of Software Architectures*, volume 4214/2006 of *Lecture Notes in Computer Science*, pages 111–126. Springer Berlin/Heidelberg, 2006.

> Erik Grönvall, Alessandro Pollini, Alessia Rullo, and David Svensson. Designing game logics for dynamic Active Surfaces [46]. MUIA 2006: third international workshop on mobile and ubiquitous information access. Espoo, Finland, September 2006.

Chapter 2

Devices and services

Interacting with services is the main way to use PalCom systems. The basic services are the ones that are offered on the network by devices in the environment of a user, for example in a room. In PalCom, these services are referred to as *native services*, because they are tied to the hardware. This chapter discusses such services, and how they can be interacted with through user interfaces rendered in a browser. Assemblies, as discussed in Chapter 4, can also offer services, so called *synthesized services*. Those are structured and announced in the same way as the native services presented here.

The most important aspect of a service is how it behaves on the network. In order to work together with other services, it has to follow the PalCom communication protocols, as described in Chapter 5. Chapter 8 discusses how the PalCom reference implementation, and its Service Framework, support a programmer in building services that work this way. The current chapter, Chapter 2, discusses the main features of PalCom services, with a focus on the underlying design choices and taking user scenarios as the starting point.

In the following section, two illustrating scenarios will be presented, where users connect to services and use them. After that follows more detailed descriptions of how PalCom services are structured on a device, how service descriptions are built, and how user interfaces can be rendered from service descriptions. The chapter ends with a discussion about how PalCom services relate to other architectures built around services.

2.1 Scenarios

The two simple scenarios presented in this section illustrate how PalCom services are used. They are not as elaborate as the scenarios that have been

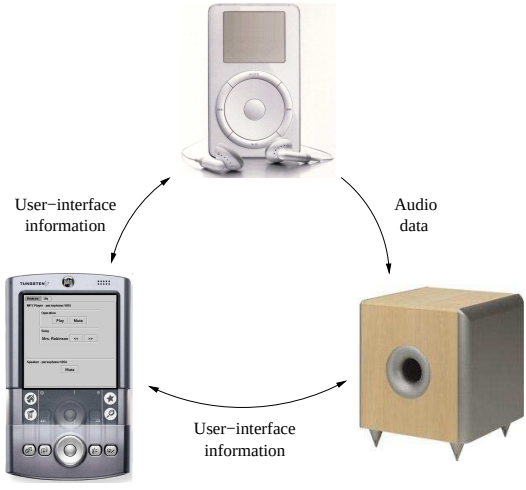


Figure 2.1: *An example scenario with a handheld computer, an MP3 player, and loudspeakers.*

studied in cooperation with end users in PalCom, but serve to give requirements for the technical mechanisms involved when using PalCom services.

2.1.1 A music scenario

Figure 2.1 shows an example scenario where the PalCom architecture is at work:

> *With a handheld computer in his hand, and a portable MP3 player in his pocket, John enters a room where a set of loudspeakers are in the corner. The loudspeakers and the MP3 player are automatically discovered, and they show up in a browser application on the handheld. There are audio services on both the MP3 player and the loudspeakers. John can see that they match, and connects them by joining them in the browser. Now, the MP3 player sends its music to the loudspeakers. The volume is a little low, though, so he chooses to control the loudspeakers by clicking their volume control service in the browser. A user interface is moved to the handheld and shown. It looks as in Figure 2.2(a). John 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 to that of Figure 2.2(b). If John wants to control also the MP3 player from the handheld, a user interface can be obtained for it in the same way.*

(a) Before adjusting the volume.

(b) After adjusting the volume.

Figure 2.2: *A migrated user interface for the loudspeakers.*

2.1.2 A slide show scenario

As another example of a scenario where PalCom 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 with a wire to the projector. When it is time for the next speaker, the user 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 PalCom 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 than the traditional setup 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. Second, 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 slide show 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 2.3 shows a setup for this scenario. The arrows show how commands and images flow between services. The devices in this scenario, projectors, laptops, etc., run PalCom software. The projector has a PalCom service, Screen, that can receive JPEG images and project them onto the physical screen. The laptops have PalCom services, Control, which give user interfaces for controlling a slide show (with buttons Play, Stop, Next, etc.), and a service Slides which can send out slide show JPEG images over network connections. The mobile phone has a PalCom browser, which can discover nearby devices and their services. Through the browser, the user can ask the Screen to connect itself to the Slides service 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 slide show user interface, which

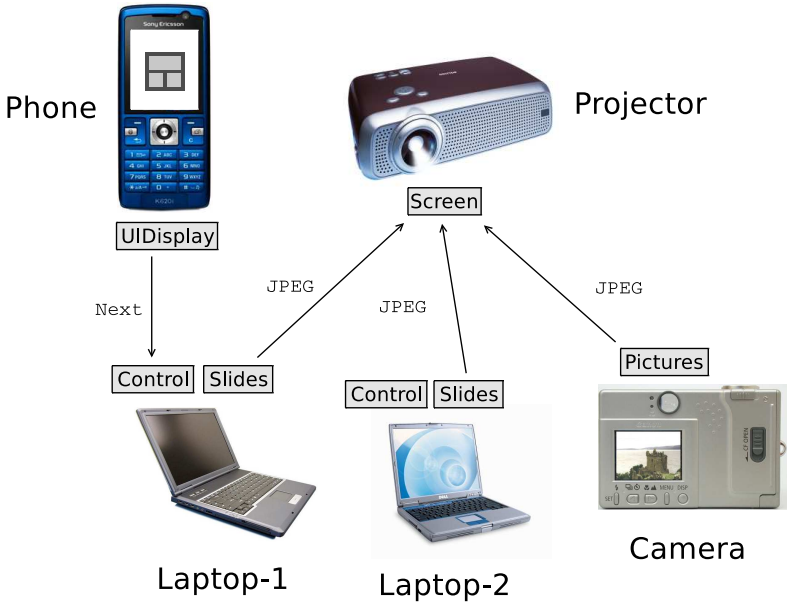


Figure 2.3: *The slide show scenario.*

causes this to migrate from the service Control on the laptop and pop up on the display of the phone. Now, the user can use the phone to change slides during the presentation. Alternatively, the user can set up service connections and issue user-interface commands (Play, Stop, Next, ...) using a PalCom browser on the laptop. 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.

Extending the scenario: adding a camera

PalCom 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 service Pictures 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. The projector and its services have automatically been discovered in the browser on her camera.

2.2 Devices

In the two scenarios presented in Section 2.1, physical devices play a central role. The MP3 player, the loudspeakers, the handheld computer, the projector, the screen, etc., offer PalCom services and are what users interact with physically in different ways. Devices also have a central place in the PalCom architecture. This is reflected, e.g., in the Discovery Protocol, where devices are announced as first-class citizens (see Chapter 5). The underlying intention is to make the connection between the software and the hardware devices easier to see for users. In contrast, Jini [121] and Web Services [22] focus on the services, making the connection to hardware devices secondary.

PalCom devices have two kinds of names, that are both announced in the Discovery Protocol. There is a short, friendly name that is shown in browsers, and, as discussed below in Section 2.5, there is also a unique name that is used for identifying the device on the network. Both names are independent of network-specific addresses, such as IP addresses.

The devices range from "small", specialized devices that provide a simple service—the loudspeakers and small devices carried around by users are examples—to powerful devices such as laptops. The more powerful devices have general functionality and can execute assemblies and load *unbound services*, which are PalCom services that are loaded dynamically at runtime.

2.3 Three kinds of services

In the scenarios, three kinds of PalCom services are used: *control*, *streaming* and *meta* services. All PalCom services fall into one of these categories:

- An example of a **control** service is the loudspeaker volume control service in the music scenario. Communication with a control service is done using *commands*, i.e., individual messages sent to or from the service. When John pressed "Volume up", a command was sent to the volume control service. Communication with control services can be bi-directional. This could be seen when the volume control service sent out a command indicating that the volume level was "Normal", which led to an update in the migrated user interface. The graphical notation for a connection between two control services was given in Figure 1.5 in Chapter 1.

- The MP3 player audio service is an example of a **streaming** service. Streaming services send out or receive a uni-directional stream of data of a certain type, such as the audio data sent from the music

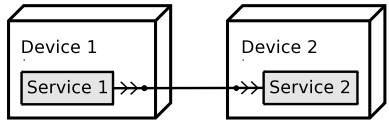


Figure 2.4: *A connection between two streaming services. Service 1 sends streaming data to Service 2.*

service to the loudspeakers. MIME types [52] are used for describing the data types of streaming services: `audio/mpeg` in this case. In the other example scenario, the Screen service is a receiver of a stream of images of type `image/jpeg`, and can be connected to senders of the same type, such as those in SlideShow and SendPicture. Figure 2.4 shows a connection between two streaming services, with double arrows indicating the direction of the streaming.

- **Meta** services are services that provide no "domain" functionality, but interact with other services based on their descriptions. An example of a meta service is *UIDisplay*, which is part of the browser on the mobile phone, and which renders user interfaces based on descriptions of other services. Other examples of meta services are logging services, that log messages from other services, and services for debugging. A connection between a control service and a meta service is shown in Figure 2.5.

It is the user that connects suitable services, either based on information about the data types the services can handle, or by initiating a user interface migration, and inspecting services in order to see if they match. All services have *service descriptions* that describe the set of commands or the data types supported. Examples of service descriptions are given in Section 2.6 below.

For the streaming services, the ability to connect them based on MIME types permit a simple kind of ad-hoc combinations. This is in contrast to systems that rely on standardized service types, like Jini [121], where the

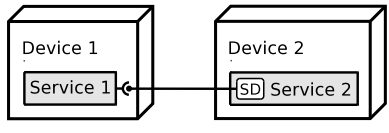


Figure 2.5: *A connection between a control service, Service 1, and a meta service, Service 2. Service 2 interprets the service description of Service 1, e.g. by rendering a user interface for it.*

Figure 2.6: *The tree of information about devices and services, as shown in John's PalCom browser in the music scenario.*

set of operations of a service has to be known when writing another service that uses it. I.e., in PalCom 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 Slides, or vice versa. This allows a service to be used in new, perhaps unforeseen, ways. Slides can be connected to any other service that can receive JPEG images as well, e.g., printers, file storage devices, etc.

2.4 Trees of services

There can be several PalCom services on a device. The services can be arranged in a tree structure, as shown in Figure 2.6. Services are at the leaves of the tree. Non-leaf nodes are used for grouping related services, and are not services themselves (the group "Music" in the figure is used for grouping the "Audio out" and the "Track selection" services on the MP3 player). This structure is presented to other devices on the network through the PalCom Discovery Protocol, and can be used when visualizing available services graphically. The grouping is for making the services on a device easier to understand for users: it has no pre-defined semantic meaning, and each of the services in the tree can be accessed individually.

2.5 Naming and versioning

For each device and service, there is a human-readable name that is displayed in browsers (the names shown in Figure 2.6). Devices and services also have other names, that are used below the surface for identifying them uniquely, and for keeping track of different versions of a service. These unique PalCom names abstract over technology-specific addresses, such as IP addresses, and are used independently of the underlying network technology. Chapter 5 presents the details about unique names and versioning information, which, because of the uniqueness requirements, are often quite verbose and are therefore normally not shown to the user.

The scheme for naming and versioning is strict in the sense that exact matching of names and versions are required in order for an assembly of services to execute. If the right version of a service cannot be discovered, the assembly will not interact with it. This is a corner-stone for the ad-hoc combinations and assemblies to work reliably. It is important that when a service is used in an assembly, the user can trust that the assembly has been tested with that particular version of the service. If not, he may have to test it himself first.

The naming scheme supports several instances of one service on a device. This is especially useful for unbound services, which can be loaded dynamically, as the result of an action by a user or an assembly. There is also support for simultaneous execution of multiple versions of the same service, which makes graceful upgrading possible.

2.6 Service descriptions

The details about how a service can be used and interacted with is specified in a *service description*. The two main uses of service descriptions are

(1) as **blueprints** for generated user interfaces, as will be discussed further in Section 2.8, and

(2) as **programmatic interfaces** to services when combining them in assemblies (see Chapter 4).

Service descriptions have different contents for different kinds of services:

- For control services, the service description contains a number of commands. Each command has a name and a direction, either *ingoing* or *out-going*. In-going means that the command goes over a connection to the service, out-going means that the service can send that command out over connections to other services. A command can

have parameters, and each parameter has a name and a MIME type. Before a command is sent, parameter values are filled in with data of the indicated type. In service descriptions, there is also a group construct, which can be used for grouping commands hierarchically. As an example of a service description for a control service, consider the Loudspeakers' "Volume control" service that is controlled via the handheld in the music scenario: an XML description for a simplified version of this service description is shown in Figure 2.7.

- For streaming services, the service description specifies the direction of the stream, and its MIME type. The XML service description for the "Audio out" service of the MP3 player is in Figure 2.8.

- For meta services, the service description lists the MIME types of command parameters, or streams, that the service can handle. The meta service does not restrict itself to particular names of commands. An example of a service description for a meta service, the UIDisplay service of John's handheld, is in Figure 2.9.

As can be seen in the service description examples, there is an XML format for service descriptions, which gives human and machine readability. We have chosen not to announce and use a binary class interface for services, as is done in Jini [121] and OSGi [6]. Those systems require Java programming for using services, while PalCom services are accessible in browsers and through assemblies. Another advantage with XML is that systems providing PalCom services, and those that do use them programmatically, can be implemented in any language, not Java only.

2.7 Asynchronous, peer-to-peer communication

An important thing to note is that all communication with PalCom services is *asynchronous*. All commands are sent in one direction between two services, and there is no built-in blocking of the sender, waiting for a reply from the receiver (for streaming services, there is no blocking either). One reason for this design decision is that the communication between services often takes place in networks with varying connectivity, where network delays and package drops can be frequent. In RMI [107], and other mechanisms based on Remote Procedure Call (RPC), 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. We therefore do not want to build waiting for responses into the protocols. Where synchronous communication is needed, that can be built on top of the asynchronous communication, and where guaranteed delivery of messages is wanted, mechanisms

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<SD id="Volume control">
  <CmdI id="Volume up" direction="in" />
  <CmdI id="Volume down" direction="in" />
  <CmdI id="Volume" direction="out">
    <PI id="level" type="text/plain" />
  </CmdI>
</SD>
```

Figure 2.7: *A service description for a control service, the "Volume control" service of the loudspeakers.* `CmdI` *elements are for commands, and* `PI` *elements are for parameters.*

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<SD id="Audio out">
  <StrI direction="out" type="audio/mpeg" />
</SD>
```

Figure 2.8: *A service description for a streaming service, the "Audio out" service of the MP3 player. The* `StrI` *element holds information about the stream.*

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<SD id="UI Display">
  <Type name="text/plain" />
  <Type name="image/jpeg" />
  <Type name="audio/mpeg" />
</SD>
```

Figure 2.9: *A service description for a meta service, the* UIDisplay *service of John's handheld. A* `Type` *element contains a MIME type that the meta service supports.*

for reliable communication in the Wire Protocol can be used (see Chapter 5).

Another point is that for control services, after the service description has been distributed and the connection has been set up, 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 not the case that a client always sends a request to a service that replies, or any other such pre-defined order. A PalCom service may send commands to other services at any time. It is up to the service programmer, who also writes the service description, to decide upon the details of this service protocol.

2.8 User interfaces

As illustrated by the two scenarios in Section 2.1, an important basic case is when a user connects to a PalCom service in a browser, and a user interface for the service is rendered there. The user can interact with the service remotely. The user interface may look as in Figure 2.2(a), which corresponds to the service description in Figure 2.7.

This simple user-interface mechanism is one of the key points in our approach to ad-hoc, non-preplanned interaction, as discussed in Chapter 1. It lets the user inspect the functionality of the service, examine the service operations in the user interface, and experiment with the service by interacting with it directly. When a user encounters services in a new environment, this is how they can be tried out. 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.

In the previous project [35], 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. In addition, our XML service descriptions can be used as programmatic interfaces from assemblies (as elaborated in Chapter 4), in a way that such Java applications cannot.

2.8.1 User in the loop

The user interface follows the structure of the service description, and in that sense the service becomes self-describing. It is the intention that users

should be able to understand how the service works, both when using it directly, when combining it with other services in assemblies, and when adapting assemblies after changes in service interfaces. This *user-in-the-loop* idea has similarities with the approach taken by Obje (see Section 1.8.2). The responsibility of sorting out the semantics of services is partly left to the user.

When rendering a user interface, the browser 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 service 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.

2.8.2 Rendering of user interfaces

Another aspect of the user-interface migration is that it gives the possibility to interact with devices with no or limited input/output capabilities. A small device without a display, or a device that is inconveniently placed in a room, can be reached and controlled. The browser typically runs on a device with suitable input/output resources, such as a display and some type of keyboard. The handheld works as a "universal remote control" for devices in the room.

Rendering user interfaces from abstract descriptions, such as PalCom service descriptions, also has the advantage that the different browsers on different devices, having different display capabilities, can use different ways to present the user interface. It can be rendered differently depending on screen-size, e.g. This is a research area in itself, see for example [51, 92], and there are a number of XML-based user-interface markup languages, e.g. UIML and XForms [4, 8]. Ubiquitous computing work in this area includes ICrafter [90], which also generates user interfaces from service descriptions, and the iStuff toolkit [7]. The adaptation to different target devices has not been our focus, though, and we have not worked on finding new dynamic mechanisms for it.

2.9 Related work

There are many systems where services communicate in networks. This section will relate the PalCom services to some of the most important ones.

2.9.1 Jini

Jini from Sun [121] is an early approach to support for combination of distributed services. Communication with services is done through *proxy ob-*

jects, Java objects that are kept in lookup directories. To use a service, a client first discovers a lookup directory, and then requests a proxy object for the service from there. The interaction with a service is done through method calls on the proxy object, which are transferred to the service across the network. A central part of Jini is the use of *leasing*: services lease their place in a lookup directory for a short while at a time, and if a service goes down unexpectedly, its entry can be removed from the directory when the lease has timed out. This gives a kind of *self healing* to the system.

The signatures of proxy objects have to be known when client applications are written. As discussed in Chapter 1, this problem has been tackled by defining domain-level standard service types. We argue this leads to a too static situation, where services cannot be combined in an ad-hoc way. As a partial relaxation, there is a specification for user interfaces in Jini, in the ServiceUI API [117]. 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 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.

The focus of Jini is programmatic, i.e. it is about programs that communicate. Proxies for services are defined as Java code, and in practice also the service provider is a Java program. In contrast, PalCom has a user focus, i.e., it is a user that finds and combines services, at least initially. In practice, Jini's tight connection to the Java language, with Java objects moved across the network, puts demands on all participating devices of running a JVM. This is too limiting for smaller devices, as mentioned in Section 1.8.2 about Obje. 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. Further, Jini is tied to Remote Method Invocation (RMI, [107]), 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.

PalCom uses a lightweight, XML description of services rather than Java code, which enables PalCom services and browsers to be implemented in any language. The PalCom service descriptions can be used both to directly drive user interfaces, and also as programmatic interfaces. In the latter case, an assembly will bridge between the services, rather than relying on standardized Java APIs that are defined and must be known prior to connecting to the service. Jini has no concept corresponding to the assembly.

2.9.2 UPnP

UPnP [114] is a technology designed to support automatic discovery and zero-configuration use of devices and services in a network. Devices are discovered and controlled from client applications, which are referred to as *control points*. UPnP specifies a set of protocols, not dictating a specific implementation on the devices. Device descriptions and messages to services are in XML, and UPnP does not rely on mobile code. There are no intermediate lookup directories, but control points discover devices directly, using a multicast scheme.

One difference, compared to PalCom, 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.

Like Jini, UPnP is based on domain-level standardization of device types. In addition to the 16 UPnP device types [113], there is also the DLNA [116] technology, which builds on UPnP and which specifies Digital Media Servers and Digital Media Players for the home. There is no concept in UPnP that corresponds to the assembly. In his master's thesis [64], Johan Kristell built bridging software that connected PalCom and UPnP networks. Using that bridge, a PalCom assembly could combine UPnP devices into a larger application. Given that different UPnP device types use different conventions for how UPnP actions and their arguments are structured, specific adaptations had to be done for each device type.

2.9.3 Zeroconf

Zeroconf [99] is a set of technologies for allowing IP devices to communicate in small, local networks, without infrastructure in the form of DHCP or DNS servers.[1] The goal of Zeroconf is the same kind of zero-configuration use of services as for UPnP.

Zeroconf has three legs: *addressing*, *naming*, and *browsing*. The addressing leg uses IPv4 Link-Local Addressing for obtaining an IP address in the absence of a DHCP server (or a manually assigned address). The naming leg uses multicast DNS for assigning names to devices that are not listed in any available DNS server. The browsing leg uses a protocol called DNS-SD [20] for allowing browsing for services from a Zeroconf-enabled application. In contrast to Jini and UPnP, Zeroconf does not specify any further steps that that: it does not specify how services are described or interacted with. There is a published list of public, standard Zeroconf service types [28], but the interaction with the service is entirely service-type specific:

[1] Apple's name for Zeroconf is Bonjour.

Zeroconf only distributes an IP host and port for connecting to a service. Therefore, we can only compare PalCom to Zeroconf at the discovery protocol level (see Chapter 5), not at the service interaction level discussed in this chapter, which is not covered by Zeroconf.

2.9.4 OSGi

OSGi [6] is a technology that is often mentioned in the context of *Service Oriented Architecture (SOA)*, the way of structuring systems into services that has gained much attention recently. OSGi decouples services' interfaces from their implementations, and, with its package concept called the *bundle*, it provides a much more flexible model for loading and unloading of services, than with the standard Java model which uses class files and JAR files. For the comparison with PalCom services, though, OSGi is not at the same level. The OSGi specification defines how services are discovered and used *within* one node on the network, and is not about networked services. In addition, OSGi builds heavily on Java, which has runtime implications in our context, as discussed for Jini above.

2.9.5 Web technologies

An 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 of devices, 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 [61]. The Wireless Microservers project put small Web servers into things, so they could be controlled using WAP over Bluetooth [50]. A fundamental limitation of the Web technologies, though, is that they are based on pull mechanisms. From a user interaction perspective, the user experience has been improved with the introduction of Ajax and related technologies [41], 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 PalCom services communicate. After a service description has been transferred in PalCom, there is a true peer-to-peer situation between two communicating services, with messages flowing in both directions.

Web Services

It is also interesting to compare *Web Services* to PalCom services. 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 [22]), for describing Web Services, by listing messages sent and received. Web services described using WSDL most often communicate using SOAP [48], a protocol defined on top of HTTP which is generally used for RPC-like communication. There are also so called *RESTful* Web services, which view services as Web resources and communicate without SOAP. With RESTful services, HTTP methods such as *POST*, *GET*, *PUT* and *DELETE* are used for manipulating the services directly, in a more light-weight way.

Regardless of representation, the description of a Web service 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 [120], current description technologies are not sufficient for describing the complete semantics of complex services. The notion of physical devices, which is important to facilitate in pervasive computing, is missing in Web Services. Further, the way of announcing and discovering available Web Services, called UDDI (Universal Description Discovery and Integration, [79]), is based on a central registry where services are looked up. As will be discussed in Chapter 5, relying on a central registry is not feasible for the PalCom architecture, even if the registry can be managed collectively by multiple nodes, as in UDDI.

The Semantic Web

In relation to Web Services, the effort to build a Semantic Web should also be mentioned [10]. 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 classifications and relationships among terms. When constructing a service, the programmer describes the service, using the OWL language (Web Ontology Language [26]). The OWL description can then be used by *agents*, 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.

2.10 Summary

- PalCom devices and services offer textual, XML descriptions of themselves, and no domain-level standardization or ontology is used.
- Service descriptions can be used both as blueprints for generating user interfaces, and as programmatic interfaces.
- Rendered user interfaces make the services self-describing, and facilitate experimentation with discovered services.
- Communication between services is asynchronous, and services are in a peer-to-peer relationship when they communicate.

Chapter 3

Connections

When services communicate, through commands or through streaming, data is transmitted in the network between the services. In PalCom, such a path of communication is referred to as a *connection*. In the scenarios discussed in Chapter 2, there are connections between the audio service on the MP3 player and the audio service on the loudspeakers, between the volume control service on the MP3 player and the UIDisplay service on the handheld computer, etc.

Connections are made explicit in the PalCom architecture. They are announced in the Discovery Protocol, and can be viewed in a browser when inspecting the communication situation in, e.g., a room. This support for connections gives more visibility for the user, and the announced connections are useful for the execution of assemblies. While executing an assembly, an assembly manager monitors what connections are currently up and what need to be established, as discussed further in Chapter 4.

This chapter discusses the concept of connections from a user perspective. At the end, there is a brief discussion about the relation to *connectors* in the field of software architecture. Chapter 5 goes into details about how the PalCom communication protocols support connections.

3.1 Connecting two services from a third device

In PalCom it is not only possible to connect a service on a local device to a service on a second device, but it is also possible to connect two services from a third device. This was exemplified in the slide show scenario, where the user connected the laptop's slide show to the projector's Screen, using the browser on the mobile phone. In order to allow this, there is a simple protocol, called RemoteConnect (see Chapter 5), which can be used by browsers or services 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 Slides service. Similarly, it is possible to disconnect two services that are currently connected. This functionality builds on the property of the Discovery Protocol, that devices announce information not only about the devices themselves, and about their services, but also about established connections. The user can see not only what devices and services there are, but he can also view and control the connections. This is important in order to be able to inspect the communication situation.

3.2 Properties of connections

An established connection is a path of communication between PalCom services. However, there is no built-in overall assumption about the delivery of data over the connection, like, e.g., for TCP connections. Not all PalCom connections implement guaranteed delivery of messages: one of the properties of a connection is whether it is used for *reliable* or *unreliable* communication, as defined by the PalCom Wire Protocol (see Chapter 5). For neither reliable nor unreliable communication, there are any periodic connection maintenance messages, which could give problematic overheads. Instead, timeouts in the Discovery Protocol are used for cleaning up connections where one of the parties has gone down uncleanly.

Another property of a PalCom connection is its *topology*. The basic case, and the case appearing in the scenarios in Chapter 2, is a unicast, one-to-one connection between two services. There are also *radiocast* connections (one-to-many), and *groupcast* (many-to-many). In Figure 3.1, Service 1 sends to Service 2 and Service 3 over radiocast connections. Radiocast connections can be used where several services connect to and receive from one service, which sends out commands or streaming data, but does not itself receive. The receiving services are aware of the identity of the sender, but not the other way around. A typical example is streaming of real-time data over a shared medium, where performance is increased by having several services listen to the same transmission, instead of sending to each in turn. With multiple loudspeakers receiving audio data in the music scenario, this would be beneficial.

Groupcast connections are used where a set of services communicate according to a common protocol, and all can potentially both send and receive. In Figure 3.2, Service 1 and Service 2 are both connected to the group named *Group*. Neither service needs to be aware of the identities of the others in order to communicate. Examples of this are communication between PalCom assembly managers for distributing information about assembly updates (see Section 6.6), and communication between tiles in the Active Surfaces scenario during puzzle games (see Sections 1.5.2 and 9.4).

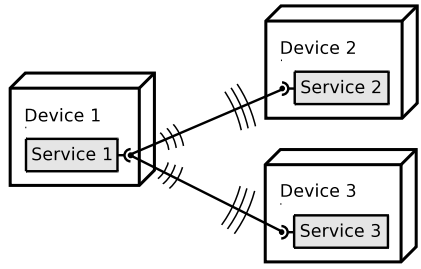


Figure 3.1: *Radiocast connections. Service 1 sends to Service 2 and Service 3.*

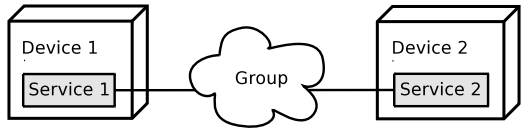


Figure 3.2: *Groupcast connections. Two services are connected to the group named* Group.

3.3 Multiple networking technologies

In several of the user scenarios studied in PalCom, devices communicate across different networking technologies. This can be seen, e.g., in the GeoTagger, Active Surfaces and Tall Ships' Race scenarios, introduced in Section 1.5. In the GeoTagger scenario, the landscape architect's handheld computer communicates both with the other devices that he has with him out in the field, using short-range wireless communication such as Bluetooth, and with the storage server at the office, using GPRS. In Active Surfaces, the *assembler tile* communicates both with other tiles using infrared, and, during the configuration phase, with a PC using IP over Ethernet. At the Tall Ships' Race, there was both short-range and long-range wireless communication, as well as wired.

Thus, there is a requirement that PalCom connections can span multiple network technologies. This requirement has been taken into account in the communication protocols, presented in Chapter 5. The approach is to define a PalCom addressing scheme and message format that abstracts the technology-specific details, hiding them down in the lowest protocol layers. At the level of connections, and communication between services, the services are only aware of PalCom messages, and do not treat different networking technologies specifically.

3.4 Tunnels

As mentioned in Chapter 1, PalCom discovery is, in the first step, limited to the local network. Services cannot directly discover each other, and communicate, across multiple routers on the Internet. This is for shielding both the user and the hardware and software from overwhelming numbers of discovered devices and services.

Still, a service should be able to connect to a service outside the local network where needed, potentially at the other side of the world. This has been solved through the use of tunnels, as discussed in Section 1.6.6 in Chapter 1 (see [38]). Tunnels, which are set up manually, connect geographically distant PalCom networks in a secure way. Through each tunnel, there can be multiple PalCom connections of the topologies discussed in the previous section, connecting services and assemblies to each other.

3.5 Related work

In the field of software architecture, *software connectors* make interactions between components explicit, and they are treated as first-class elements [74]. Architecture Description Languages (ADLs) model connectors as links of communication between components [73], and connectors are explicit in the language ArchJava, which extends Java with mechanisms for expressing the architectural structure of an application within the source code [5]. The concept of software connectors is used not only between nodes in a distributed system, but also for describing transfer of control, and for conversions and adaptations between components. A background to this work is that in software implementations, interactions between components are often much less clearly described than the components themselves. In ArchJava, the explicitly declared required interfaces associated with connectors are intended to reduce coupling between components, and promote understanding of the components in isolation. Further, a connector used to bind two components together is specified in a higher-level component, so that the communicating components are not aware of and do not depend on the specific connector being used [5].

The way of making connections explicit in the PalCom architecture, and in the information given to the user through the Discovery Protocol, has similarities with the concept of software connectors. The points made in ArchJava about loose coupling, and about specification of connectors in higher-level components, are valid also for PalCom connections, which are established by the user or by an assembly, and not initiated by services themselves. This property is discussed more in the following chapter, about assemblies.

3.6 Summary

- PalCom connections are first-class, and discoverable in par with devices and services.
- It is possible to connect and disconnect two services from a third device.
- Connections can have different distribution schemes (one-to-one, one-to-many, many-to-many), and transfer commands as well as streaming data.
- Connections can connect services on a local network, networks connected through PalCom routers, and aggregated PalCom networks through PalCom tunnels.

Chapter 4

Assemblies

The elements of the PalCom architecture discussed in Chapters 2 and 3 let the user discover and interact with *individual* PalCom devices and services. The concept of an *assembly* is defined in order to enable the use of *combinations* of services. The assembly can be defined by a user or by a developer. Optionally, it can contain logic in the form of a script, extending the combined behaviour. As discussed in Chapter 1, the assembly is intended to be used with services available in a particular situation. The user can try out the services and the assembly in the interactive assembly editor in a browser, both when creating a new assembly, and when adapting one that has been created for similar services on other devices. This possibility to explore available services and try things out is important for being able to create useful ad-hoc combinations of services.

In the next section, two scenarios follow that illustrate how PalCom assemblies are used. After that, the assembly concept is discussed in more detail: how assemblies are structured, and their role in providing flexible combinations of services.

4.1 Scenarios

Consider the following simple scenario:

> *A university professor has weekly lectures in room E:1406 at the university. On the first lecture, she creates an assembly RemoteSlideShow by connecting a slide show service on her laptop, the PalCom browser on her mobile phone, and the video projector in room E:1406. This is done by a few graphical commands on the phone. After that, she uses her phone to select the desired presentation, and to step through the slides. At the next lecture, she simply activates the existing assem-*

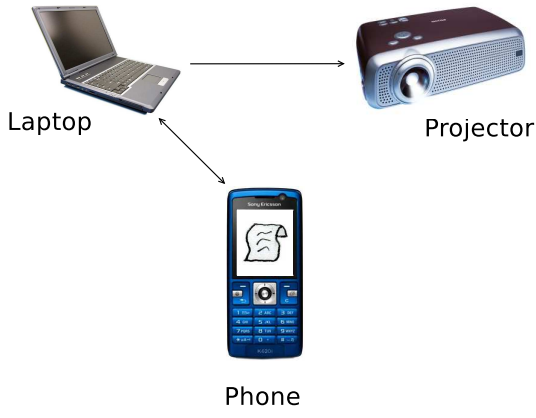


Figure 4.1: *RemoteSlideShow. An assembly on the mobile phone (shown as a scroll symbol) sets up connections for controlling the slide show from the mobile phone, and for displaying slide show images on the projector.*

> *bly, which will then discover and connect the devices according to the assembly descriptor. She can then immediately select the appropriate presentation and step through the slides.*

RemoteSlideShow is an example of an assembly that consists only of a set of connections between services on particular devices, and has no further logic of its own. It builds on the slide show scenario presented in Section 2.1.2. The assembly composes a video projector, a laptop, and a mobile phone, as shown in Figure 4.1. Slides are sent from the laptop to the video projector, and the user controls the actions, next slide, previous slide, etc., from the mobile phone, where the assembly itself also resides. The RemoteSlideShow assembly is a way for the professor to automate the establishment of connections, avoiding having to do it manually each time she comes to the lecture room.

Another example of an assembly is one that combines a photo service on a camera device with a coordinate service on a GPS device and a direction service on a compass (see Figure 4.2). This is the GeoTagger scenario, which is a little more complex, and which was introduced in Section 1.5.1:

> *Mark is a landscape architect at a landscape architecture firm. He regularly goes out in the field for* visual assessment, *documenting a site for understanding how a planned project would affect the scenery. Over coffee, Mark and his colleagues get the idea that it would be nice if the equipment that Mark carries with him could automatically tag the photos he takes with GPS coordinates and compass directions.*

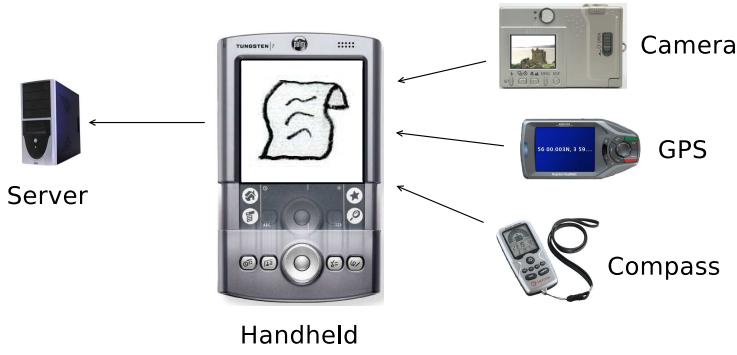


Figure 4.2: *The GeoTagger scenario. Assemblies on the handheld computer combine services on the camera, the GPS and compass, tagging images before sending them to the server at the office.*

That would help when sorting the pictures back at the office. Mark has devices performing all the necessary functions—the camera takes pictures, the GPS and compass deliver coordinates and directions—but the desired combined behaviour is not provided by any of the devices.

As all the devices are PalCom-enabled, Bill, the firm's computer specialist, realizes that it would be easy for him to create an assembly that performs the desired actions. Bill borrows Mark's devices and opens the PalCom browser on his computer, where the devices and their services show up as discovered. He inspects the service descriptions and experiments with the services by opening their user interfaces in the browser. In order to reduce complexity, and to make the assemblies more reusable, Bill chooses to to make two assemblies: one, which he calls TaggingCamera*, for handling the problem of tagging the pictures, and another one,* Uploader*, that uploads the pictures tagged by TaggingCamera to the storage service at the office.*

For TaggingCamera, Bill starts from a service CoordinateStuffer *that he finds in a repository of PalCom unbound services on the Internet. CoordinateStuffer takes a JPEG image, a GPS coordinate and a compass direction, and writes the coordinates and directions as metadata in the JPEG image. By dragging-and-dropping in the browser, Bill declares connections from the assembly to services on the camera, on the GPS, on the compass and on the CoordinateStuffer. In the assembly, he makes a small script. This is done using drag-and-drop, by dragging commands from discovered service descriptions. The script saves the latest coordinate and direction each time they are received from the the GPS and the compass. It also reacts when a new picture has been taken by the camera, by asking CoordinateStuffer to tag*

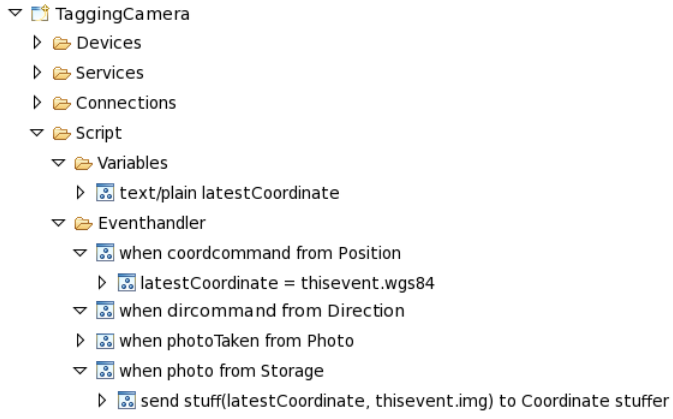


Figure 4.3: *A view from Bill's assembly editor, while he works on the TaggingCamera assembly. He has just added an event handler that sends the image to CoordinateStuffer for tagging, when it has been received from the storage service on the camera.*

the image with the latest saved coordinate and direction. Figure 4.3 shows TaggingCamera, as it looks in the assembly editor while Bill is working.

Further, Bill declares a synthesized service, provided by the TaggingCamera assembly, that makes the tagged image available for users of the assembly. After that, he tests the assembly by starting it and connecting to its synthesized service from the browser. A user interface is shown. When Bill takes a test picture with the camera, it shows up in the interface.

Bill continues with the Uploader assembly. He declares connections from the assembly to the synthesized service of TaggingCamera, and to the storage service at the office. When a new tagged picture is available from TaggingCamera, a script in Uploader sends the picture to the storage service. Bill tests Uploader by starting it and taking a picture with the camera. He checks on the server and sees that the picture has been saved there. When he looks at it in his image viewing program, he can see that the coordinate and direction have been saved as image meta-data.

On the next day, Mark brings his equipment to a site for doing visual assessment. On his handheld computer, he has the two assemblies and the CoordinateStuffer that he got from Bill. Bill has set up a tunnel over GPRS, connecting the small network formed by Mark's devices to the office. Out in the field, Mark starts his PalCom browser on

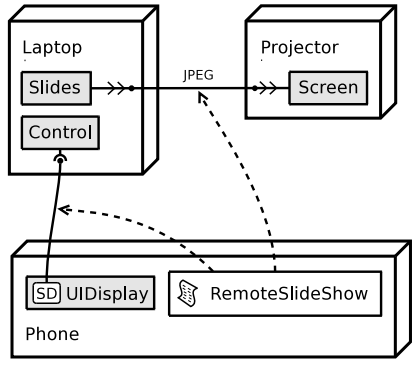


Figure 4.4: *Deployment diagram of the RemoteSlideShow assembly.*

> *the handheld computer, where the devices and their services are automatically discovered. He starts the two assemblies, and can see in the browser that connections are established between the services and the assemblies. He starts working, and each time he takes a picture, it is tagged and sent to the storage server back at the office.*

This example illustrates several key aspects of assemblies: how they combine services of other devices, how behavior can be automated, how an assembly itself can be seen as a service (through its synthesized service), and thereby how new assemblies can be built on top of existing ones.

4.2 The anatomy of an assembly

An assembly consists of

- references to **devices** included in the assembly,
- references to **services** on those devices,
- declarations of **connections** between the services, and between the services and the assembly,
- an optional **script**, defining what actions the assembly takes upon receiving messages from services, and
- an optional set of **synthesized services**, offered by the assembly.

When the assembly is executed, it monitors the network and makes sure the listed connections are established. Figure 4.4 shows how RemoteSlideShow and the involved services are deployed on the different devices. The

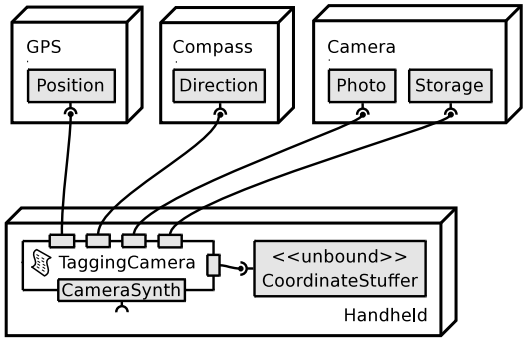


Figure 4.5: *Deployment diagram of the TaggingCamera assembly.*

assembly, marked with a scroll symbol, executes on the mobile phone. The dotted arrows point to the connections managed by the assembly. On the laptop, there is a streaming service called Slides that sends out JPEG images (the double arrows denote a streaming service). The service Screen on the projector receives the images and displays them. For controlling the slide show, the meta service UIDisplay on the phone is connected to the service Control on the laptop. The meta service takes the service description of Control (marked with SD) and renders a user interface for it.

It is possible to add coordinating logic to the assembly in the form of a script. Figure 4.5 shows the TaggingCamera assembly, which executes on the handheld. TaggingCamera has five connections to services: to Position on the GPS, to Direction on the compass, to Photo and Storage on the camera (the camera offers its functionality split into multiple services), and to the unbound service CoordinateStuffer on the handheld. The script in TaggingCamera receives and sends commands over these connections. The synthesized service of TaggingCamera, CameraSynth, is shown as a service symbol at the border of the assembly: it is a service offered by the assembly.

Uploader also has a script, and also executes on the handheld, as shown in Figure 4.6. It builds on the functionality provided by TaggingCamera, and therefore has a connection to TaggingCamera's synthesized service. There is a connection to the service PhotoDB on the server at the office. The server is on a remote network, and Uploader is connected to it via a tunnel. The tunnel has been set up by Bill, and is handled at the PalCom communication protocol level. The Uploader assembly is not aware of it.

As shown in the examples, an assembly sits between a number of services, governing their interaction. It can be seen as a kind of tailored wiring between the services. Connections directly to an assembly come into play for assemblies with scripts, as for TaggingCamera and Uploader. The assembly-

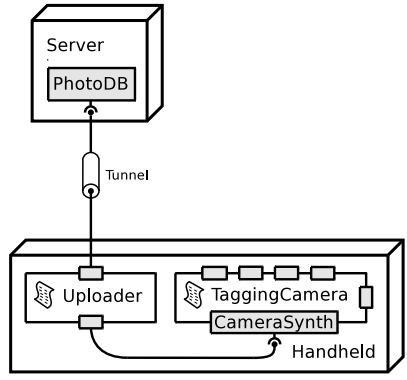


Figure 4.6: *Deployment diagram of the Uploader assembly.*

side end-points of these connections can be seen as services, representing the part of the script functionality that deals with that specific connection. We refer to these services as *implicit services*, because their behaviour, in terms of commands sent and received, is implicitly defined by the behaviour in the script. Connections to implicit services cannot be initiated from the outside: only the assembly itself can establish them. In the figures, this is indicated by showing implicit services as gray service symbols without hooks. For each implicit service, a service description can be automatically generated from the event handlers that describes the interaction with that specific service. Taken together, the service descriptions of implicit and synthesized services describe an assembly's command communication.

4.3 Assembly descriptors

An assembly is deployed according to an *assembly descriptor* that defines the devices, services and connections present within the assembly, and the optional script and synthesized services. The script is written as a set of event handlers, and the actions possible in the script are to send messages to services, and to store values in local variables.

Assembly descriptors are stored in XML, but there are several presentation formats. One is used for presentation and editing in the browser (Fig-

```
1  assembly TaggingCamera 1.3 released {
2    this = ServiceID;
3    devices { ... }
4    services { ... }
5    connections { ... }
6    script {
7      variables {
8        text/plain latestCoord;
9        text/plain latestDirection;
10     }
11     eventhandler {
12       when coord from Position {
13         latestCoord = thisevent.value;
14       }
15       when dir from Direction {
16         latestDirection = thisevent.value;
17       }
18       when photoTaken from Photo {
19         send getPhoto() to Storage;
20       }
21       when photo from Storage {
22         send stuff(latestCoord, latestDirection,
23           thisevent.img) to CoordinateStuffer;
24       }
25       when imageTagged from CoordinateStuffer {
26         invoke taggedPicture(thisevent.img);
27       }
28       when takePicture from CameraSynth {
29         send takePhoto() to Photo;
30       }
31     }
32     synth CameraSynth {
33       in  takePicture();
34       out taggedPicture(image/jpeg img);
35     }
36   }
37 }
```

Figure 4.7: *Assembly descriptor for the TaggingCamera assembly.*

ure 4.3), and a more programming-language-like syntax is used in Figure 4.7, which shows the assembly descriptor for TaggingCamera:[1]

- Line 1 shows the name of the assembly and versioning information, and also line 2 contains versioning information, as explained in Chapter 6.
- There are variables in the script for holding the latest GPS coordinate and compass direction, which are sent out periodically by the devices (lines 8 and 9).
- When a coordinate arrives, it is saved (line 13), and similarly for the direction (line 16).
- When an indication comes from the Photo service that a photo has been taken, a request for the photo is sent to the Storage service on the camera (line 19).
- In the next event handler, the returned photo is sent to the CoordinateStuffer (line 22).
- As the tagged image is returned to the assembly, it is sent out through the synthesized service by means of an `invoke` action (line 26).
- The synthesized service is defined on lines 32–35, with one in-going and one out-going command.
- The in-going command `takePicture` is handled on line 29: a command `takePhoto` is sent to the Photo service. This makes the synthesized service provide the functionality of "pressing the button" on the camera.

The assembly descriptor for Uploader has a similar structure, but without variables and with only one event handler that forwards a tagged picture from CameraSynth to PhotoDB. RemoteSlideShow has no script, but only declarations of devices, services and connections. See Chapter 6 for an in-depth discussion of assembly descriptors.

4.4 Bindings

In the assembly descriptor, different kinds of *bindings* can be specified, which govern how the assembly behaves when services appear and disappear. The two types of bindings are (1) whether the presence of a service is *mandatory* or *optional* for the assembly to function as intended, and (2)

[1] The contents of the `devices`, `services` and `connections` sections have been omitted for brevity, see Chapter 6 for examples.

alternatives. Alternative bindings mean that one of a prioritized list of services should be used. This can be used for redundancy, where services act as back-ups for others, e.g. in situations with unreliable network connectivity. Optional bindings can be used in scenarios where one or more of a set of services are adequate to use, or where a service is not necessary for the core functionality of the assembly. Bindings are specified in the `services` section, as detailed in Chapter 6.

4.5 Synthesized services

Synthesized services, as exemplified in the TaggingCamera assembly, are services that offer some combined functionality of the included services to other devices in the network. They can capture aspects that are not covered by the individual services themselves. Synthesized services can forward messages to and from the services included in the assembly.

Synthesized services behave as any other service on the network, and can be interacted with directly through user interfaces in a browser. This property makes it possible to use synthesized services in other assemblies, and build a hierarchy of assemblies. Such a hierarchy is exemplified where Uploader builds on TaggingCamera. This way, complex systems of services can be handled, with assemblies grouped at multiple levels (see also Figure 1.8 on page 14).

4.6 Unbound services

The assembly language is simple, with only a limited set of possible actions. But, as seen in the GeoTagger scenario, services written in a general-purpose programming language, such as Java, can be incorporated into an assembly. This way, the assembly can perform more advanced functionality than what can be specified directly in the script. CoordinateStuffer is an example of such a service, referred to as an *unbound service*. The name reflects that these services are not tied to the hardware of any particular device, but built for performing computations in assemblies.

One of the ideas behind the unbound services is to keep the script language simple. The language should be possible to work with for an end user. We do not want to add features to the language that allow computations as complex as the ones performed by CoordinateStuffer. Instead, they are left to unbound services. The possibility of unbound services means that a user with knowledge of Java is not locked in by the restrictions of the script language.

4.7 Assembly managers

Assemblies are executed by *assembly managers*. The assembly managers can be integrated in browsers, or run stand-alone on devices in the network. They store assembly descriptors, and handle the execution of assemblies by monitoring the network and establishing connections that are possible but not already established. The assembly managers interpret scripts, and execute event handlers when messages arrive at an assembly. The running assembly is announced as running on the device where the assembly manager runs.

An assembly manager runs on one device, but the assembly descriptor can be moved to another assembly manager with no or little modification and execute there. This was done in the GeoTagger scenario, when the assemblies made and tested by Bill were moved to Mark's handheld. It can also be done in order to optimize use of network bandwidth. E.g., if the camera in the GeoTagger scenario had an assembly manager and capabilities of executing unbound services, TaggingCamera and CoordinateStuffer could have been put there, minimizing the number of JPEG image transfers over the network. Note that moving an assembly does not affect the services it uses, and is thus a lightweight operation that can be performed by an end user.

4.8 Configuration, coordination and computation

An assembly combines a set of services on devices into a larger system. A useful way to look at such a system is from the perspectives of *configuration*, *coordination* and *computation* (see [16, 63, 124]). Different parts of the assembly descriptor capture those different aspects of the system, and this separation is key to the usefulness of assemblies as a mechanism for ad-hoc combinations:

- **Configuration** is the most coarse-grained of the three system aspects. It specifies what devices and services are involved in the assembly, and how they are connected.
- **Coordination**, at the next level, regards the communication between the connected devices and services, and how they interact over the connections by sending messages to each other.
- **Computation**, finally, is the most fine-grained aspect, specifying the internal operation of individual devices and services.

The three aspects can be illustrated by the deployment diagram for TaggingCamera (Figure 4.5 on page 58):

- The *configuration* specifies that there are four devices (GPS, compass, camera and handheld) and five used services on those devices (Position, Direction, Photo, Storage and CoordinateStuffer), and how these are connected to the assembly. Further, the interface of the synthesized service is also part of the configuration, because it defines how the assembly can work as a building block in other assemblies. Specifications of bindings are part of the configuration as well.
- The *coordination* part is specified in the script, where the sequencing of messages over the connections is defined.
- *Computation*, finally, is in the implementation of the services, both in native services (which are tied to the devices), in unbound services (CoordinateStuffer in the example), and, recursively, in synthesized services of other assemblies. The computation is thus primarily in the implementation of a service, rather than in the assembly descriptors.

The main advantage of separating configuration, coordination and computation is that the aspects can be varied independently:

- The *configuration* can change dynamically at run-time, if flexible bindings are used, or be changed manually by a user, who modifies an assembly descriptor using a tool. Manual modifications are typically made for adapting an existing assembly to a new set of available devices.
- If the behaviour of services on the new devices differ from the old ones, the next step is to modify the *coordination* in the assembly by changing the script. That would typically require deeper understanding of the behaviour of the services.
- Adapting *computation* in the assembly, finally, means to modify one of the involved services, which requires programming in Java or another general-purpose language, or the incorporation of a pre-written unbound service.

Chapter 6 will discuss the language of assembly descriptors in more detail, and how it supports the separation of the three aspects.

Linda [16] is the first example of a *coordination language*, a language that makes an explicit split between computation and coordination. Linda is based on a *tuple space* model, where processes communicate by inserting tuples into an abstract "space" using the operation *in*, and by reading from the space using *out* or *read* (blocking and non-blocking, respectively). One

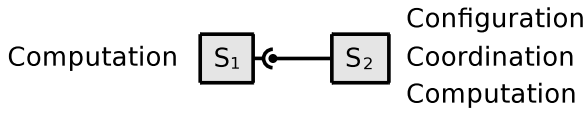


Figure 4.8: *The traditional approach.*

motivation behind the language design is to give an uncoupled programming style, where senders and receivers are not tightly dependent on each other. The language has the property of *communication orthogonality*, which means that the sender of the message needs not have any prior knowledge of the receiver of the message, just as the receiver has no knowledge of the sender in the case of a procedure call. This way of using the few simple, atomic operations for communication between processes is intended to give the mentioned separation of coordination from computation.

The separation of the concerns of computation, coordination, and configuration has also been studied in the field of software maintenance, e.g. in the work of Wermelinger et al. [124]. That paper proposes a three-layer architecture, separating each concern into one layer, for supporting system evolution. The presented system allows reconfiguration at runtime, and the separation makes it possible to vary the three aspects independently.

4.9 The assembly is external to the services

From the perspective of ad-hoc combinations, the separation of computation from the other two aspects, configuration and coordination, deserves special focus in PalCom. The property that computation is in the services, and the other two are expressed in the assembly descriptor, makes the assembly *external* to the services. A service has no knowledge of other services that it is connected to. This makes it possible to adjust the assembly after changes in new versions of service descriptions, and to include new services after the initial construction of the assembly: the assembly concept is *designed for handling changes in services*. In addition, combinations can be made 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.

A metaphor with wires and connectors can be used for illustration. The traditional case of interoperation between two services is shown in Figure 4.8, 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. As indicated in the figure, the implementation of S_2 specifies both configuration, coordination and computation.

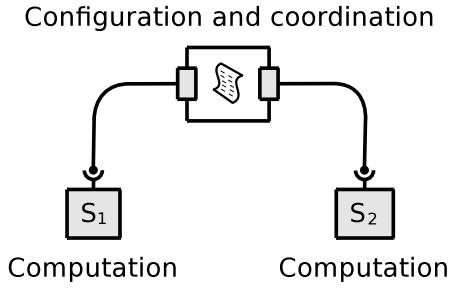


Figure 4.9: *Our approach.*

In our approach, shown in Figure 4.9, both services have female connectors. It is the assembly in the middle that connects to both services using male-connector wires, and there are no direct dependencies between services. Here, computation is in the services, and the assembly specifies configuration and coordination. The assembly works as a mediator, and the advantage is that when either S_1 or S_2 changes, it is possible to change the mediator, 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.

In the GeoTagger example, it can be noted that the camera, the GPS and the compass are not prepared in advance for this scenario, except being PalCom-compliant at a general level. The service descriptions are used as programmatic interfaces by the assembly, which runs on the handheld. The preparation of the handheld consists of installation of the special CoordinateStuffer service, and of writing the assembly script. Further, if after some time of use the service on the GPS is updated to a new version, it is possible to accommodate for the new service interface by modifying the assembly. The modifications do not have to be done in the implementations of the services on the handheld, the camera and the compass.

The idea of having an assembly in the middle can be compared to the Mediator design pattern [39, p. 273]. The Mediator pattern gives similar advantages of looser coupling, and the possibility to vary the interaction between objects independently. Mediators have also been studied in the field of software architecture [101].

4.10 Connecting services directly to each other

When discussing the wire-connector metaphor, the following question may arise: does having the assembly in the middle mean that communication always has to go via the assembly? The answer is no: the RemoteSlide-

Show assembly (Figure 4.4 on page 57) is an example where data flows over connections directly between services. That assembly does not violate our principles of ad-hoc combinations: the configuration is still in the assembly, external to the services (as illustrated by dotted arrows in the figure). In addition, none of the services has been created especially for cooperating with the other:

- One of the connections is a streaming connection, for which the matching of MIME type decides whether the connection makes sense. This kind of data type matching enables connections that are more *ad-hoc*, compared to matching based on named commands with typed parameters. Matching based on data types allows meaningful connections of services from different domains. One example is that the JPEG slides could equally well be connected to a printer service that receives a JPEG stream, without the Slides services having to be aware of the notion of printing. This kind of type-based connections have also been explored in the Obje project [81].
- The other connection in RemoteSlideShow is to the UIDisplay meta service, which can render any type of command in a user interface. Meta services are not created for cooperating with any specific service.

Common to all direct connections between services is that there is no coordination part in the assembly for these connections—no event handlers are used. This is simply because commands flowing over the connection do not reach the assembly. An advantage of direct connections can be performance, that communication does not have to go over the network to the assembly, but can go directly between the services. This is important in particular for services which send or receive a continuous flow of information, such as audio and video.

4.11 End-user work patterns

From the scenarios and application prototypes studied in PalCom, we envision that end users will define and interact with assemblies in an exemplar, or prototype-based way, in analogy to prototype-based object-oriented programming languages, as advocated by Lieberman [69] and in the language Self [112]. An assembly is built as an example directly in an environment where it can be used. A user can switch between explicitly remote controlling services, and constructing an assembly that connects and controls the services. When the user so decides he or she can share an assembly with others using an object cloning metaphor rather than object instantiation. A user will typically also change an existing assembly "in vivo", that

is, when the devices and services it depends on are available. This allows changes to the assembly to be tested right away. The user can also experiment with assemblies using simulated devices and services, as will be discussed in Chapter 8.

An example of this kind of live editing could be the landscape architect who in the field realizes that the GeoTagger assembly he brought with him does not work. He opens up the assembly in the browser and notices that the GPS is shown as not available. Closer inspection of the device shows that it is out of battery. Luckily he has a spare GPS in the car. He turns it on and interacts with it directly through his handheld (where the assembly is also stored). He concludes that the spare GPS works the same way as the failing device, and modifies the assembly so it will use the new device. He runs the complete assembly while inspecting both the assembly and the GPS, to verify that the pictures taken are marked with the proper coordinates from the GPS.

4.12 Programming at different levels

For the use of assemblies 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. Our aim is to enable end users to work with a large portion of assemblies. Configuration, and coordination up to a certain level of complexity, can be expressed in the assembly descriptor. For the most complex cases, we provide the possibility of incorporating an unbound service. The unbound service can handle computation-level tasks, and can also be used for cases with more complex coordination than what can be expressed in the assembly script.

Putting together different kinds of assemblies requires different levels of programming skill. Interacting directly with services, in a *remote control* fashion, is possible for an end user, using simple operations in a PalCom browser. The same is true for setting up simple assemblies, that are basically a set of connections between services, saved for later re-establishment. Working with unbound services requires knowledge about a general-purpose programming language. We see the middle case, constructing a scripted assembly with an assembly script, reasonable to manage for an advanced end user.

Recognizing these different levels of user expertise, the main distinction is still between the programmer of the services on a hardware device, and the user of an assembly that combines two or more such services—between the *computation* on the one hand, and *configuration* and *coordination* on the

other. The pre-programmed services on the device cannot be assumed to be possible to modify for the end-user. Those are often proprietary software, and not available for change. They may access hardware functionality, and be implemented in a low-level language for running on a resource-constrained device. If the services are PalCom services, possible to combine with others in an assembly, the cooperation between services *is* possible to modify, because the assembly in the middle can be changed. This is not the case if the interoperation is built into the services.

4.13 Related work

The research area of service composition is very active. In [13], Brønsted et al. make a survey of service composition mechanisms in ubiquitous computing, and place PalCom and other projects in a categorization matrix. Projects are categorized according to the composition specifications used, and according to runtime and deployment issues. Of projects that let the end-user create service compositions, like PalCom does, Aura [40] is more geared towards agents and autonomic behaviour. The same is true for Amigo [115], which is based on Semantic Web technologies. ICrafter [90] has mechanisms for service composition, and for generating user interfaces for the combinations. ICrafter's service composition mechanisms are based on letting services implement certain generic Java interfaces, such as DataProducer and DataConsumer, and on *patterns* of such generic interfaces for which user-interface generators can be written. The patterns are expressed using a regular expression syntax, and the matching relies on matching data types. Our impression is that the PalCom assembly is more powerful than ICrafter's regular-expression patterns, and it is not dependent on Java programming, as the user-interface generators are. Obje [81] is focused on letting the end user create direct connections between components, based on matching data types, while PalCom has the assembly concept for more complex interactions. The *application bridges* in Gaia [94] are created by developers, and only used by end users.

Tuple spaces, as introduced by the Linda language [16], have been used in several ubiquitous computing projects, for achieving loose coupling and more data-oriented interaction between services. An example is Interactive Workspaces [57]. One.world [44] uses tuples for representing data. PalCom has, however, not opted for service communication based on tuple spaces, partly because PalCom has no fixed surrounding environment where the tuple space can be implemented, like the Interactive Workspaces iRoom has. We have instead chosen to structure the communication between services and assemblies in commands with typed parameters, and in typed data streams.

In the Web Services world, composition of Web Services can be done using both *orchestration* and *choreographies* [89]. Both mechanisms are intended for describing *workflows* including the services:

- Orchestration, as specified using the language BPEL (Business Process Execution Language [80]), is most similar to our assemblies: it gives an executable process that can actively send messages to its included services.

- A choreography, as specified in WS-CDL (Web Services Choreography Description Language [58]) is more like a contract that is signed by a number of cooperating service providers, and which can be checked while the services cooperate.

Both BPEL and WS-CDL are expressed in XML, and work with Web Services described using WSDL [22]. Both have constructs for sending messages to services, for receiving messages from services, for performing actions in parallel and in sequence, and for loops. BPEL orchestrations are executed by orchestration engines, corresponding to our assembly managers.

BPEL orchestrations specify *partner links* that correspond to connections in a PalCom assembly. For each partner link, the interface of the Web Service at one or both ends is specified in the BPEL description (partner links can only be one-to-one, not one-to-many or many-to-many like PalCom connections). In general, one of the partner links goes to the user of the process. The process side of that link can be compared to a PalCom synthesized service, because it serves as the service interface of the process itself. However, the BPEL specification does not mention the use of such interfaces for structuring BPEL processes at multiple levels, like PalCom assemblies can be structured with synthesized services. Furthermore, the service interface of the link to the user is not treated especially in a BPEL description, unlike the interface of the synthesized service for a PalCom assembly.

A point in common with assemblies, at a more conceptual level, is that with both BPEL and WS-CDL, the composition is external to the services, and modifications can be done without modifying the included services. But that is not the focus or goal of Web Services, and the level of complexity in the composition specifications, with, e.g., BPEL *partner link types*, means that substantial technical skills are required. Further, there is no notion of physical devices, which are important in our approach, and in pervasive computing in general.

Lately, approaches based on *mashups* of RESTful Web Services have also gained much attention. Mashups are generally made by external developers, not under control of the providers of the original services, for combin-

ing a set of services on a Web page. There is current research on how to make mashups easy to create for end users [67, 95].

As Web Services were originally designed for business-to-business communication, primarily in wired networks, it is not self-evident that they scale down to the resource-constrained devices involved in ubiquitous computing scenarios, or that they are suitable for use in networks where services join and leave frequently. BPEL has some support for dynamic modification of partner links by calling *assign* statements, imperatively, but there is no declarative way to specify dynamic partner links, as with connection declarations in a PalCom assembly. The WSAMI project [54] presents one approach to use of Web Services in a pervasive setting, and demonstrates a language and middleware for dynamic composition of Web Services, which executes on PDAs running the CDC configuration of Java ME [105]. One point where WSAMI differs from the situation targeted by PalCom is that it relies on Web Services descriptions being available in so called *universal repositories* in the network. PalCom cannot rely on such central directories, as will be discussed in the following chapter.

4.14 Summary

- A PalCom assembly is a customizable multi-connector wiring between services.
- An assembly stores a particular configuration of services on devices, and connections between them.
- Assemblies can handle configurations with fixed sets of services, and dynamic bindings support situations where services join or leave dynamically.
- An assembly stores a script used to mediate between services, enabling ad-hoc combinations of services that were not designed to work together.
- Assemblies enable end-users in the loop, without detailed programming needed for configuration and coordination.
- Unbound services can be programmed for involved computations where needed.
- An assembly can offer new services, so called synthesized services, and can be further aggregated by other assemblies.
- An assembly is executed by an assembly manager, which can reside on any suitable device.

Chapter 5

Communication protocols

The PalCom protocols are used both for announcement and discovery of devices, services and connections, as well as for interaction between services and assemblies. Their design has been made according to functional requirements drawn from the PalCom scenarios, as presented in Chapters 2–4. At the same time, once the protocols are defined, supporting them is the main requirement for being a PalCom device or service. This is important in order to enable independent implementations of the PalCom open architecture in different languages.

5.1 A layered model: overview

The PalCom communication architecture is designed as a layered architecture, as shown in Figure 5.1. There are five layers:

- At the lowest level, the **Media Abstraction Layer (MAL)** has the purpose to bridge between the PalCom communication model and the actual network technology used. This layer effectively hides all the implementation details of the used network from the upper layers. Figure 5.1 shows three implementations in this layer, for the Infrared, IP and Bluetooth communication technologies, respectively. The MAL Layer also handles the basic heartbeat mechanism that lets PalCom devices become aware of each other.

- For PalCom devices that support more than one media interface, there are some issues which are handled in the **Routing Layer**. If a receiver can be reached in more than one way, the choice of which media to use for communication is made here. The layer supports the implementation of tunnels, i.e., secure communication over hostile

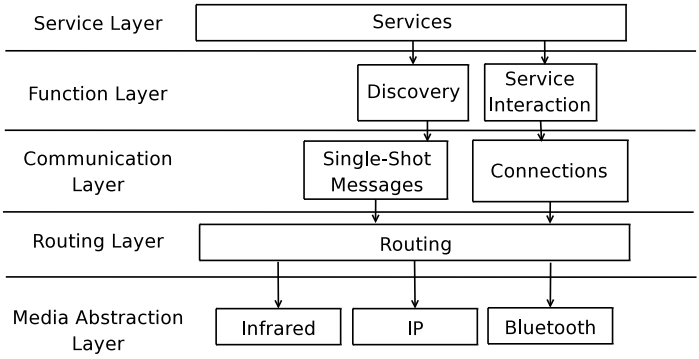


Figure 5.1: *The PalCom Communication Model.*

media between two or more local PalCom networks. It also handles messages that are too large to send in one go on a particular interface or network.

- In the **Communication Layer**, the common part of all PalCom communication is implemented. The layer supports both message-based communication and streams. There is support for communication between two endpoints (unicast), as well as radiocast and groupcast. There are also facilities for establishing stable connections, sending of single-shot messages, and reliable communication.

- The **Function Layer**, on top of the Communication Layer, implements two specific protocols. The *Discovery Protocol* is internal to the PalCom architecture, and supports the discovery of devices, services, and established connections. The other protocol in the Function Layer is the *Service Interaction Protocol*, used by services for exchanging messages. The message format of this protocol is defined, but the contents of the actual messages depends on the services involved.

- The **Service Layer** is the application layer, structured as services that use the communication architecture to communicate with other services. Services can be individually addressed, so at this level the model is that a service connects to another service, and then the two services can start exchanging messages.

5.2 Requirements

The requirements behind the PalCom communication protocols naturally have many dimensions. This section presents the main ones, as we have identified them. First, there are a number of general requirements for the communication between devices and services:

1. **Device awareness.** In addition to discovering and addressing services, users and programs should be able to discover *devices* with meaningful identities, and address them. As discussed in Section 2.2, the identities of the physical devices are important in pervasive systems.

2. **Names of devices and services.** Devices and services should have both human-readable names, and globally unique names that can be used for addressing and identification (Section 2.5).

3. **Communication with services.** There have to be mechanisms allowing services to communicate over unicast, radiocast and groupcast connections, using messages and streams (Section 2.5). Where needed, a service should be able to communicate over reliable connections, and to send long messages which are guaranteed to be delivered as a unit (Section 2.7). Further, for communication between services, the protocols have to be open, allowing special service protocols for situations with special requirements on encryption, bandwidth, etc.

4. **Efficient communication.** The format of messages sent to and from devices and services has to be compact, allowing small devices with limited bandwidth to offer services, e.g. over IR (Section 2.2). At the same time, we strive for a format which is convenient for debugging while messages arrive over the network.

5. **RemoteConnect.** A device should be able to connect a service on a second device to a service on a third device (Section 3.1).

6. **Routing between networking technologies.** There has to be support for communication with services using different networking technologies, with routing between technologies (Section 3.3).

7. **Versioning.** There have to be versions of services and assemblies, which are followed strictly when an assembly connects to services, and which allow a device to run more than one version of a service at the same time (Section 2.5).

8. **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 [107]), is not suitable, because that may leave the sending service in a blocked state if the network goes down (Section 2.7).

For the Discovery Protocol, there are a number of additional requirements:

9. **Discovery of connections.** In addition to discovery of devices and services, it should be possible to discover the currently established connections (Chapter 3).

10. **Descriptors.**

    (a) For devices, the descriptors that are announced in the Discovery Protocol have to contain name, addressing and versioning information (Chapter 2).

    (b) For services, there have to be name, addressing and versioning information, information about what *distribution* the service uses (i.e., if it communicates using unicast, radiocast or groupcast), and whether it uses reliable communication or not. There also have to be *service descriptions* describing the in- and outgoing commands (Chapter 2).

    (c) Together, the descriptors for devices and services have to describe the tree of services on the device (Chapter 2).

    (d) Descriptors of connections have to contain the identities of the two involved services, or the service and the group for a groupcast connection (Chapter 3).

11. **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, as will be discussed more in Section 5.7.1.

12. **Scalability.** The Discovery Protocol has to allow implementation on small devices, while scaling up to reasonably large numbers of devices (Section 3.4). Still, we want a human-readable descriptor format, for the same reason as for requirement 4 above.

13. **Advertisements, not queries.** It must not be necessary to specify the properties of a service in order to discover it, as discussed more in Section 5.7.1.

14. **No central directory.** The Discovery Protocol should not rely on a central lookup directory, because in the ad-hoc networks we target, such a central node cannot always be expected to be available. This is discussed more in Section 5.7.1.

5.3 The PalCom protocols

Three PalCom protocols have been defined, based on the requirements and the layered model. A central design decision, made in response to requirement 6 on page 75, *Routing between networking technologies*, is to let the PalCom protocols abstract from the protocols used for different network technologies, such as IP, Bluetooth and IR. This means that PalCom device and service descriptors can look the same, regardless of what kind of network the device is in. By addressing each other using PalCom addresses, which abstract from concrete addresses such as IP addresses, PalCom devices can communicate across network technologies in several hops, relying on PalCom routing functionality. For limited environments, the mapping to a concrete technology can be tailored specifically for resource constraints. The protocols are the following:

- The **Wire Protocol** abstracts from concrete network technologies, and defines the format of messages in the communication. It has been designed as a compromise between compactness, flexibility, and readability, and is based on a terse representation in readable characters. Message headers are structured as a set of nodes, where new nodes can be introduced with little or no changes to existing nodes. This supports a high level of flexibility for new demands, at the price of some parsing of the header information. The Wire Protocol also defines a heartbeat mechanism, known as the Pacemaker Protocol, which lets devices become aware of each other, as a basis for the upper-level Discovery Protocol. The Wire Protocol is handled by the lower layers in the communication architecture: the MAL Layer, the Routing Layer and the Communication Layer.

- The **Discovery Protocol** enables devices to find and get information about other devices, services and connections. This is a standard protocol, supported by all PalCom devices. At the heart of the discovery mechanism, in the heartbeats, broadcasting is used, but as soon as devices are in contact, information is transported using unicast, and on demand. The protocol is designed with embedded status information, which makes it safe to cache information, trusting that the originator will flag when the information is no longer valid.

- The **Service Interaction Protocol** is used after services have been discovered, for establishing connections between PalCom services, using the RemoteConnect sub-protocol, and for communication between services, by means of commands that can be sent and received. There is a standard format for command 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 (requirement 3 on page 75). 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 following sections will present each of the three protocols in turn.

5.4 The Wire Protocol

The Wire Protocol is handled in the three lower layers of the PalCom communication architecture, up to and including the Communication Layer (see Section 5.1). To upper layers, that layer offers functionality for sending and receiving messages over different types of connections, using a common PalCom message format, and access to the heartbeat mechanism that lets devices discover each other at a fundamental level.

There are two basic kinds of addressing information that make a PalCom message reach the right receiver: a *DeviceID* points out the device, and a *selector* points out the receiver within the device that will get the message. Such as receiver is typically a service, but can also be, e.g., a *discovery manager* that handles discovery centrally on a device. The Communication Layer presents addresses to upper layers in terms of DeviceIDs and selectors.

5.4.1 DeviceIDs

Each PalCom device is identified by a DeviceID, which is a unique identifier that remains stable. It represents the device itself, *rather than a particular network interface*, and is used both for addressing of the device, and for unique naming of the device and its services. A DeviceID can be formed in several ways:

- **Assigned names** are DeviceIDs that are assigned by a manual process, which makes sure that no two such names are given to two different devices. These names start with an 'A', followed by a suitable unique bit pattern, and are used in situations where it is important to be able to give devices short unique identities.
- **MAC names** start with a 'B', followed by the 48-bit binary address of a network interface card of the device.
- **UUID names** start with a 'C', followed by a 128-bit number, using the UUID standard [68].

Independently of the method used, the unique identities are guaranteed to be unique, so that a DeviceID created in this way always signifies one

and the same specific device. Because of their global uniqueness, they are generally quite long and non-intuitive. When written out in hexa-decimal, using ASCII characters, a MAC name occupies 13 bytes (1 byte for the 'B', and 12 bytes for the 48-bit address).

The DeviceIDs are independent of what communication medium a device uses. As discussed below, they are mapped to network-specific addresses in the MAL Layer, depending on the available communication interfaces. It is important that DeviceIDs identify devices, rather than network interfaces. This makes it possible to support PalCom routing in several hops, with seamless use of multiple networking technologies without affecting implementations at the service level. This is in contrast to IP networking, where devices that communicate in more than one network have separate, typically unrelated, IP addresses for its different network interfaces. In PalCom, it is up to the Routing Layer to decide which of several possible network paths to use when sending a message to a device, while the same DeviceID is used at the service level.

A related important point is that when a PalCom device moves to a new network, the DeviceID is kept and can be used for addressing the device the same way as before. It is not like in IP, where a new IP address is assigned and handling of the move is needed at the application level.

5.4.2 Selectors

Selectors are compact, local identifiers, allocated dynamically by a device for pointing out a receiver within the device. While the lower MAL and Routing layers handle the problem of delivering messages to the right device, based on DeviceIDs, the Communication Layer handles internal delivery of incoming messages, based on selectors.

The selectors put minimal requirements on the underlying network, in that they enable *multiplexing* over lower-level addresses, such as IP ports. In an IP network, it is possible for a PalCom device to receive all messages on the same IP port. The selectors in the PalCom messages, which are packaged inside the IP-level messages, are used for handling multiple receivers when messages arrive. This can be useful, e.g., in situations where all but a few IP ports are blocked, and, in general, for technologies where the underlying addressing mechanisms are restricted.

The Communication Layer provides a connection mechanism, which builds on selectors. Selectors are used both for establishment of connections, and for communication when connected. A service listens on one selector, and when a connection set-up message is received on that selector, the Communication Layer provides a new, dynamically allocated, selector for that particular new connection (much like how TCP handles ports). Selectors

for established connections are short-lived, and become invalid after a device reboots.

5.4.3 Heartbeats

The Wire Protocol defines a heartbeat protocol, which we have named the Pacemaker Protocol, with the purpose of letting devices on the network discover each other's addresses. This basic functionality is at the device level: when devices have become aware of each other, they can get information about each other's services and connections through further requests at the Discovery Protocol level.

According to requirement 14 on page 76, the discovery mechanism cannot be based on a central directory. Instead, in the heartbeat scheme devices *broadcast* information about themselves on the local network. Broadcasting facilities are assumed to be available in the underlying network technologies, facilities which are also used for the implementation of radiocast and groupcast communication. The Pacemaker Protocol works as follows:

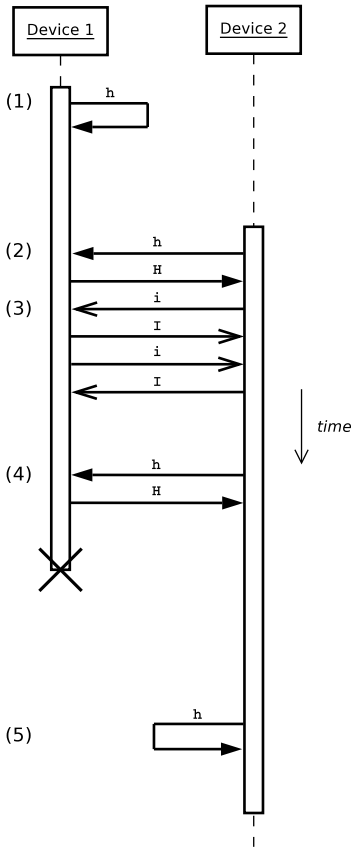
1. When a new device enters, it broadcasts a *HeartBeat* message containing its address. The address can be used by other devices for requesting more information about it.

2. When any device hears a HeartBeat from another, it broadcasts a *HeartBeatAck* with information about its own address.

3. When a device has not heard any HeartBeat within a time frame that exceeds its own upper limit for times between HeartBeats, it sends out a new HeartBeat.

This simple scheme makes discovery of new devices instant, because the new device makes a first HeartBeat. Discovery of device disappearances are also important, according to requirement 11 on page 76, *Transient devices*. For handling this, each device has a time limit of its own. When no HeartBeat or HeartBeatAck has been heard from a particular device within that time limit, that device is considered gone, and removed from caches. The most "eager" device (with the shortest time limit) will drive the heartbeat sequence for all the devices. All devices will *listen in* on broadcasted HeartBeats and HeartBeatAcks, irrespective of who triggered them.

The Pacemaker Protocol has been designed so the usage of network bandwidth is kept down, in order to meet requirement 12 on page 76, *Scalability*. The total number of broadcasts in every heartbeat period is equal to the number of devices, and the length of the period is controlled by the most eager device. In non-critical situations, devices can extend the period, for decreasing bandwidth usage.



(a) A heartbeat sequence.

IP address	DeviceID
10.0.0.5	C1E34A0...
10.0.0.12	C6E8291...
10.0.0.3	C0501AB...

(b) Mappings in the MAL Layer.

Figure 5.2: *An example heartbeat sequence is shown in (a). (1) At first, Device 1 broadcasts a HeartBeat* `h` *(broadcasts are shown with filled arrowheads). But, Device 1 is alone on the network, so the HeartBeat only reaches the device itself. (2) Next, Device 2 appears and immediately broadcasts a HeartBeat* `h`*. In response to that, Device 1 broadcasts a HeartBeatAck* `H`*. Now, both devices have each other's technology-specific addresses (IP addresses, or similar). (3) Following that, two pairs of* unicast *HBInfoRequests* `i` *and HBInfoReplies* `I` *give the devices each other's DeviceIDs. Mappings between technology-specific addresses and DeviceIDs are maintained in tables in the MAL layer, as illustrated in (b) for an IP example. (4) Device 2 has the fastest heartbeat rhythm, so after a while it sends out the next HeartBeat* `h`*, and Device 1 replies with a HeartBeatAck* `H`*. Both devices are now known, so nothing more happens as a result of the heartbeat. (5) Then, Device 1 goes down unexpectedly (shown with a cross). It does not respond to the next HeartBeat* `h` *from Device 2, and after a while Device 2 removes Device 1 from its cache.*

```
h;6;2564;G
```

(a) A HeartBeat.

```
H;5;532;G
```

(b) A HeartBeatAck.

Figure 5.3: *Heartbeat messages in the Pacemaker Protocol.*

It should also be mentioned that the timeout in the heartbeat scheme is not the only way for a device to disappear from caches. When a device is about to make an orderly shut down, it broadcasts a *HeartAttack* message. This enables other devices to promptly remove information about the device from its cache, or mark the device as inaccessible. Devices that do shut down without sending a HeartAttack message will eventually be removed as well, when other devices notice that they do not provide HeartBeat messages. This will be the case, e.g., when a device moves out of reach, or simply crashes.

In the implementation of the heartbeat mechanism, there is a space optimization, intended to help meet requirement 4 on page 75, *Efficient communication*. This optimization lets some of the addressing information be omitted from many PalCom messages. Conceptually, the DeviceIDs and selectors of both the sender and the receiver are present in every message. However, especially the DeviceIDs can be quite bulky, because of their global uniqueness. Therefore, DeviceIDs are translated to the primitive addressing mechanism of the underlying network as often as possible, while maintaining the guarantee of binding to the particular device.

This DeviceID optimization is handled through a mapping between DeviceIDs and technology-specific device addresses in the MAL Layer (in the IP case, the technology-specific address is an IP address-port combination). This mapping is established when a first HeartBeat or HeartBeatAck (which does not contain the DeviceID) is heard from a device. In the MAL Layer, the technology-specific sender address of the HeartBeat or HeartBeatAck is saved, and a unicast *HBInfoRequest* is sent to the device. An *HBInfoReply*, sent in return using unicast, contains the DeviceID (both HBInfo messages are unicast, because they should only be processed by the intended receiver, not by all devices). After this request-reply sequence, the DeviceID mapping is saved in the MAL Layer, and DeviceIDs do not have to be included in future messages sent to that device: for devices within the same local network, the network-specific address at a lower layer will suffice (see the next section for a presentation of how routing is handled). Above the MAL layer, the technology-specific addresses are not visible, and DeviceIDs are used as if they were transferred in every message. Figure 5.2 shows an example heartbeat sequence.

The HeartBeats and HeartBeatAcks themselves contain no DeviceIDs, only technology-specific addresses, which makes them very compact. This is

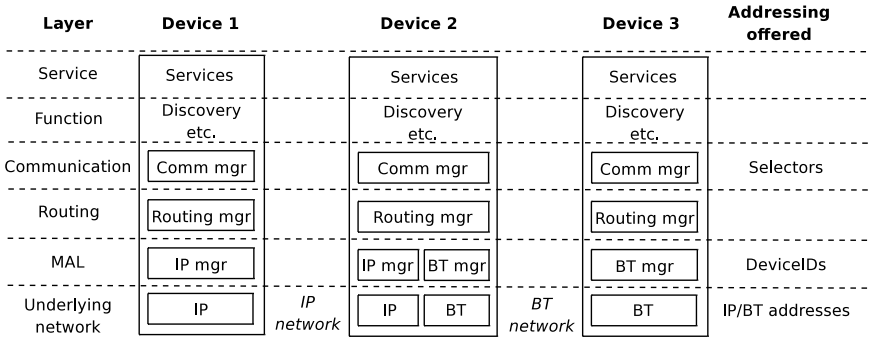


Figure 5.4: *A schematic view of three PalCom devices, communicating in IP and Bluetooth networks. The right-most column shows what addressing mechanisms are offered by the respective layers: above the MAL layer, no medium-specific addresses are used. Device 2 is on both networks. The other devices are only on one network each, but can communicate by addressing each other using DeviceIDs and selectors. The communication is routed through Device 2.*

important, because HeartBeats and HeartBeatAcks are sent repeatedly also when no new devices appear, and should put as little load as possible on the communication channel. Figure 5.3 shows examples of the contents of heartbeat messages (see Section 5.4.6 for an explanation of the PalCom wire format). The size of the two messages are 10 and 9 bytes, respectively. All addressing information is in technology-specific message headers, so there is no addressing overhead in the heartbeats. The information in the example heartbeats are that *device cache numbers* are 2564 and 532, and that the device status is *G*, for green (discussed below in Section 5.5.3). The direct representation of heartbeats at this level, in the Wire Protocol, is motivated by the need for very compact communication in some situations, where the heartbeats can be tailored for specific technologies.

5.4.4 Routing between networking technologies

The optimization of omitting device addresses from PalCom messages, as described above, works within one local network, where a technology-specific address can be used for addressing the receiver directly. For communication between networks, there are additional mechanisms in the protocols. As illustrated in Figure 5.4, some PalCom devices (Device 2 in the figure) sit on more than one network, with one MAL layer manager for each. These devices are referred to as PalCom *routers*, and communication is routed through them. PalCom DeviceIDs and selectors span across net-

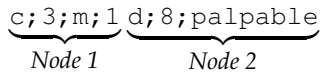


Figure 5.5: *A non-routed messages over a connection.*

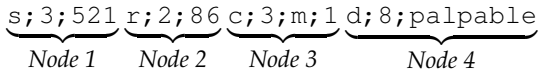


Figure 5.6: *A routed message over a connection.*

works, while technology-specific addresses are assumed to be valid only within one local network.

For messages going to a PalCom device that is not on the same local network, the local device cannot send directly to the technology-specific address of the receiver. Instead, the message goes to a PalCom router that knows how to reach the receiver on some other network. As a secondary optimization at this level, PalCom *ShortIDs* are used instead of sending DeviceIDs in all messages. ShortIDs are names that are assigned by routers for referring to PalCom devices on remote networks (the limited scope, devices known by one router, makes them more compact than DeviceIDs). Routers perform translations of ShortIDs as messages pass through them.

The initial distribution of ShortIDs is done in the heartbeat phase. HeartBeats and HeartBeatAcks are routed, and as they pass a router, the router attaches the sender's ShortID to them. When a device receives a ShortID in a routed HeartBeat, it looks up the corresponding DeviceID through an HBInfoRequest, which is sent to the originating device, routed back through the router that forwarded the HeartBeat. An HBInfoReply is returned, which contains the DeviceID, and the DeviceID can be exposed to upper layers. In continued communication with the device, the ShortID is put as receiver device address in messages. It should be noted that this mechanism works also across more than one router, for multiple hops.

Figures 5.5 and 5.6 show examples of messages sent over a connection, without and with ShortIDs as routing information. The node structure of the messages is shown, as explained below in Section 5.4.6. The difference between the contents of the messages are the two nodes added at the beginning of the message in Figure 5.6 (13 bytes in total). The ShortIDs for the sender and the receiver of that message are 521 and 86, respectively.

In summary, the DeviceID optimization and the mechanism of ShortIDs mean that DeviceIDs are only transferred once per pair of communicating devices. PalCom-specific device addressing information is only needed in routed messages, in the form of compact ShortIDs.

5.4.5 Connections

The Communication Layer gives support for unicast, radiocast and groupcast connections:

- **Unicast** (one-to-one) connections build on selectors. When a service on one device wants to connect to a service on another, the Communication Layer reserves a local selector and sends a connection establishment message to a *listening* selector on the other device. The Communication Layer on the other device reserves a new selector for the communication, and sends a reply containing that. On both sides, the layer then presents a connection interface to upper layers, which hides the handling of selectors and lets messages flow in both directions.

- **Radiocast** (one-to-many) connections are uni-directional: data flows from a sender on one device to multiple receivers, and the sender is not aware of the exact identities of the receivers. This is implemented in the Communication Layer, on top of a broadcast mechanism provided by the MAL Layer. A sender is identified by a DeviceID and a selector, where the selector points out the sender within the device. No connection establishment messages are exchanged for radiocast connections: the Communication Layer at the receiver simply starts to listen for broadcasts that have been packaged as radiocasts from a specific sender.

- **Groupcast** (many-to-many) connections do not involve selectors. For these connections, where all members of a group can send and all can receive, there is a *GroupID* that identifies the group. When the Communication Layer receives a broadcasted message, packaged as a groupcast and intended for a group to which the local device has one or more connections, the message is delivered up to those connections. There are two types of groupcast connections: one which is local to one network, and one which works across routers. For the former, devices can just start sending to the group and receive from it, but for the latter there are connection-establishment messages, which let routers keep track of group memberships in the networks they are in. This way, groupcast messages can be forwarded between networks when needed.

5.4.6 Message formats

PalCom defines a general wire format for messages, as exemplified in Figures 5.3, 5.5 and 5.6. This format is used for packaging of messages in the Discovery Protocol, and for messages between services, except for those

services that use their own, private message formats (those special services implement their communication against the network layer directly, as discussed in Section 5.6). For each type of underlying network, the MAL layer implementation handles how PalCom wire format messages are packaged in technology-specific messages (such as UDP packets for IP).

A wire format message is structured as a set of *message nodes*, concatenated after each other. The general format of a message node is the following:

```
MessageNode ::= <F>;<L>;<d>
```

Here, `<F>` is a one-byte format identifier, `<L>` is the length of `<d>`, and `<d>` is the contents of the rest of the node, whose format depends on the value of the format byte. Semicolons separate the three parts of a message node.[1] The format byte is an ASCII printable character, usually a letter, and the length is written in decimal, with ASCII characters. This means that the whole node can be printed in readable characters, if the data is readable, which makes inspection and debugging easier, and helps meeting the PalCom visibility challenge (see Section 1.4). The node structure also gives flexibility, because, if the protocol changes, it is possible to introduce new nodes with little or no changes to existing nodes, in contrast to fixed-width header formats. When a message goes down through the layers on its way out from a device, nodes are added by lower layers at the beginning of the message. At the receiving device, the nodes are processed from the beginning, as the message goes up through the layers. Appendix A lists the different message node types.

There are message nodes for addressing (selectors), to be used for messages sent over connections (unicast, radiocast and groupcast) and for single-shot messages. The actual data in a message is packaged in a data node. In Figure 5.5, node 1 indicates that it is a message over a unicast connection, to selector 1 on the receiver device (`c` is for connections, and `m;1` is the `<d>` part of the node, which is in turn semicolon-separated, with `m` meaning a message over a unicast connection and 1 being the selector). Node 2 contains the data in the message (format byte `d`): the 8 bytes of the word `palpable`.

There are additional nodes for routing. In Figure 5.6, node 1 contains the ShortID of the sender (format byte `s`), and node 2 the ShortID of the receiver (`r`). The rest of the message is independent of the routing, and similar to the non-routed message.

The message format also permits hierarchically structured messages, with messages inside messages at any number of levels. There is a format byte '+'

[1] Calculating the `<L>` involves taking the length of `<d>`, but not including the `<F>;<L>;` parts in the length. This means that the reader of a message needs to parse the `<F>;<L>;` parts and then add the `<L>` to the current position in order to get just past the current message node (and thus to the start of the next message node).

for *multi-part* messages, which can be used for grouping a set of data (d) nodes, possibly mixed with inner multi-part-nodes. This can be used by upper layers as a mechanism for structuring data, or for grouping a number of related pieces of data to be sent in one message to the same receiver, as an optimization that decreases the network overhead. An example of structuring using multi-part nodes is the packaging of commands in the Service Interaction Protocol (see Section 5.6 for an example). Regarding this hierarchical structure, and the human readability, an obvious choice would have been to base the wire format directly on XML. We have not done that for the basic message format, though, because we need a more compact format, and binary data in the messages has to be permitted. However, the message formats for discovery and service descriptions, sent as data inside wire format messages, are XML-based.

5.4.7 Large messages and reliable delivery

The message format supports transfer of large messages and reliable message delivery, handled in the Routing and Communication layers. There are special message nodes for these mechanisms at the PalCom level, which means that PalCom services can communicate reliably on top of unreliable networks, and send messages that are larger than the maximum transmission unit of the underlying network. As an example of this, the implementation for IP in the MAL Layer uses UDP, and does not require TCP. UDP has an upper packet size limit that can be as low as 8K, and there are no guarantees for packet delivery. Using the PalCom mechanisms gives a uniform handling, without TCP-specific overheads for IP.

The nodes supporting large messages let a message be chopped up into numbered small pieces, which are put together at the receiver side. When the message has been completely received, it is forwarded to the layer above, or it is all dropped. The nodes for reliable communication put sequence numbers on messages, which let the Communication Layer on the receiver side ask for re-sends if messages arrive out-of-order, or report failure if a message fails to arrive. For detecting dropped messages, the Communication Layer keeps track of a timeout for each message during reliable communication.

5.5 The Discovery Protocol

On top of the heartbeat mechanism in the Wire Protocol, the Discovery Protocol defines how PalCom devices can describe themselves and their services and connections, and discover other devices, services and connections. Section B.1 in Appendix B contains the grammar of the descriptors, addresses and messages used in the Discovery Protocol.

5.5.1 Service naming

As given by requirement 2 on page 75, *Names of devices and services*, two different kinds of device names are distributed in the Discovery Protocol. There are unique names in the form of DeviceIDs, used for identifying devices unambiguously, and there are human-readable names, which are shown to users. The short device names, ShortIDs, are not dealt with at the Discovery Protocol level. They are hidden by lower layers.

Services also have human-readable and unique names, which are distributed in the Discovery Protocol. The unique names, which identify services unambiguously, are called *ServiceIDs* and *ServiceInstanceIDs*. These are used for making sure that connections are made to the same device and the same service, in the same version from one time to another.

A service can be created on one device and later deployed, cloned, to many other devices. In some situations it is interesting to know that two services have exactly the same interface. For this reason it is necessary to have an identity of a service that is independent of the device it is residing on. On the other hand, when addressing a service, the identity of the device it resides on is important. Furthermore, particular services provided by assemblies, or unbound services, can be updated many times on different devices after being deployed. We need to represent the version of a service so the relations between such modifications can be understood and managed. Our approach is to let versioning information be part of the unique service names.

A ServiceID is an aggregated, unique identifier of a version of a service. Part of the ServiceID stays stable when a service is moved or cloned to another PalCom device. ServiceIDs are structured so they can be used for constructing the *version tree* of a service: when an update is made to a service on one device, the DeviceID of the updating device is included in the new ServiceID. A ServiceID is put together as follows:

```
ServiceID ::= CreatingDeviceID CreationNbr
              UpdatingDeviceID UpdateNbr
              PreviousDeviceID PreviousNbr
```

- *CreatingDeviceID* is the DeviceID of the device where this service was initially created. *CreationNbr* is a number that, together with the CreatingDeviceID, uniquely identifies the service even if there are several services created on the same device.
- *UpdatingDeviceID* is the DeviceID of the device where this service was last updated. *UpdateNbr* is a number that uniquely identifies this version of the service, together with UpdatingDeviceID, chosen such that newer versions of the service have higher numbers, within the device.

- *PreviousDeviceID* is the UpdatingDeviceID of the previous version of the service. *PreviousNbr* is a number that uniquely identifies the previous version of the service, together with PreviousDeviceID.

The six parts of the ServiceID identify a version of the implementation of the service. The first two parts will stay the same even when the service is updated and forms new versions. All six parts will be the same on whatever device this particular version of the service is instantiated.

A ServiceInstanceID identifies an *instance* of a service. There can be several instances of the same version of a service, provided simultaneously by the same device. An example of this is when an unbound service is started twice or more on the same device. ServiceInstanceIDs are managed by the device where the service is instantiated. After an instance has been started the first time, the ServiceInstanceID can be used unambiguously for referring to that instance in the future:

```
ServiceInstanceID ::= [DeviceID] ServiceID
                      [InstNbr]
```

- The DeviceID identifies the device on which the service runs. This is optional, as indicated by the brackets, because the DeviceID may be left out in situations where the device is given by the context, (e.g., in the `services` section of assembly descriptors, as explained below in Chapter 6). Conceptually, a DeviceID is part of every ServiceInstanceID.
- The ServiceID identifies the version of the service.
- *InstNbr* is a string that distinguishes otherwise similar instances of a service on a device. InstNbr is optional: when there is only one instance of a service on a device, the ServiceInstanceID does not need to contain an InstNbr part.

When establishing a connection, a ServiceInstanceID is translated into a selector, which is used for connecting to the service. Selectors to be used for connecting to services are communicated as part of the Discovery Protocol. ServiceInstanceIDs are also stored in assembly descriptors. They make it possible to safely change to a service on another device, knowing that it provides exactly the same interface. ServiceIDs are used when updating to a new version of an assembly.

In the same way as for devices, services also have short names. The short names for services are called *LocalServiceIDs*, and these *are* handled in the Discovery Protocol. A LocalServiceID maps one-to-one with a ServiceInstanceID on a given device. It can be more compact, because, in contrast

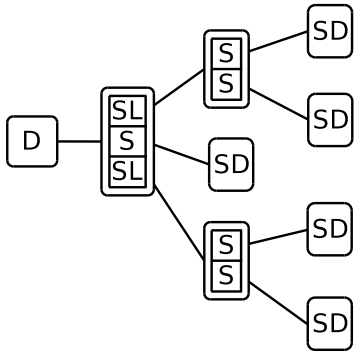


Figure 5.7: *The logical tree of information about services on a device.*

to a ServiceInstanceID, it only needs to be unique within the device. When a service is updated, its ServiceInstanceID is changed, and the service is given a new LocalServiceID. It is possible that the device will continue to offer several versions of a service in parallel.

There is a request/reply pair of messages in the Discovery Protocol, used for fetching the LocalServiceID of a service, given its ServiceInstanceID. In a similar way, there is one pair of messages for looking up a ServiceInstanceID from a LocalServiceID, and one pair for fetching the current selector the service listens on, given a ServiceInstanceID. These messages are used for translating names directly, without looking them up in the service descriptors, that are announced as presented in the following section. The possibility of direct lookup is useful, e.g., when establishing a connection to a service whose ServiceInstanceID is in an assembly descriptor.

5.5.2 Descriptors

The Discovery Protocol distributes descriptors of devices, services and connections, as given by requirement 10 on page 76, *Descriptors*. The information about services on a device forms a tree, as illustrated in Figure 5.7. These descriptors are the following:

- **PRDDevice**[2] (marked with D in the figure) holds information about the device, including the human-readable and the unique name.

- **PRDServiceList** describes a list of services (shown as a box, with S and SL inside). It can have the following elements:

[2]PRD stands for PalCom Resource Descriptor.

  - **PRDService** (marked with S) contains name and addressing information about a service, including the human-readable and unique names. Each PRDService has a service description attached.
  - **PRDSubList** (marked with SL) is a placeholder for a child list in a PRDServiceList. A PRDSubList has a PRDServiceList below it. In the PRDSubList there is an attribute *kind*, which indicates if the services in the child list are ordinary services, services provided by an assembly, or unbound services.

- A **service description** (marked with SD) contains the in- and outgoing commands for a service. The commands have zero or more parameters, and can be grouped hierarchically within the service description.

For connections, the Discovery Protocol contains the following descriptors:

- **PRDConnectionList** is a list with information about the currently established connections that a device has initiated.
- **PRDConnection** is a descriptor for one connection in a PRDConnectionList.

Connections of different topologies are described in slightly different ways when announced in the Discovery Protocol:

- **One-to-one** connections are described by the identities of the two involved services.
- For **radiocast** connections, each service that connects to the sending service announces a new descriptor, containing the sender's identity and its own.
- When a service makes a **groupcast** connection, it announces in a descriptor that it is connected to the *group*, which has an identifier known by the involved services.

The descriptors follow an XML format, which is human-readable. Figures 5.8 and 5.9 show descriptors for the loudspeakers in the music scenario of Chapter 2. For the PRDDevice, it can be noted that the DeviceID is not part of the XML. It is obtained in the earlier heartbeat phase, and having it in the XML would make the PRDDevice heavier. For similar reasons, ServiceInstanceIDs are not in the PRDServices. Instead, they are looked up from the LocalServiceIDs, which are in the descriptors. Examples of XML for service descriptions, with commands, parameters and streaming information, were shown in Figures 2.7–2.9.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<D name="Loudspeakers" v="0.1" ds="G" rcs="5" />
```

Figure 5.8: *The PRDDevice (`D`) for the Loudspeakers device, as shown in the tree of Figure 2.6. `name` is the human-readable name of the device, `v` is the device version, `ds` is the current device status, and `rcs` is a selector used in the RemoteConnect protocol (see Section 5.6).*

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<SL>
  <S name="Audio in" ls="3" d="1" p="P1" r="0"
     vn="1" h="In-going audio stream" />
  <S name="Volume control" ls="2" d="1" p="P1"
     r="0" vn="1" h="Loudspeaker sound volume" />
</SL>
```

Figure 5.9: *The PRDServiceList (`SL`) for the two services of the Loudspeakers device in Figure 2.6. The descriptor contains two PRDService elements (`S`). `ls` is the LocalServiceID of a service, `d` indicates the type of distribution (unicast, radiocast or groupcast), and `p` says what version of the Discovery Protocol the services follow (allowing graceful upgrading). `r` being zero means that the services do not communicate using reliable communication. `vn` is a human-readable name of the version of a service. `h` is a help text.*

At each point in time, a device holds a set of descriptors about itself and its services and connections. This information is distributed to other devices by means of the Discovery Protocol. For efficiency reasons (requirement 12, *Scalability*), the whole set is not sent out in one big message each time. This is because

(1) not all devices are interested in all the information (e.g., when visualizing a number of discovered devices in a browser, the user might only be interested in inspecting the services of some of them), and

(2) when the state of a service or connection changes, it is inefficient to send all descriptors each time.

Especially the service descriptions are envisioned to sometimes be quite bulky. Therefore, the descriptors are distributed in separate messages, and there are request messages for requesting exactly the information a device needs. These requests, and the corresponding replies, are *unicast*. A caching scheme is used for optimizing this communication, as discussed in Section 5.5.5.

5.5.3 Status information

The Discovery Protocol supports distribution of status information about devices and services. This information can be shown for discovered entities in browsers, helping users when interacting with services or combining them in assemblies. The main status information is one byte with one of three values:

- `R` means "not operational" (red).
- `Y` means "partially operational" (yellow).
- `G` means "fully operational" (green).

This information can, e.g., be visualized as traffic lights for devices and services. For services, the status byte is accompanied by *StatusHelpText*, a text field, possibly empty, that explains the status further. Exactly how the status should be interpreted depends on the device and service itself. It is mainly intended as a source of feedback to a human user when things are not working as expected. The StatusHelpText can be used to further explain for an interested user what the source of the problem is. Taking a digital camera as an example, one way of using the status mechanism could be:

- `R`, with the text "No flash card available", when there is no flash card.

- `Y`, with the text "Flash card full", when the flash card is full.
- `G`, with no text, when everything is fine.

The device status is sent out in every heartbeat from the device, as shown in Figure 5.3 (where the status is `G`). For obtaining status information for a service, there are special request and reply messages in the Discovery Protocol. As for the descriptor requests and replies, the caching scheme is used for minimizing this traffic.

5.5.4 Versioning

The support for versioning of services is built into the naming scheme, as discussed above in Section 5.5.1. When a service is updated, the DeviceID of the device where the update is made is put into the new ServiceID, in a way that makes it possible to reconstruct a (partial) version tree from a number of available versions of a service.

There is versioning support also for devices. This is not part of the DeviceIDs, which stay stable, but there is a device version field in the PRDDevice, which uniquely defines a version of the "platform" of a device. When the device version is the same, assembly managers and services can trust that the underlying software on the device is the same, including the device firmware and the implementation of PalCom managers. The device version changes if manager implementations change, or when the hardware or underlying software changes.

5.5.5 Caching

The basic functionality in the Discovery Protocol builds on the light-weight heartbeat mechanism, defined in the Wire Protocol. A second aspect that makes the Discovery Protocol bandwidth-efficient is the use of caching. The purpose of this caching is to minimize the unicast request-reply traffic, and the basic idea is that when some of a device's descriptors have changed, it should be possible to see that by looking at the device's *cache number*, which is part of the heartbeat. If the cache number is unchanged, compared to the latest saved cache number from that device, all descriptors are unchanged and cached information can be used. For small devices, with a fixed set of services, and which initiate no connections that need to be announced, this means that at most one sequence of requests will be received from each device. After that, the other devices will see that the cache number is unchanged, and not ask again.

As a refinement of the caching scheme, there are also a number of more fine-grained cache numbers, indicating what kind of information has changed when the device cache number changed. The purpose of this is not

having to request many types of descriptors, when only one type has changed. There are cache numbers for services, connections, selectors, service status and device version. As an example, the set of connections typically changes more frequently than the set of services on a device, and the service descriptions might be bulky, so it is a useful optimization to look at the connections cache number and request only connection descriptors.

5.5.6 Small devices

As mentioned above, it is important that the overhead inflicted by the Discovery Protocol is as small as possible for resource-constrained devices. Such a device would be one that simply offers one or a few services, does not initiate any connections and does not dynamically start any assemblies or unbound services. The small device can take further advantage of the request-reply scheme in the protocol: it will never initiate a discovery procedure, because it is itself not interested in replies. It can be reactive and answer HeartBeats and requests with simple, fixed answers, and it can ignore the HeartBeatAcks from other devices. The small device does not need to keep track of a heartbeat period, which simplifies the implementation of the Discovery Protocol and reduces runtime overhead.

5.6 The Service Interaction Protocol

Once services have been discovered using the Discovery Protocol, they can be connected and start communicating. At this level, there is a PalCom protocol called the Service Interaction Protocol. According to requirement 3 on page 75, *Communication with services*, the communication between services must, however, be open and not restricted for services that have special demands. These demands may, e.g., be that a set of services require encryption according to a certain scheme, or that data is compressed in a special way. Therefore, the Service Interaction Protocol is not a standard protocol in the same way as the Discovery Protocol, in the sense that all PalCom devices have to follow it. Instead, it is the *default* protocol used for communication between services, used when services do not announce in their descriptors that they use another, custom protocol.

In the Service Interaction Protocol, communication takes place in an asynchronous manner, in the form of commands sent over connections. There is a wire format for packaging the commands with their parameter values. This format builds on the multi-part message format, defined in the Wire Protocol (see Section 5.4.6), and on the XML format of commands (`CmdI` elements, as shown in Figure 2.7). The format is constructed so that messages for commands with textual parameter values consist entirely of human-readable characters. For commands with binary parameter values,

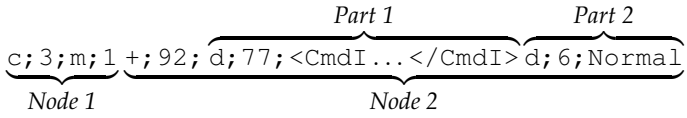


Figure 5.10: *A command packaged in a multi-part message.*

the parameters are transmitted in binary. Figure 5.10 shows a command, as sent out over a connection from the loudspeakers' volume control service, when the sound volume has been changed to *Normal* (see Figure 2.2). Node 1 contains addressing information. Node 2 is the multi-part node, which has two parts: part 1 is a data node that contains the XML descriptor of the command, and part 2 contains the value of the single parameter. If there had been more parameters in the command, more parts would have followed.[3]

The Service Interaction Protocol also has a sub-protocol, called RemoteConnect, whose use precedes the command communication. RemoteConnect is used for connecting two services from a third device, typically from a PalCom browser or by an executing assembly. After the services have been connected, they can communicate. The protocol consists of two unicast messages: RemoteConnect and RemoteDisconnect. The RemoteConnect message asks a device to establish a connection between one of its services and another service, and RemoteDisconnect asks it to close a previously opened connection.

5.7 Related work

The PalCom communication and discovery protocols have been designed according to the requirements presented in Section 5.2. Protocols developed in other research projects, or for other technologies, are all designed for more or less different requirements, and therefore we cannot compare all aspects of the PalCom protocols with any of them. As an example, DEAPspace [77] focuses on single-hop short-range wireless systems, where low power consumption is crucial, while Jini [121] targets larger and more fixed networks, where devices can be more powerful. In this section, we will compare with solutions for different issues or subproblems that have been used in related protocols.

PalCom is not tied to one particular network technology, but allows support for different technologies in the MAL Layer, which are abstracted through the use of addressing with DeviceIDs and selectors, and a common message format. This abstraction is done in order to support devices

[3] In Figure 5.10, 92 is the length of the contents of node 2. This is the sum of the lengths of part 1 (5 bytes + 77 bytes of XML data) and part 2 (4 bytes + the 6 bytes of the word *Normal*).

with limited capabilities in restricted environments, where messages can be tailored for specific requirements. Several other protocols are instead based solely on the Internet Protocol (IP), which PalCom handles in one of the MAL Layer implementations. One example of an IP-based protocol is UPnP [114], which has zero-configuration assignment of IP addresses, and which uses HTTP-based protocols on top of that, with variants of HTTP for multicast UDP, unicast UDP, and TCP. Zeroconf [99] is also based on IP, with similar zero-configuration address assignment, and DNS-based protocols for handling naming and discovery. Web technologies are based on HTTP and IP.

Haggle [100] is an architecture that aims to separate application logic from the underlying networking technology, thereby making it more adaptive and flexible—with seamless connectivity—for example in a situation without a central network infrastructure. A key idea is *just-in-time binding*, which means that decisions about protocols and lower-level addresses are deferred as late as possible, so they can be switched when conditions change.

PalCom has routing between technologies within local networks, and tunnels between networks. UPnP relies on network-based multicast for discovery, which means that devices cannot discover each other over a wide-area network. This issue has been tackled for UPnP in research approaches based on proxies and gateways [76, 116].

The rest of this comparison with other protocols will be structured according to the two problems of

(1) **discovery**, how a service can be discovered and contacted, in order to communicate with it, and

(2) **service interaction**, how communication with services is done.

5.7.1 Discovery

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 [77, 18], some by software vendors as part of operating systems, such as Jini, UPnP, and Zeroconf, and some by various consortia, such as Salutation [96], 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 [125].

Some technologies, such as Bluetooth, have discovery mechanisms that are closely tied to the lower layers in their respective protocol stacks. For sup-

porting multiple technologies in PalCom, one option could be to delegate to the native discovery protocols, and let the MAL Layer present the results to upper layers. This approach has been chosen by Haggle [100] and also by Obje [81], which abstracts each discovery protocol as an aggregate component. In PalCom, we have chosen to design a generic PalCom Discovery Protocol, which relies on unicast and broadcast communication from the underlying technology. But, if necessary, the choice of using native discovery protocols is still open, when building a MAL Layer implementation.

The following discussion of different discovery protocols will take PalCom design choices as its starting point.

PalCom is device aware

The importance of *devices* in PalCom has been emphasized in previous chapters, and devices are announced as stand-alone entities in the Discovery Protocol. Other projects argue for announcing services only. In Jini, this is referred to as *device agnosticism*, meaning that hardware and software can be treated in a unified way by clients [31]. Zeroconf has a similar principle, and argue that discovering the service is what matters, because it is the service that handles the actual communication [99]. For a pervasive computing system like PalCom, our stand-point is that the devices should be visible, as a way of guiding the user. In addition, pervasive services are localized (as opposed to, e.g., Web services), and people can be expected to care more about which of several devices is used. 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 for landscape architects, biomonitor sensors for emergency personnel [65], enhanced incubators at the hospital, the Stone device supporting pregnant women [119], etc. In some of 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,[4] 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 [114].

PalCom uses advertisements, not queries

Another point that differs between protocols is how the discovery process is initiated. The starting point in many protocols is that the user searches

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

for services, by formulating some form of query. In PalCom, the scenario is rather that the user browses for services on nearby devices. Jini is one example of a query-based approach. In Jini, a discovery request takes the form of asking a lookup service for an object which implements a particular Java language type, or which has certain attribute values [121]. Attributes for specifying queries are also used by SLP [53]. In UPnP and Bluetooth, searches can be made for all devices, or for specific device or service types.

A potential advantage of query-based approaches is that the query, which is quite compact, can be broadcasted, while replies with device and service information, which are generally more bulky, can be sent in unicast to the requesting device, and need not be processed by all devices. This keeps the load on the network down. Still, we have not built a query mechanism in the PalCom Discovery Protocol. The main reason for this is the problem of formulating the query. Using Java types, or attribute values, puts us in a situation similar to the domain-specific standardization discussed in Chapter 1. This is a risk also for systems based on semantic ontologies, such as GSD and DReggie [18, 19]. Secondly, relying on queries means that the connection to the physical devices in your vicinity, as advocated above, is easily lost, because the focus is on services, not devices.

PalCom uses 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 [121] and Salutation [96] 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, and for clients to go directly to the service for interacting with it. This approach is taken, e.g., by UPnP [114]. There are also hybrid approaches. Example are VSD [60], where more powerful devices can volunteer to serve as directories, when it is needed, and One.world, which uses a dynamic election mechanism [43].

Based 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 [77], 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. This view is also supported by Shenoi et al. [97], who write that centralized registry-based architectures are mainly for stable wired infrastructures.

PalCom supports transient devices

It is in the 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 often wireless, with varying signal strengths, which means that the protocols must be prepared to handle dropouts in the communication, and that replies to messages cannot always be expected. For discovery protocols that target pervasive computing environments, 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 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. This problem is called *partial failures*, one of the issues that makes distributed systems more difficult to handle [31, 59]. There is always a risk that one device goes down in an unclean way, without having the time to inform the other devices. This could, e.g., be due to a software crash, power loss, or physical damage. From the other devices, it can be hard to detect this, because it may look like a device 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 some of the partial failure problems. This is a kind of garbage collection: all services, that want to announce themselves, lease a place in a lookup directory for a limited time. If the service crashes and the lease is not renewed regularly, the resource in the directory will be removed. This keeps old resources from filling up memory on directory devices, and gives a form of self-healing to the system. But it does not solve the problem of services that join and leave frequently. For as long as the timeout of a certain lease, the Jini directory will provide proxies for the service to clients. If the service crashes, lookup directories will distribute out-dated proxies, and clients will not notice that the service does not respond, until they try to communicate with it. This reflects the fact that Jini is mainly targeted for larger, fixed networks, not ad-hoc networks.

UPnP is not designed for rapidly changing networks either, but mainly for more static set-ups. UPnP uses periodic advertisements. In [114, p. 14], 30 minutes is given as the minimum time for the duration of an advertisement. Such a long duration keeps the load on the network down, because not so many advertisements are sent, but we argue it is much too long for a pervasive setting.

Zeroconf [99], on the other hand, has a very drastic philosophy: there are no periodic idle packets at all in the mDNS discovery protocol, on which

Zeroconf builds [21, Section 9.3]. There is no polling or heartbeat at all for detecting partial failures. Instead, the failure or crash of one device is detected next time any device tries to communicate with it, and then that information is spread in the network. Unless anyone tries to use a device, it can remain indefinitely in users' lists of discovered devices. The motivation is to keep the load on the network down.

PalCom chooses a middle way, with a very compact heartbeat at the Wire Protocol level. As devices listen in on broadcasts, each device has to broadcast only once per period, and it is the most eager device that controls the period. In less critical situations, the period can be increased.

DEAPspace [77], which targets single-hop short-range wireless systems, presents *proactive service discovery*, in which each member device sends out the full list of known services, not just its own. The goal is to discover services faster as they appear, while keeping the load on the network down. In DEAPspace, devices cancel and reschedule broadcasts when a broadcast is heard from another device, an idea which has some similarities with the idea of listening in on broadcasts.

Techniques for discovery efficiency

PalCom's way of combining compact heartbeats with unicast requests for more detailed information, in case a device is interested in it, is somewhat similar to the split between *discovery* and *description* in UPnP [114]: in the discovery step, only a few essential specifics about a UPnP device or service is distributed, and interested devices retrieve descriptions in the description step.

The second main performance measure in the PalCom Discovery Protocol is the caching and the *cache number* flags. Techniques based on caching are used in other protocols as well. Zeroconf [99] has the techniques of Known Answer Suppression, Duplicate Question Suppression, Duplicate Answer Suppression, and Opportunistic Caching, which are all ways for devices to avoid network communication by utilizing their caches in different ways. GSD [18] uses peer-to-peer caching of advertisements, and group-based intelligent forwarding of service requests, in order to reduce the network traffic. Chapter 10 will present quantitative measurements, for evaluation of the scalability of the PalCom protocols and the reference implementation.

5.7.2 Service interaction

After discovery of services, the second phase is the actual communication with a service. Not all of the protocols we have looked at have specifications at this level. E.g., the Zeroconf protocols end where the IP host and

port of a service is transmitted to the discovering party: after that, the format of the communication is completely service-type specific. Examples of protocols that have specifications at this level are Jini, for which the communication is done through method calls on a proxy object, and UPnP, which has SOAP communication according to an XML service description.

PalCom does not rely on mobile code

PalCom does not make use of mobile code in the Service Interaction Protocol. Instead, the protocol specifies an XML format for commands, which are packaged in PalCom message nodes and sent over connections. One reason is that we want to allow implementations of the protocols on different platforms, and not tie the protocols to a specific platform, like Jini does in practice with the dependence on the JVM. An independent implementation of the PalCom protocols has been demonstrated in the master's thesis [72]. Our view on mobile code is supported by [43, p. 28], where experiences from the One.world project are discussed. The author writes, in retrospect, that One.world would have been better off with a data-centric model, such as XML Schema, rather than expressing and distributing data schemas in the form of code. Reasons for this are that data is easier than code to distribute and share. It is not tied to a specific execution platform, and thus gives better interoperability. Compatibility with Internet protocols is also mentioned as an important factor.

PalCom has asynchronous communication

Some protocols use service interaction mechanisms that are based on, or related to, *Remote Procedure Calls* (RPC). In RPC, a client executes a procedure (method) on a service remotely. The RPC system packages the procedure call with parameter values and sends it to the service over the network. The method executes in the service, and a return value is sent back over the network. Meanwhile, the client is blocked until the return value arrives. One protocol that is based on RPC is Jini with RMI [102]. UPnP uses SOAP for communication with services, which is generally used in an RPC way, and Web Services described in WSDL also use SOAP for their communication.

One problem with RPC is that the client gets blocked during the call, and if there are network errors, or a device disappears, the client remains blocked. In PalCom, we have instead based the Service Interaction Protocol on asynchronous communication, where commands are sent without waiting for replies. If a service wants to send back a response to a command, the response is sent in a separate command. This, we argue, is a model that fits much better in unreliable networks.

A related question is who initiates the communication. With RPC, we have a client/server model, where the client always takes the initiative. In Jini

and UPnP, there are also eventing mechanisms, which let a client subscribe to events from a service, and when something happens the event is sent out to subscribers. In PalCom, we have chosen to unify this, and use the same command mechanism for communication initiated by the service. There is no separate eventing mechanism, but *pull* and *push* are handled the same way. This fits with the peer-to-peer relationship between devices normally involved in PalCom scenarios.

5.8 Summary

Adhering to the PalCom protocols is what defines a PalCom device or service. The Wire Protocol

- abstracts from different network technologies, by defining a common wire format and by using PalCom addresses that abstract from technology-specific addresses.
- uses an extendable, node-based message format.
- contains the Pacemaker Protocol, which
  - adjusts its heartbeat frequency to the needs of the most eager device.
  - detects non-available devices within the needed time delay.
- supports connections, single-shot messages, reliable communication, large messages and different distribution schemes (unicast, radiocast and groupcast), as needed by PalCom services.
- gives compact support for routing between networks, in multiple hops and between different network technologies. The use of DeviceIDs, rather than network interface addresses, supports transparent routing and mobility of devices between networks, without affecting service level implementations.

The Discovery Protocol

- distributes names, addresses and descriptors of devices and services, including human-readable names and unique identifiers that are independent of the network technology used.
- supports a strict versioning scheme, with explicit version control for services.
- distributes status information about devices and services.

- uses a caching scheme, where detection of changes supports efficient updates of cached information.
- is partitioned in a way that lets small devices implement only a small, reactive part of the protocol.

Further, the Discovery Protocol

- keeps PalCom device aware by announcing devices at the top level.
- uses no central directory.
- is based on advertisements, not queries.
- supports transient devices that join and leave networks frequently.
- does not rely on mobile code.
- uses asynchronous communication.

The Service Interaction Protocol

- supports command communication between PalCom services.
- contains the RemoteConnect sub-protocol, which lets pairs of services be connected from a third device.

Chapter 6

The language of assemblies

An assembly sits between a set of services on devices, and can be seen as a customizable, multi-connector wiring between them. In Chapter 4, it was discussed how the structure of assemblies supports a separation of configuration, coordination and computation. This chapter will present the language of assembly descriptors, and how those three aspects are expressed in different parts of the assembly. Further, different representation formats of the descriptors will be discussed, followed by a look at the tasks an assembly manager performs when executing an assembly, and mechanisms for updating and versioning of assemblies.

6.1 Configuration

The configuration aspect of an assembly concerns what devices and services are included, and how they are connected. The parts of an assembly descriptor that have to do with this are the lists of devices, services and connections, flexible bindings to devices and services, and the interface of synthesized services.

The assembly descriptor language is small: its abstract syntax fits on two pages in Section B.2 of Appendix B. Yet, it is sufficiently powerful for many interesting applications. In this section, we will first use the Uploader assembly, which was introduced in Chapter 4, to illustrate the basic features of the language. The task of Uploader is to send tagged images, produced by the TaggingCamera assembly, to the photo database on the server at the office. The assembly descriptor for Uploader is shown in Figure 6.1.

```
1  assembly Uploader 1.0 released {
2    this = ServiceID;
3    devices {
4      Handheld = DeviceID;
5      Server = DeviceID;
6    }
7    services {
8      CameraSynth = ServiceInstanceID on Handheld;
9      PhotoDB = ServiceInstanceID on Server;
10   }
11   connections {
12     CameraSynth <-> this;
13     PhotoDB <-> this;
14   }
15   script {
16     eventhandler {
17       when taggedPicture from CameraSynth {
18         send storePicture(thisevent.img)
19           to PhotoDB;
20       }
21     }
22   }
23 }
```

Figure 6.1: *The Uploader assembly. The descriptor is shown in* concrete syntax*, with keywords in bold (see Section 6.4 for information about different presentation formats of assembly descriptors).*

6.1.1 Name and versioning information

The name of the assembly is given on line 1 in the Uploader assembly descriptor (Figure 6.1). Furthermore, its human-readable version number 1.0 is shown, and its status is shown as `released`. This means that version 1.0 of the assembly is released, i.e., copies of it can occur elsewhere, and if this assembly is further edited, the editing tools should automatically update the version number. See Section 6.6 for more information about versioning of assemblies.

On line 2, a more elaborate version number of the assembly is given, using the syntax "**`this`** `=` *`ServiceID`*". The ServiceID is structured as described for service identifiers in Chapter 5. This way, the assembly gets a version identifier that is guaranteed to be unique, according to the scheme for ServiceIDs.

6.1.2 Devices and services

Devices and services are identified by globally unique names, as discussed in Chapter 5. These names have an internal structure including versioning information, and typically they are quite long, and not intended to be very readable to a human. In the Uploader example in Figure 6.1, we simply display them as non-terminals: *`DeviceID`* and *`ServiceInstanceID`*. 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 versioning information in the names is used by tools to make safe upgrades of an assembly when a service or a device has been upgraded.

On lines 3–6 in Uploader, the devices involved in the assembly are listed. For each device, a *local name* and the DeviceID are given. The local names (`Handheld` and `Server`) are identifiers relevant only inside the assembly. The assembly author is free to use any identifier, as long as all local names for the devices are unique within the assembly. When editing assemblies in an editor, the editor can suggest the local name based on the human-readable name of a device, as currently discovered in the browser at editing-time. The name can be changed by the author, typically for better showing the purpose of a device in the specific assembly. The name used for actually identifying a device is the DeviceID. The editing tool can automatically retrieve also this name and insert it automatically into the assembly descriptor.

On lines 7–10 the services are listed, and these names are handled in the same way as for devices. Two different devices, e.g., two projectors, can have (different instances of) the same service on them. To uniquely identify a service instance on a particular device, ServiceInstanceIDs are used. The service declarations also state which devices the services execute on,

using an explicit "**on** *LocalDeviceName*" syntax. This facilitates assembly rebinding (see Section 6.6).

There are certain static-semantic constraints on how assemblies may be constructed, which can be checked by an editor while an assembly is being edited. This static consistency checking boils down to common name- and type checking rules similar to those in simple programming languages:

- That local names are unique.
- That used local names are declared.
- That device names are not used where service names are expected, or vice versa.

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.

Even if a device is available, 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 updated so that a device declaration is changed to another device, the new device does not have all the declared services. Tools can flag these situations as errors or warnings, and guide the user in trying to change to another service on the same device, or possibly to a service on another device.

6.1.3 Connections

The `connections` section, lines 11–14 in Uploader, lists how pairs of services are connected to each other during execution of the assembly. The assembly is here viewed as a service, denoted by `this`. More specifically, `this` denotes one of possibly several *implicit services*, as discussed in Section 4.2. The implicit services mark assembly-side end-points of connections between services and the assembly. Given a connection declaration *s1* `<->` *s2*, commands can flow in both directions between *s1* and *s2*.

In the Uploader example, all the listed services are connected to the assembly (`this`), but it is also possible for an assembly to connect one listed service directly to another. The RemoteSlideShow assembly, as presented in Chapter 4, is an example of such an assembly. It is shown in Figure 6.2. Also for this type of connection declarations, local device and service names are used.

There are semantic constraints that can be checked only dynamically, when activating the assembly. If the services of a connection are available, it can

```
assembly RemoteSlideShow 1.0 released {
  this = ServiceID;
  devices {
    Projector = DeviceID;
    Laptop = DeviceID;
    Phone = DeviceID;
  }
  services {
    Control = ServiceInstanceID on Laptop;
    Slides = ServiceInstanceID on Laptop;
    UIDisplay = ServiceInstanceID on Phone;
    Screen = ServiceInstanceID on Projector;
  }
  connections {
    Slides <-> Screen;
    Control <-> UIDisplay;
  }
}
```

Figure 6.2: *The RemoteSlideShow assembly in concrete syntax.*

be checked that the connection is well formed: the service descriptions of the involved services can be checked, making sure that the commands or streams match, regarding names and MIME types. For the streaming connection `Slides <-> Screen` in Figure 6.2, it is checked that the MIME types of both the involved services is `image/jpeg`, and that they have opposite streaming directions. The `Control <-> UIDisplay` connection is OK, because UIDisplay is a meta service that can be connected to any control service.

6.1.4 Bindings

Bindings that specify *alternative* services is one way to prepare an assembly for a situation where the set of available services varies. Alternatives are specified as prioritized lists in the `services` section. When running the assembly, it will bind to the top priority service if it is available, and if not, to the next service on the priority list, etc. In the following example, the assembly defines two alternative GPS services, giving the first priority over the second:

```
devices {
  MainGPS = DeviceID;
  BackupGPS = DeviceID;
  Compass = DeviceID;
```

```
  Camera = DeviceID;
  Handheld = DeviceID;
}
services {
  Position = {
    ServiceInstanceID on MainGPS priority 1,
    ServiceInstanceID on BackupGPS priority 2
  }
  ...;
}
```

When the assembly is running, the `Position` service may be rebound on the fly: Suppose that initially both MainGPS and BackupGPS are available. `Position` will then be bound to the service on MainGPS, due to priority. If MainGPS fails or disappears while the assembly is executing, `Position` will automatically be rebound to the service on BackupGPS (if still available). If the MainGPS becomes operational again, `Position` will again be rebound to the service on MainGPS, and so on.

It is assumed that MainGPS and BackupGPS both have the same service protocol for their services. This will typically be the case if the two GPSs are two devices of the same model. However, if the two GPSs are completely different, say from two different manufacturers, they might well have different service protocols. In that case, it is possible to create an assembly which adapts one of the GPSs. It can use a synthesized service to act as a proxy, and an event handler to translate the messages as required.

The other type of dynamic binding defines whether a service is *mandatory* or *optional*. This is indicated by marking mandatory services using the keyword `mandatory`:

```
services {
  ...
  Direction = ServiceInstanceID on Compass;
  mandatory Photo = ServiceInstanceID on Camera;
  mandatory Storage = ServiceInstanceID on Camera;
  mandatory CoordinateStuffer
    = ServiceInstanceID on Handheld;
}
```

In this example, the assembly author has considered the Photo and Storage services on the camera, and CoordinateStuffer on the handheld, as mandatory for the functionality of the assembly. The way to define a service as optional is to leave out the `mandatory` keyword, such as for Direction in the example. Leaving all services as optional means that the assembly will work on a best-effort basis, using all services that are available.

6.1.5 The interface of synthesized services

The TaggingCamera assembly, whose deployment diagram was shown in Figure 4.5 and which was listed in Figure 4.7, provides a synthesized service. Through the synthesized service, a user or assembly can tell the assembly to take a picture, and also retrieve the latest tagged picture. The interface of the synthesized service is declared on lines 32–35:

```
32      synth CameraSynth {
33        in  takePicture();
34        out taggedPicture(image/jpeg img);
35      }
```

The `synth` clause declares a name, CameraSynth, and defines in-going and out-going commands. If a command has parameters, those are also declared. In this example, the command `taggedPicture` has a parameter called `img` of the MIME type `image/jpeg`. The definition of how the synthesized service works is part of the coordination aspect, which will be discussed in the following section.

Typically, a synthesized service provides an aggregated interface to a set of composed services, such as in the CameraSynth example. It is, however, perfectly possible and useful to provide synthesized services for assemblies with only one or even zero constituent services: an assembly containing a single constituent service can be used as a wrapper or adapter of that service. An assembly with zero constituent services is useful for building simulated services in a development situation: the synthesized service of the assembly can be made the same as a not yet developed real service, and used in the development of other assemblies.

Synthesized services can be defined to participate in groupcast or radiocast communication, as discussed in Chapter 3. In the following example, from the Active Surfaces puzzle game assembly which will be presented in Chapter 9 (Figure 9.12), the keyword `groupcast` is used for indicating that the service participates in a group named *HappyGroup*:

```
38      synth HappySynth groupcast HappyGroup {
39        in out challenge();
40        in out veto();
41      }
```

In the declaration of a groupcast service, the commands have both directions `in` and `out`, because the service receives all commands sent to the group, and can send any command as well.

For a synthesized service using radiocast communication (defined using the keyword `radiocast`), all commands are either `out`, for a sending service, or `in`, for a receiving service.

6.2 Coordination

The coordination aspect of the assembly is where additional functionality is added, compared to when a set of services are connected directly, as defined by the configuration. The coordination is in the script and its event handling clauses, where commands can be sent or forwarded to services, and values can be saved in variables.

6.2.1 Script and event handling clauses

The optional `script` section of an assembly descriptor, lines 15–22 in Uploader (Figure 6.1 on page 106), defines the automated behavior of the assembly. This is done in terms of a set of event handling clauses, in this case only one:

```
17        when taggedPicture from CameraSynth {
18          send storePicture(thisevent.img)
19            to PhotoDB;
20        }
```

This clause states that when the assembly receives the command `taggedPicture` from the CameraSynth service, a new command `storePicture` is to be sent to the PhotoDB service. The example illustrates how the parameter `img` of the `taggedPicture` command is passed on as a parameter of the `storePicture` command. The parameter is accessed through the `thisevent` notation, referring to the currently received command, i.e., the `taggedPicture` command in this case. Generally, in an event handler clauses are written as

```
when command from service {
  actions
}
```

The actions can access data in the command, send new commands to other services, and perform assignments of local variables. Event handlers are *atomic*: received commands are handled in incoming order, one command at a time. The assembly manager queues incoming commands, and lets each event handler finish before it continues with the following command. This atomicity means that there is no need for concurrency mechanisms, such as mutual exclusion primitives, in the language for event handlers. Incoming commands for which there is no event handler clause are simply ignored.

There are some static-semantic constraints for event handlers that can be checked by an editor, e.g. there can only be one event handler clause for each command from each service. In principle, certain information in the

assembly descriptor could have been inferred from the event handlers: this includes the declaration of connections between the assembly and services, and the declaration of the synthesized services. We have chosen not to infer such information, but instead keep it explicit in the declaration sections. This makes the separation of configuration, coordination and computation clearer, and it allows the script to be treated like a black box by tools that are not interested in the internal logic of the script, but only interested in the interface of the assembly (its synthesized services) and the overall communication structure (its connections). If desired, it would be easy to build a specialized assembly editor that does not require the user to add this information, but where the tool can automatically infer it from the script.

6.2.2 Variables

Assembly scripts can keep state in variables. An example is the TaggingCamera assembly, where the latest GPS coordinate and compass direction are saved, every time they arrive. The script part of TaggingCamera, and an example sequence of messages, was presented in Figure 4.7 and Section 4.3. The `variables` section contains variable declarations, declaring variables of the `text/plain` MIME type :

```
7     variables {
8       text/plain latestCoord;
9       text/plain latestDirection;
10    }
```

Two of the event handlers save parameter values in the variables:

```
12      when coord from Position {
13        latestCoord = thisevent.value;
14      }
15      when dir from Direction {
16        latestDirection = thisevent.value;
17      }
```

The values saved in the variables are used later, when an image should be sent to CoordinateStuffer for tagging.

6.2.3 Synthesized services

Part of the definition of a synthesized service, the definition of its functionality, belongs to the coordination aspect. The assembly can handle an incoming command arriving to its synthesized service in an ordinary event

handling clause. Looking at the TaggingCamera assembly again:

```
28        when takePicture from CameraSynth {
29          send takePhoto() to Photo;
30        }
```

Here, the event handler clause takes care of a command `takePicture`, that has arrived at the synthesized service. Further, the assembly can invoke (i.e., generate) new outgoing commands from the synthesized service:

```
25        when imageTagged from CoordinateStuffer {
26          invoke taggedPicture(thisevent.img);
27        }
```

This creates a new command `taggedPicture` which is sent to all assemblies and/or services that are connected to the CameraSynth service. It has been utilized here that there is only one synthesized service. If there were more than one, the message could be invoked as follows, pointing out a specific synthesized service:

```
invoke taggedPicture(thisevent.img) of CameraSynth;
```

6.2.4 A loopback mechanism

It might be the case that the assembly is located on the same device as some of the included services. For obtaining good performance in this case, 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 commands to go out unnecessarily on the network. An example of this is the unbound service CoordinateStuffer, which is used by the TaggingCamera assembly, and which sends and receives potentially heavy JPEG images. The loopback mechanism is used also if the assembly connects two services on the same device: the network is transparent, and commands 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, share hardware and communicate with each other directly. 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.

6.3 Computation

The computational aspect of a PalCom system is expressed in the services included in an assembly. These services may be native services (tied to the hardware on included devices), unbound services (built for being included in assemblies), or synthesized services of other assemblies. This means that there are no computation constructs in the language for event handlers, which helps keeping the assembly descriptor language simple. The possibility to include specially constructed unbound services enables solutions to specific "algorithmic" problems in assemblies, such as tagging images with coordinates.

6.4 Representations of an assembly

It is useful to discuss assemblies from several different perspectives: the end user, the expert user, the tools manipulating the assembly, etc. There are different *representation formats* of an assembly descriptor, relevant for these perspectives:

1. An **XML representation** that is used for storing the assembly descriptor in a file system or on a storage service, and for copying the assembly descriptor from one device to another.

2. An **attributed abstract syntax tree** (AST) that is used internally by tools accessing and manipulating the assembly. The XML (and also the concrete syntax discussed below) can be generated from the AST representation by a simple unparse operation. The AST representation can likewise easily be constructed by parsing the XML description. The AST contains an API in the form of attributes that makes it easy for the tools to access its information. Examples of attributes are links from uses to definitions of identifiers, semantic checking operations, etc. The AST representation can also be used for keeping run-time state while executing an assembly, as will be discussed in the following section.

Based on the representation formats, assembly descriptors can be *presented* in different ways for reading or editing. In addition to a presentation of the raw XML data, some of the possible presentation formats are:

(a) A **concrete, textual syntax** that is used to show the same level of detail as in the XML representation, but in a more concise readable form. The concrete syntax is used in this chapter. With tool support, it could also be used by an expert user for creating or editing assemblies. End users would typically prefer a more graphical syntax.

(b) **Tool-specific** editing representations for displaying parts of the assembly information to end users, typically in a graphical or semi-graphical way. For example, the PalCom Overview Browser displays the connections between services as lines between boxes (see Chapter 7 for a presentation of the different browsers). The PalCom Developer's Browser displays the assembly descriptor as a hierarchical expand/collapse listing and supports drag-and-drop based editing.

6.5 Execution of assemblies

Assembly descriptors are handled by assembly managers. The assembly manager stores descriptors, and can load an assembly from a descriptor and start executing it. The manager's tasks during execution are the following:

- **Announce the assembly** on the network. This is done by announcing implicit services for connections declared to the assembly.[1] The announcement makes the running assembly visible on the network. It is announced as running on the device where the assembly manager runs, which means that the running assembly has a well-defined location. In Figures 4.5 and 4.6, the implicit services are illustrated as gray service symbols without hooks. They show the structure of the assembly, and are announced even if the corresponding services on other devices are currently not available. Connections to implicit services can only be initiated by the assembly manager, not from the outside using the RemoteConnect protocol (see Section 5.6).

- Announce any declared **synthesized services**.

- Using PalCom discovery, start **monitoring the network** for devices and services listed in the `devices` and `services` section. As soon as a listed connection is possible to establish, try to establish it. This can be through a direct connection request, if the connection is to `this`, or through a RemoteConnect request, if the connection is between two external services.

- Handle **dynamic bindings**, with alternative, mandatory and optional services as discussed in Section 6.1.4.

- If the assembly has a **script**, handle **incoming commands**, arriving to the assembly over listed connections or to synthesized services.

[1] Implicit services are grouped under a PRDSubList showing the assembly name, with the *kind* attribute specifying that the services belong to an assembly (see Section 5.5.2 for information about PRDSubLists). Synthesized services are announced under the same PRDSubList.

Execute event handlers by

- **sending commands** to services, over listed connections (`send`) or to services connected to a synthesized service (`invoke`), and
- saving parameter values in **variables**.

Execution of assemblies involves discovery of devices and services, announcement of services, and communication with services using commands. Therefore, an assembly manager uses the PalCom Discovery and Service Interaction protocols, as described in Chapter 5. In other words, it relies on an implementation of the PalCom communication architecture, up to and including the Function Layer (see Figure 5.1). Beyond that, assemblies do not introduce additional protocol layers for their execution: the services generated by the assembly, called synthesized services, are discovered directly, not the assembly per se. The only assembly-specific information announced is that assembly managers use a particular value of the *kind* attribute of the PRDSubList that groups its services. Assembly descriptors are kept locally, and interpreted by the assembly manager during execution.

For the distribution and update of assembly descriptors, between assembly managers on different devices, there is, however, a separate protocol, as discussed in the following section. That protocol is defined at the PalCom service level.

6.6 Updating and versioning of assemblies

In addition to execution of one version of an assembly, the structure of assembly descriptors also supports strictly version-controlled updating of assemblies. This is a key factor in making assemblies useful as a flexible mechanism for combination of services. The versioning supports that an assembly is made in one context, for one set of devices and services, then adapted to another context and tested there, and then *released* as a new version for the new context. The assembly does not just run in the new context, but has to be adapted and tested before release: the bindings in the assembly descriptor (mandatory, optional and alternative) are to *specific* devices and services in specific versions, pointed out by DeviceIDs and ServiceInstanceIDs. Thus, the user of a released assembly can trust that it has been tested in the relevant context.

The possibility of adapting assemblies to a local situation means that it is not necessary to write the assembly from scratch each time, but useful assemblies can be distributed and reused by others. Assembly descriptors are envisioned to be updated much more frequently than services, because they are easier to change and adapt.

Assembly editors in PalCom browsers can support rebinding of device and service declarations to currently discovered devices and services. E.g., this can be done in a situation similar to when Bill developed the TaggingCamera assembly in the GeoTagger scenario (see Chapter 4). If he had first created the TaggingCamera assembly for his own camera, tested that it worked, and then brought the assembly to Mark, he could have rebound it to Mark's camera through a simple drag-and-drop operation in the browser. The effect of that operation would depend on how similar the cameras were:

- If Mark's camera was from the same manufacturer, it would likely have identical services, and the browser could rebind the services automatically or semi-automatically (by suggesting to the user). After that, the assembly could be tested directly.
- If, on the other hand, Mark's camera was of a different make, the adaptation would include manual rebinding of services, and probably updating of event handlers in the script, before the assembly could be tested.

The changes made to an assembly descriptor, when a new version of an assembly is released, are made on the first two lines of the assembly descriptor (see Figures 6.1 and 6.2): on line 1, the human-readable version number is updated and the keyword `released` is added, and on line 2, the ServiceID is updated. The update of the ServiceID follows a scheme that makes it possible to recreate partial version trees when new versions of an assembly are obtained:

- The *CreatingDeviceID* and *CreationNbr* parts are unchanged.
- *UpdatingDeviceID* is set to the DeviceID of the device where the update is made, on which the browser and editor runs. *UpdateNbr* is selected so there is no conflict with other updates of the assembly on that device, and so it is higher than for earlier versions created on that device.
- *PreviousDeviceID* and *PreviousNbr* are set to the values of *UpdatingDeviceID* and *UpdateNbr* in the old ServiceInstanceID.

The distribution of assembly descriptors between assembly managers on different devices may happen in different ways. Files can be copied manually and installed, or downloaded from the Internet. In PalCom, we have also worked on a more automatic, *epidemic* scheme. Assembly managers have a protocol for this, defined at the PalCom service level. The assembly managers use groupcast communication for coming in contact with each

other, and then ask about each other's installed assemblies through command communication. If other versions of an installed assembly are available, an assembly manager uses the structure of the ServiceInstanceIDs for rebuilding a partial version tree. If it ascertains another available version to be newer, the manager requests the descriptor from the other device, installs it and informs the user. The user can choose to use the version he or she prefers.

In relation to the distribution of assembly descriptors, it should be mentioned that there is nothing in the assembly descriptor that ties it to execution on a particular device. So, as discussed in Section 4.7, it is possible to move the assembly descriptor to another device and start the same assembly there. This can be useful for performance optimizations: if the assembly can run on the same device as a service that produces large messages, those messages need not go over the network. On the other hand, if extensive calculations are needed, by means of an unbound service, it can be moved to a more powerful computer without change.

6.7 Related work

BPEL [80] is a language for workflow specification, orchestration, of a set of Web services. Its domain is interaction between business partners on the Web, which is different from our domain of composition of pervasive services. BPEL targets developers, and not end users, while PalCom targets end users for manipulation of assembly descriptors, as discussed in Section 4.12. Still, it is relevant to compare with BPEL, because the role of a BPEL orchestration can be compared to the role of the assembly, as discussed in Chapter 4.

The BPEL language is considerably more complex than the language for PalCom assemblies. There are constructs for parallel activities, with synchronization between them. There are `if` statements for conditional execution, loops, variables with both simple and complex types, and complex expressions. A BPEL process has an execution state, on which the next action depends, whereas PalCom assemblies have a more state-less model, with a number of event handlers that are all constantly enabled. The only state in a PalCom assembly is in the values of its variables. An advantage of PalCom's approach is that we have non-blocking communication, which fits with unreliable communication in the pervasive computing setting.

For some of the more complex behaviour that may be needed in an assembly, we rely on the delegation to an unbound service, as exemplified by the CoordinateStuffer in the GeoTagger scenario. The expressiveness of the language of assemblies has sufficed in the scenarios we have worked with in PalCom. We are, however, open for adding some more constructs to our language, which may be variants of the ones in BPEL. Our goal is to

strike the balance of finding a simple and light-weight mechanism for ad-hoc combinations of services, which is still sufficiently powerful. In [78], Emma Nilsson-Nyman has applied PalCom assemblies to health care scenarios, and suggested language extensions for synchronization, states and instances. See Chapter 11 for a discussion about future work on the language of assemblies.

An approach that is closer to traditional programming languages is AmbientTalk [27]. AmbientTalk is an attempt to incorporate abstractions handling the special characteristics of mobile networks into a programming language. There is non-blocking communication, discovery, and adaptation to varying connectivity supported directly in the language, as an alternative to middleware support. The focus of AmbientTalk is, however, different than PalCom's, in that a full, object-oriented language is proposed.

6.8 Summary

Assembly descriptors, as presented in this chapter,

- support specification of configuration, coordination and computation aspects of an assembly.
- can handle configurations with fixed services, as well as dynamic bindings for situations with where services join or leave dynamically.
- can have a script part with event handlers for coordination, which are executed as atomic units, in response to asynchronous events.
- can define interfaces of synthesized services.
- can be executed on any device having an assembly manager, irrespective of the location of the services it connects to.
- can be moved to another assembly manager and execute there, with little or no modification.
- supports controlled, versioned updating of assemblies.

Chapter 7

Browsers

The browser is a key tool in a PalCom system. It lets the user discover devices and services, connect services to each other, and control them through migrated user interfaces. More advanced browsers can execute assemblies, and some also have assembly editing functionality. The style of user interaction in a browser may differ, depending on the target user group. In the PalCom project, we have developed a number of different browsers, which are part of the reference implementation.

7.1 The Handheld Browser

There is an initial, prototypical browser, referred to as the *Handheld Browser*. It has been developed by the author of this thesis. Discovered devices and services are presented in a tree view, and there is functionality for discovering existing connections, for establishing new connections, and for controlling services through remote user interfaces. The user interaction in the Handheld Browser is mostly menu-based and could be considered rather primitive, because it is designed to fit on the small screen of a handheld computer.

Figures 7.1 through 7.3 show screenshots of the three tabs in the tab-based GUI that make up the Handheld Browser. The Devices tab in Figure 7.1 contains a tree view of discovered devices and services. In addition to the handheld computer where the browser runs, an MP3 player and a set of loudspeakers have been discovered. In Figure 7.2, the Connections tab is shown, which lists currently established connections. There is one connection between the Audio out service on the MP3 player and the Audio in service of the loudspeakers, and one between Track selection on the loudspeakers and UI Display on the handheld computer. Figure 7.3, finally, shows a user interface that has been migrated from the loudspeakers and

Figure 7.1: *The Devices tab of the Handheld Browser.*

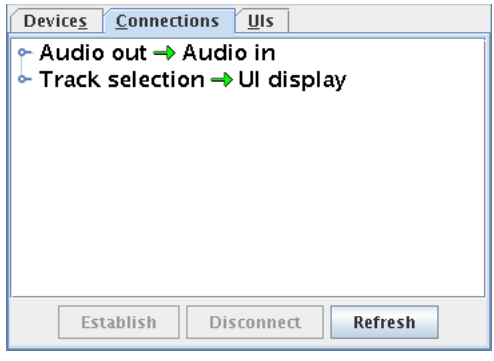


Figure 7.2: *The Connections tab of the Handheld Browser.*

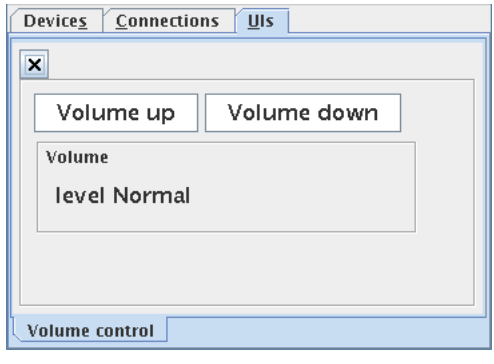


Figure 7.3: *The UI tab of the Handheld Browser.*

shown in the UI tab. The user can control the loudspeakers by clicking buttons in the user interface.

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 MP3 player device directly to the loudspeakers, using drag-and-drop on the Devices tab, without opening up to see what services the devices have. In this case, there will be exactly one matching service pair, and this pair—the `audio/mpeg` sender of the MP3 player and the `audio/mpeg` receiver of the loudspeakers—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.

The Handheld Browser can display user interfaces for PalCom services. When a connection to a service is established, a service description for the service is fetched from the *discovery manager* on the device, which handles the Discovery Protocol. A user interface is created, according to the structure of the service description, by means of an available user interface library on the device. The Handheld Browser uses Swing [106], but it could also be, e.g., MIDP [103] or SWT [30]. The user interface is tied to the service description. When the user performs actions in the user interface, parameter values are filled in and commands are sent to the service. When the service sends out commands, the user interface is updated accordingly. The discovery manager, and other middleware managers, will be presented further in Chapter 8.

The current browser implementation creates buttons for in-going commands, with text fields for parameters of type *text*. For out-going commands, text parameter values are shown in labels, and image parameters (type JPEG) are shown as images in the user interface. Groups of commands are rendered as nested panels. This simple support has worked for the needs we have had for services in PalCom scenarios, but support for more data types, and other ways of entering input values, could naturally be useful.

7.2 The PalCom Overview Browser

A second browser is the *PalCom Overview Browser* [87], implemented at the University of Aarhus. The Overview Browser visualizes discovered devices, services and assemblies as hierarchical boxes in a window, showing lines between boxes for discovered connections. This way, it gives a graphical overview of a running PalCom system. Figure 7.4 shows a screenshot of the Overview Browser in operation. With its graphical visualization, the Overview Browser is intended for devices with larger screens, such as laptops.

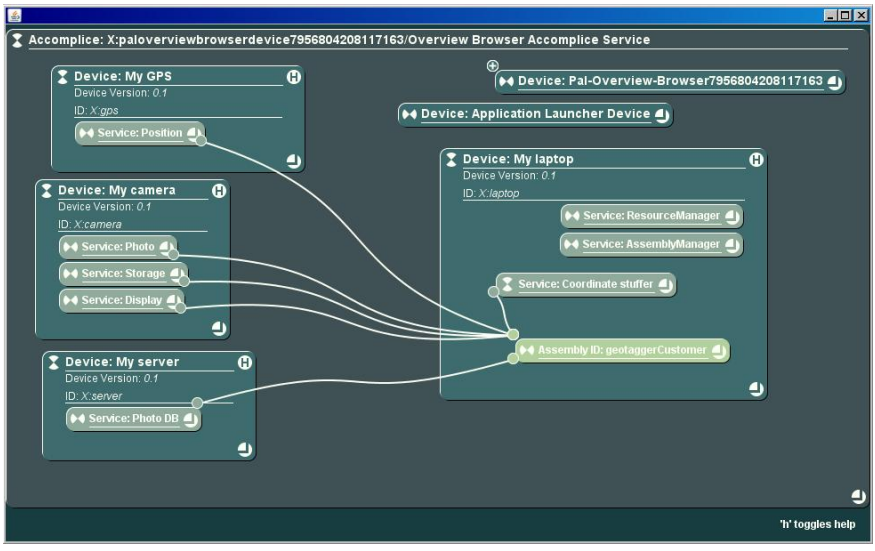


Figure 7.4: *Screenshot of the PalCom Overview Browser.*

7.3 The PalCom Developer's Browser

The third browser in the reference implementation is the *PalCom Developer's Browser*, developed by Sven Gestegård Robertz at Lund University. The Developer's Browser targets developers of PalCom assemblies, and is built as a plug-in for the Eclipse IDE [29]. It has an integrated assembly manager for executing assemblies, and an editor for assembly descriptors. There is a tree view for browsing currently available devices, services and connections, and remote views for interacting with services through migrated user interfaces.

The Developer's Browser has all the functionality of the Handheld Browser, and moreover it handles assembly editing and execution. Figure 7.5 shows a screenshot of the Developer's Browser. On the left in the browser window is the Browser View, which presents a tree view of discovered devices, services and connections. On the right is the assembly editor, where an assembly descriptor is being edited.

The Developer's Browser is an Eclipse plug-in, and can be run inside the IDE, but it can also be run as a stand-alone application in Windows, Mac OS X or Linux. For a user's guide to the Developer's Browser, see [87].

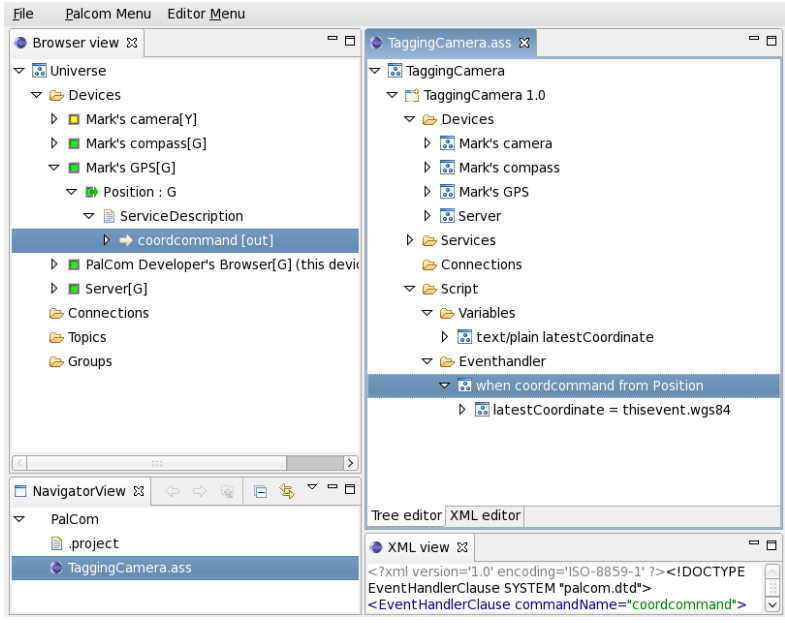


Figure 7.5: *The PalCom Developer's Browser.*

7.4 Domain-specific browsers

Finally, in the reference implementation there are domain-specific browsers, such as the NICU Browser shown in Figure 7.6. The NICU Browser is an experimental specialized browser, built at the University of Siena, Italy, for supporting the Neonatal Intensive Care Unit (NICU) scenario in PalCom. It is similar in functionality to the PalCom Overview Browser, but has another interaction interface in order to adapt to the needs of the medical staff in the particular setting of the NICU.

7.5 Summary

PalCom browsers are tools that can be used for discovering devices and services, inspecting them and their connections and combining them in assemblies. The implemented browsers in the PalCom reference implementation can be placed at different points along the dimensions of functionality, intended users and what devices they can run on. The Handheld Browser is designed for a small screen, while the others make use of a larger screen, such as that of a laptop. The Overview Browser and the Handheld Browser

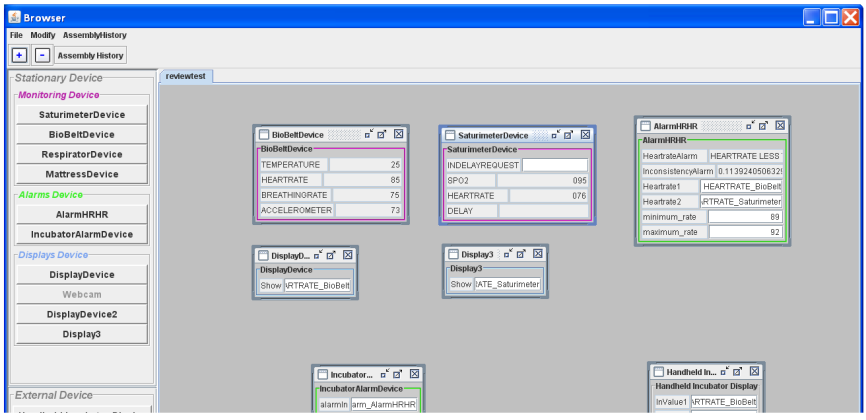


Figure 7.6: *Screenshot of the NICU browser.*

are mainly intended for end users that browse for available devices, services and assemblies, and interact with them in a straight-forward way. The Developer's Browser is for more technically-oriented users, that work with assemblies at a level close to the assembly descriptor language. The NICU browser is hand-crafted for users in the neonatal intensive care domain, while the other three are generic.

We recognize that the implemented browsers do not cover all interesting points along those dimensions. It would be useful to have a browser for small, handheld devices with more advanced functionality than the Handheld Browser, e.g. for managing assemblies. It would also be good to have a way of editing assemblies in a more graphical way than with the Developer's Browser—more like the interaction in the Overview Browser, where new assemblies could perhaps be created by selecting a number of services and choosing to build an assembly for them. This is part of future work. Nevertheless, the implemented browsers have shown potential in the development and use of PalCom application prototypes (see Chapter 9), and they are all examples of applications built on top of the Service Framework and the PalCom middleware, as presented in the following chapter.

Chapter 8

Framework and middleware

As support for the developer of services on devices, the reference implementation of the PalCom open architecture provides a framework and middleware. These handle the communication protocols and execution of assemblies, and have been used when building services for application prototypes in a number of scenarios in PalCom. The code is available as open source at the project web site [88]. The browsers, which are also part of the reference implementation, have been built using the framework and middleware.

A layered overview of the reference implementation is given in Figure 8.1. The platform, at the bottom of the figure, is closest to the hardware on the device. It contains the virtual machines used in PalCom, the Pal-VM and the JVM, and a special thread library called *PalcomThreads*. The middleware contains managers that handle the communication protocols presented in Chapter 5, assembly managers, and *service managers* that handle execution of unbound services. The Service Framework, on top of that, is an object-oriented framework, where services can be implemented by subclassing the class *AbstractService*. As shown at the top of the figure, concrete services, assemblies and browsers execute on top of the framework and middleware. The rest of this chapter will present the different parts of the reference implementation in more detail.

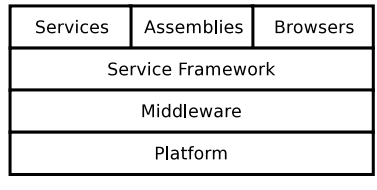


Figure 8.1: *A layered view of the PalCom reference implementation.*

```
public static void main(String[] args) {
  LoudspeakerDevice device
    = new LoudspeakerDevice(new DeviceID("A002372"));
  device.run();
}
```

Figure 8.2: *Java `main` method of a loudspeaker device.*

8.1 Service Framework

The Service Framework is provided for easing the programmer's task of creating new PalCom services. The framework is written in *Pal-J*, PalCom's subset of Java. Pal-J programs can be compiled to run on both the Pal-VM and the JVM. The abstract class AbstractService handles announcement of a service, and in a concrete subclass the programmer can define the command structure of the service, and implement handling of in-going and out-going commands. Through a *device context* object the service gets access to lower-level, device-common middleware managers on the device, and to other services on the device.

There is support for execution of PalCom services on small devices that offer a single thread of control at the Java level. This is the case when running on the Pal-VM: control is given to a Java `main` method, and when that method terminates, the virtual machine stops. An example of such a `main` method is given in Figure 8.2, where the example device is the loudspeakers in the music scenario of Chapter 2. The class LoudspeakerDevice is a subclass of the framework class *AbstractDevice*, as can be seen in Figure 8.3, which shows relationships between central classes in the framework. AbstractDevice has a `run` method that schedules middleware managers and

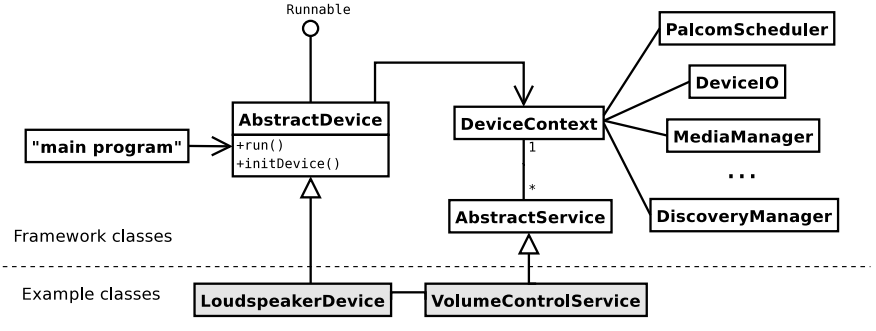


Figure 8.3: *The central classes in the Service Framework, with example classes from the music scenario below the dotted line.*

```
class LoudspeakerDevice extends AbstractDevice {
  LoudspeakerDevice(DeviceID deviceID) {
    super("Loudspeakers", deviceID);
  }

  protected void initDevice() {
    VolumeControlService service
      = new VolumeControlService(context);
    context.addService(service);
    service.start();
  }
}
```

Figure 8.4: *The AbstractDevice subclass that sets up the loudspeaker device.*

services in a *PalcomScheduler*, and runs until all services terminate. The PalcomScheduler is part of the PalcomThreads thread library. On the JVM, the code of Figure 8.2 can be used in a `main` or `run` method.

The code for a slightly simplified version of the LoudspeakerDevice class is given in Figure 8.4. In the constructor, name and addressing information is passed to the superclass for device announcement. In the method `initDevice`, which is called by AbstractDevice on start-up, a volume control service is created, added to the device context and started.

The code for VolumeControlService, which is a subclass of AbstractService, is given in Figure 8.5. In the constructor, the `createServiceProxy` utility method is called for setting up a *service proxy*, which holds the command structure of the service. The service proxy provides functionality for sending out-going commands from the service to connected services, and for handling of in-going commands coming the other way. The commands are the same as were shown for the loudspeakers in Figures 2.2 and 2.7: two in-going commands for adjusting the volume and one out-going command, with one parameter, that informs about the current volume setting. Further, the constructor creates a CommandThread thread object, that is set to receive commands that come in to the service. That thread is scheduled in the `start` method, which is called on device initialization, and where it can be seen how the device context object is used for accessing the PalcomScheduler. The implementation of the private class CommandThread is at the bottom of the class. This is a PalcomThread, and it has a `run` method like normal Java threads. In the `run` method, an eternal loop handles the incoming commands. When a "Volume up" arrives, an event for adjusting the volume is sent to the hardware through a call to the device's *DeviceIO*. DeviceIO is an interface that provides event-based communication with low-level code, such as interrupt routines. Finally, the new value of

```
class VolumeControlService extends AbstractService {
  private CommandThread cmdThread;
  private Command level;

  VolumeControlService(DeviceContext context) {
    super(context, "Volume control",
          createServiceProxy());
    cmdThread = new CommandThread();
    getServiceProxy().addInCommandReceiver(cmdThread);
  }

  private static ServiceProxy createServiceProxy() {
    ServiceProxy sp = new ServiceProxy("Volume control");
    Command up = new Command("Volume up", Command.IN);
    sp.add(up);
    Command down = new Command("Volume down", Command.IN);
    sp.add(down);
    level = new Command("Volume", Command.OUT);
    sp.add(level);
    Param value = new Param("value", "text/plain");
    level.add(value);
    return sp;
  }

  public void start() {
    context.getScheduler().scheduleThread(cmdThread);
    super.start();
  }

  private class CommandThread extends PalcomThread {
    public void run() {
      while (true) {
        CommandEvent event = (CommandEvent) waitEvent();
        String id = event.getCommand().getID();
        if (id.equals("Volume up")) {
          context.getDeviceIO().putOutEvent(Hardware.UP);

          // Send out an out-going command with new level
          level.getParam("value").setData("Normal");
          level.invoke();
        } else if (id.equals("Volume down")) {
          ...
        }
      }
    }
  }
}
```

Figure 8.5: *Code for the loudspeakers' volume control service.*

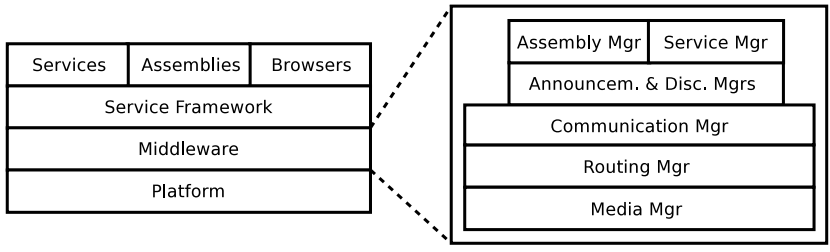


Figure 8.6: *The PalCom middleware is structured as a set of managers, handling the different layers of the communication protocols, and execution of assemblies and unbound services.*

the volume is sent out through a call of `invoke` on the service's out-going command "Volume" (using the attribute `level` in the class).

8.2 Middleware

The reference implementation contains middleware in the form of a number of managers, as illustrated in Figure 8.6. The managers handle different layers of the communication architecture, execution of assemblies and unbound services. They are used by the Service Framework, and can also be used directly by the service programmer where needed.

8.2.1 Communication

There are managers for the different layers in the communication architecture (see Chapter 5):

- **Media managers** handle the Media Abstraction Layer (MAL). There is one implementation of a media manager for each supported network technology on the device, with a common API towards upper layers. The reference implementation contains media managers for UDP, Bluetooth and IR. Media managers handle the mapping between technology-specific addresses and PalCom DeviceIDs, parse the message format in the Wire Protocol, and handle the heartbeat mechanism in the Pacemaker Protocol that is the basis of the Discovery Protocol.
- A **routing manager** handles forwarding of messages using the appropriate media manager, based on DeviceIDs. This is the functionality of the Routing Layer in the communication infrastructure. Dif-

ferent routing manager implementations can use different strategies for message forwarding.

- The **communication manager** handles internal delivery of messages, based on selectors. To upper layers, it offers connections and single-shot messages over unicast, radiocast and groupcast, as specified for the Communication Layer. 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.

- An **announcement manager** handles announcement of a device and its services and connections on the network.

- A **discovery manager** handles discovery of information about other devices. Together, the announcement manager and the discovery manager implement the Discovery Protocol in the Function Layer. The Service Interaction Protocol is handled by the Service Framework, where command communication is supported through the service proxies.

As described in Chapter 5, the format of announced descriptors, when transmitted over the network, is XML. For treatment of descriptors internally on a device, the JastAdd compiler-construction system is used [36]. The format of the descriptors is defined in a JastAdd grammar, from which the tool generates an abstract syntax tree (AST). JastAdd supports AST programming, allowing the PalCom middleware to add computations on the AST as modular aspects. This is used, e.g., in a simple JastAdd aspect where the XML syntax is unparsed from the AST representation, and where the assembly manager (see below) uses JastAdd attributes and equations, operating on the AST, for performing name lookup.

The split into one announcement manager and one discovery manager facilitates use on small, resource-constrained devices, as discussed in Section 5.5.6. Such devices, which offer services, but do not need to discover others, only need to have an announcement manager, but no discovery manager.

8.2.2 Assembly manager

The assembly manager in the reference implementation can store, load and execute assemblies defined in assembly descriptors, where the descriptors are structured as presented in Chapter 6. The assembly manager can be used as an integrated part of a PalCom browser, as for the PalCom Developer's Browser, presented in Chapter 7, or it can run in a stand-alone fashion on a device in the network. In order to support the stand-alone case,

the assembly manager is implemented as a subclass of the AbstractService class in the Service Framework. This allows a user to open a remote-control view of the assembly manager in a browser on a more powerful device, and to load and run assemblies on the small device via the browser.

The JastAdd tool is used for handling assembly descriptors. There is a JastAdd grammar for the descriptor format, according to Section B.2 in Appendix B. This grammar extends the grammar for announced descriptors mentioned above, so for devices and services the same AST nodes are used in the assembly descriptor grammar. Examples of JastAdd attributes that are used for the execution of assemblies are links from uses to definitions of identifiers, semantic checking operations, etc. The information specific to a running assembly, such as current connections and current variable values, is stored in a separate AST data structure, linked to the assembly descriptor AST. This allows several instances of an assembly to run concurrently.

8.2.3 Service manager

A service manager handles unbound services, which are services that are not tied to the device hardware, and can therefore be copied between devices. The service manager in the PalCom reference implementation can load unbound services from binary components dynamically, start them and stop them. When a component is loaded, it starts to run as a service on the device. Like the assembly manager, the service manager is itself implemented as a service with commands for loading a service, and for starting and stopping it. This is used by the assembly manager: if an assembly requires an unbound service that is currently not running, the assembly manager will connect to the service manager and ask it to start the service. If the binary component is available, the service will be started and connected to by the assembly.

8.3 Platform

Figure 8.7 shows the different parts of the PalCom platform. At the bottom, closest to the hardware on the device, are the two virtual machines (VMs) used in PalCom. Pal-VM is the VM for resource-constrained devices that has been developed within the project. It supports programs written in Java and Smalltalk, with interoperability between the languages. JVM is the Java Virtual Machine, as specified by Sun [70]. JVM implementations are available for a wide range of devices.

On top of the VMs, there is a layer which provides PalCom base libraries: `pal-base`, which runs on the Pal-VM, is implemented in Smalltalk, and `pal-jbase`, which runs on the JVM, is implemented in Java. Both have the same interface to upper layers, and can be accessed from code written

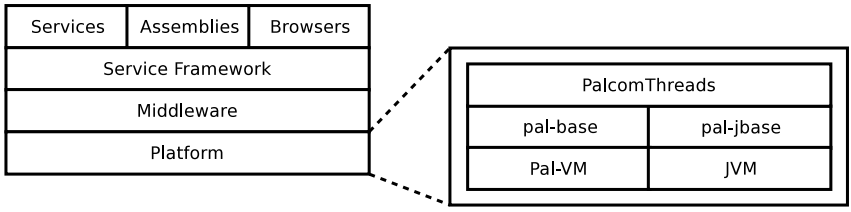


Figure 8.7: *The platform provides execution of programs on two different virtual machines, with a common thread library PalcomThreads.*

in Pal-J, PalCom's subset of Java. This has been utilized in the reference implementation, where common Pal-J code is used for both VMs, for the whole middleware and framework above the base libraries.

Having this kind of support for the JVM is practical, because JVMs exist for many platforms, and on desktop computers Java's graphical libraries can be used when implementing services. There are not yet any graphical libraries for the Pal-VM.

8.3.1 PalcomThreads

The main difference between Pal-J and standard Java is in the support for concurrent execution and threads. Pal-J programs use threads defined in the PalcomThreads library, instead of `java.lang.Thread`. The PalcomThreads are built on *coroutines*, which is the fundamental abstraction for concurrency provided by the base libraries. Coroutines are supported directly by the Pal-VM and implemented using Java threads for the JVM. The PalcomThreads are scheduled by a PalcomScheduler, which relies on a Unix *select* mechanism for timeouts and non-blocking I/O. The scheduler can be handled explicitly by the programmer, in contrast to normal Java. This makes it possible to use specialized schedulers, e.g. using other scheduling schemes.

The PalcomThreads library is modelled after SimIOProcess in the Lund Simula system [71]. It is designed for hierarchical scheduling, where schedulers can themselves be scheduled by other schedulers at any number of levels. This mechanism can be used for separating groups of threads from others, such as those implementing different services, and it allows support for migration of running threads between devices. The scheduling is currently cooperative, meaning that each thread is allowed to run until it gives up execution explicitly, but it is possible to extend it to preemptive scheduling, where the scheduler can interrupt the execution of threads and reschedule them. The PalcomThreads library contains classes implement-

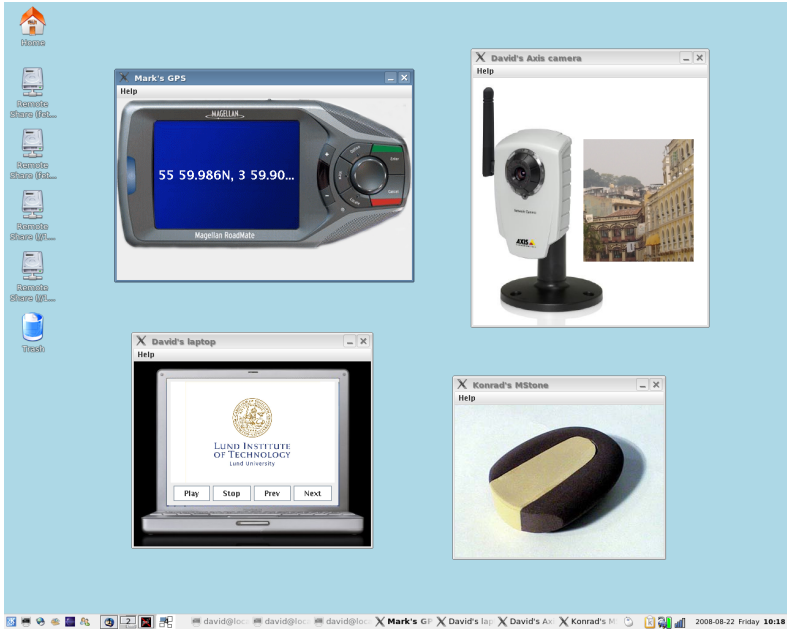


Figure 8.8: *Simulated devices on the desktop.*

ing monitors, semaphores and event-based communication for achieving synchronization between threads.

8.4 Simulated devices

The PalCom reference implementation contains a small framework for *simulated devices*. These are used as tools in a development situation, where it is valuable to be able to simulate devices, i.e., to run their services on a general-purpose computer rather than on the device itself.

The simulated devices are Java programs that run on a desktop machine, with graphical representations of the hardware of a device, as shown in Figure 8.8. Code written using the Service Framework can be run without modification in simulated devices first, before deploying onto real devices. That gives advantages in terms of easier debugging, and easy creation of multiple devices for testing purposes. This has been used in the PalCom project for simulating a number of devices, e.g., cameras, GPS devices, etc. Simulated devices run as separate operating system processes, typically on a laptop, and use normal networking. This allows easy debugging in the

development situation, as well as more realistic simulations of scenarios, if the simulated devices are placed on different physical laptops.

The simulated devices, and also the browsers discussed in Chapter 7, utilize that the middleware and framework are compatible with the JVM, as they use the Java Swing graphics library for GUI code [106]. Swing is available for the JVM, but currently not for the Pal-VM.

8.5 Summary

The PalCom Service Framework

- is an object-oriented framework, where the PalCom developer can subclass AbstractService and AbstractDevice for implementing services on a device.
- supports the developer by providing data-structures for command communication.
- handles the interaction with device-common middleware managers that perform routing, announcement and discovery, and that map to different network technologies at the MAL level.
- has been used for implementing the browsers presented in Chapter 7, and the services in the scenarios presented in the following chapter.

The PalCom reference implementation also contains a small framework for simulated devices, and the PalcomThreads thread library, which is based on coroutines and where the scheduler can be accessed explicitly by the programmer.

Chapter 9

Implemented scenarios

The mechanisms for services and assemblies presented in the previous chapters, and the supporting middleware and Service Framework, have been used by partners in the PalCom project when building prototypes for different user scenarios. The work in the project has been carried out according to a cyclic, iterative process, where the design of the application prototypes, and experiences from their use, has continuously influenced the architecture, and vice versa [82]. The Traveling Architects activity was one part of this process [24].

The iterative work process means that during the first years of the project, the software for the prototypes was developed on top of different platforms and operating systems, while during the final year the developed reference implementation was used in all presented prototypes, with the middleware and the Service Framework as the basis for PalCom services. The scenarios and prototypes have been presented in papers and demonstrated at various events, notably the project reviews in March and December 2007, the IST event in Helsinki 2006 [85], and the Tall Ships' Race in Aarhus 2007 [83]. This chapter will give some examples of services and assemblies built for the prototypes.

The author of this thesis has only been involved in the development of some of the concrete prototypes, and then mainly in development of initial, simulated devices, as discussed in Section 8.4. The work presented in the thesis has concentrated on the design and implementation of the architecture, the protocols, and the supporting framework, while this chapter exemplifies with prototypes that have been developed on top of it, mainly by other project members. The purpose is to demonstrate the variety of use supported by the PalCom architecture and its implemented framework.

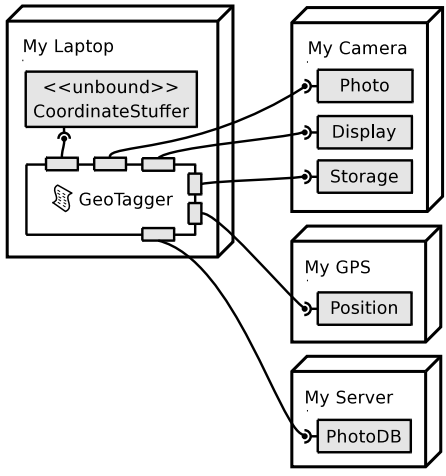


Figure 9.1: *The GeoTagger assembly.*

9.1 GeoTagger

The GeoTagger scenario has been used in examples in Chapters 4 and 6. It is one of the scenarios that have been studied by the PalCom group at the University of Aarhus, in cooperation with landscape architects in Scotland. A real application was developed and used by the landscape architects, and there is also an executable demo with simulated devices, implemented on top of the Service Framework. The demo is available at the PalCom web site [84]. The structure of the first part of the demo, as shown in Figure 9.1, is quite similar to the set-up in Chapter 4: the assembly *GeoTagger* has the combined functionality of the TaggingCamera and Uploader assemblies presented there. The latest coordinate from *My GPS* is continuously saved in a variable. When the user takes a photo with *My Camera*, the assembly gets a notification from the service Photo, and requests the new photo from the service Storage. As the photo is returned from Storage, the unbound service CoordinateStuffer is used for tagging the image with the latest coordinate, before sending it to the photo database on *My Server*.

In the demo, there is also an additional assembly, which gives useful functionality and which illustrates that a service can participate in more than one assembly at the same time. That assembly, *ExtendedGeoTagger*, places the taken photos at the right location on the world map in the Google Earth application [42]. ExtendedGeoTagger runs next to GeoTagger, as shown in Figure 9.2. The additions, compared to Figure 9.1, is ExtendedGeoTagger and GoogleEarthService on *My Laptop*. The connections managed by the existing GeoTagger assembly are drawn in gray in this figure. Extended-

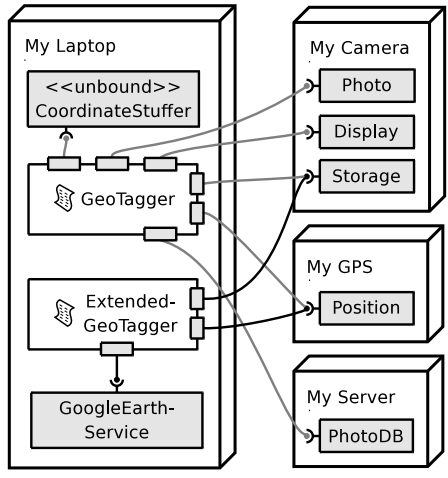


Figure 9.2: *The ExtendedGeoTagger assembly.*

GeoTagger picks up new images from the service Storage, and coordinates from Position, and sends them to GoogleEarthService, which shows the images in Google Earth.

This way of letting one assembly add functionality to another is slightly different from the way Uploader added functionality to TaggingCamera in Chapter 4. While Uploader connected to TaggingCamera's synthesized service, ExtendedGeoTagger connects directly to some of the services used by GeoTagger. This shows that systems of cooperating assemblies can be organized in different ways, and that services—Storage and Position in this example—can take part in multiple assemblies. Structuring the demo with synthesized services would have worked equally well.

9.2 SiteTracker

The SiteTracker is another landscape architecture scenario, developed by the group at the University of Aarhus and the landscape architects. Like GeoTagger, the SiteTracker prototype supports the landscape architect during *visual assessment*. The task is to evaluate how a planned wind mill would look from different points in the Scottish countryside. SiteTracker enables this evaluation out at the site, while driving in the car. Without support, it would be very difficult to visualize how the wind mill would look from a distance, especially when the car is moving. SiteTracker shows a video stream of the view, on a screen inside the car, where the wind mill is

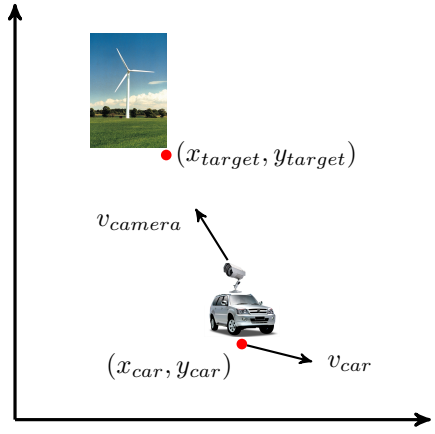


Figure 9.3: *Overview of the SiteTracker scenario.*

drawn at its planned location. That makes it much easier for the landscape architect to make the evaluation.

The SiteTracker prototype is built as an assembly of PalCom services. It includes a camera, mounted on the roof of the car, which rotates so it always looks in the direction of the wind mill. This is implemented as illustrated in Figure 9.3. The current location of the car, (x_{car}, y_{car}), and the current driving direction, v_{car}, are given by a GPS and a compass in the car. From those, and from the currently configured wind mill location, (x_{target}, y_{target}), the viewing direction of the camera, v_{camera}, is computed, relative to v_{car}.

Figure 9.4 shows one version of the SiteTracker assembly set-up. One assembly, *LocationHeading*, provides the location and compass heading in its synthesized service *LHSynth*, which is used by the assembly *VideoAssembly*. VideoAssembly handles the rotation of the camera, and the display of the video stream with the wind mill visualization. Both assemblies run in an assembly manager on a laptop in the car.

The LocationHeading assembly gets GPS coordinates and compass headings from services on the GPS and Compass devices. The coordinates come from the GPS in a format that is not suitable for use by the rest of the system, so the unbound services GPSParser and GeoConverterService are used for converting them to another format. Further, LocationHeading uses an unbound service LocationHeadingService for performing one additional computation: if a speed value from the GPS indicates that the car is moving sufficiently fast, movement direction values from the GPS are used instead of the compass headings—those values have higher precision. After that computation, the location and the direction are sent to users of the synthesized service (to VideoAssembly, in this case).

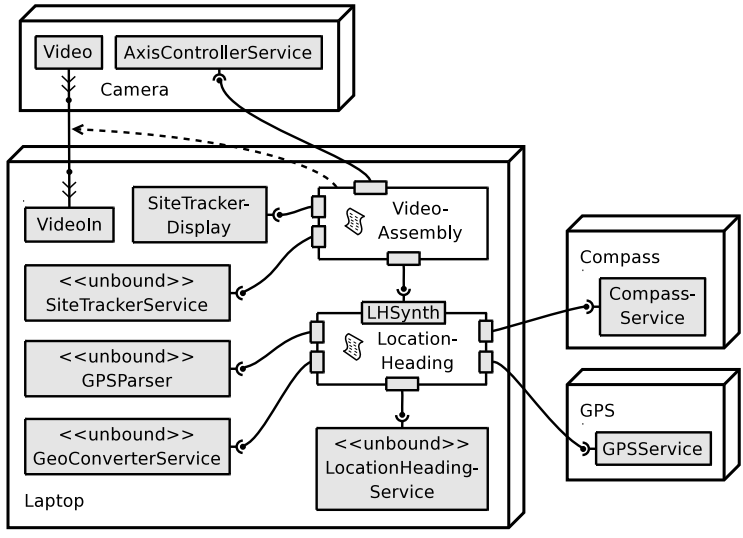


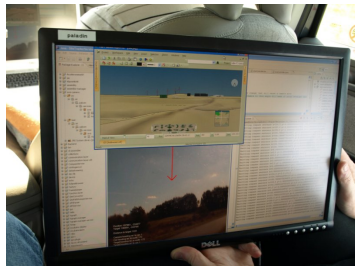
Figure 9.4: *The SiteTracker assembly set-up.*

VideoAssembly has connections to AxisControllerService on the camera, to the service SiteTrackerDisplay, and to the unbound service SiteTrackerService. It also manages a streaming connection between the service Video on the camera and VideoIn on the laptop. AxisControllerService has commands for controlling pan, tilt and zoom of the roof camera, which is an Axis dome camera that can pan 360 degrees. SiteTrackerDisplay handles the drawing of the wind mill on the video display window, at screen coordinates provided as parameters in in-going commands. The VideoIn service on the laptop displays the streaming video from the camera.

The behaviour of VideoAssembly is driven by the unbound service SiteTrackerService. SiteTrackerService gets coordinates and headings from VideoAssembly continuously, and saves them in variables. Once per second, it computes a new direction for the camera (v_{camera}), and screen coordinates for the wind mill drawing. Those are sent to VideoAssembly, which forwards them to AxisControllerService and SiteTrackerDisplay. The camera follows the wind mill location, and the visualization gets updated.

The SiteTracker prototype has been tested with the landscape architects out in the field, as shown in Figure 9.5. The version of SiteTracker presented here does, however, not exactly match the version presented by the Aarhus team in 2007. That version had additional 3D visualization, through a connection to the Topos application [1]. For practical reasons, the video connection between the camera and the laptop was not a PalCom streaming connection, but a Motion JPEG connection, as provided by

(a) Car with the SiteTracker camera mounted on the roof.



(b) On a screen in the car, the location of the planned wind mill is marked.

Figure 9.5: *The SiteTracker out in the field.*

the native implementation on the Axis camera.[1] The structure with two assemblies, where one provides its values to the other through a synthesized service, is also not exactly as demonstrated, but this set-up is functionally equivalent, and has been chosen in order to show how assemblies can be structured hierarchically.

The division in functionality between assembly descriptors and unbound services is interesting to look at, for a complex set-up such as the SiteTracker. LocationHeading and VideoAssembly have one unbound service each, that has been constructed specifically for performing SiteTracker functionality. LocationHeadingService checks the speed value from the GPS, and chooses one of two direction values based on that. With conditional execution (`if`) in the assembly descriptor language, and simple conditional expressions, that would have been possible to express directly in an event handler, and the unbound service would not be needed. The unbound service SiteTrackerService performs more complicated arithmetic computations, and has a periodic behaviour where events are generated every second. For periodic behaviour in assemblies, we have experimented with small timer services, which could probably have been used in this scenario. In order to allow computations such as those for the screen coordinates and the camera direction, the natural extensions of the assembly descriptor language would be to support arithmetic expressions, and numeric types for variables. See Chapter 11 for further discussion about potential extensions of the assembly descriptor language.

[1] Providing a video stream from an Axis camera as a PalCom service has been demonstrated in the master's thesis project [72], but it was not available in the dome camera at the time.

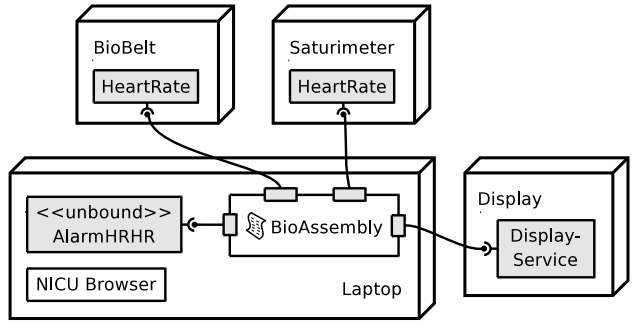


Figure 9.6: *The BioAssembly in the Incubator scenario.*

9.3 The Incubator

The Incubator prototype has been developed by the PalCom group at the University of Siena, Italy, in cooperation with doctors and neonatologists at the 'Le Scotte' hospital in Siena. It has been presented in the paper [47]. The prototype supports the work where prematurely born children are taken care of in an incubator. This is a setting where many devices and instruments are attached to the incubator and to the child. Traditionally, those devices have had their own displays and alarm systems, and not really been integrated into one system. Sometimes, the first way to notice irregularities, stemming from one instrument, has been a changed behaviour of the child and alarms on the displays of other instruments. Standard work practice for neonatologists, if a display starts showing measurement values that are outside the normal range, has been to check the equipment attached to the child, and maybe replace some instrument, in order to leave out machine errors before the child is examined further.

The work on the Incubator prototype has consisted both of finding new devices for unobtrusive measuring of biosignals on the child, and of enabling flexible integration between the devices by letting them offer PalCom services. The unobtrusive measuring work has resulted in a soft belt called the BioBelt, which is placed around the infant's chest and which has biosensors directly integrated in the tissue. The introduction of PalCom services lets doctors construct assemblies that combine a set of devices, for performing alarm functionality or other tasks.

One of the Incubator assemblies is called *BioAssembly*. It is illustrated in Figure 9.6. BioAssembly applies a redundancy technique for lowering the risks associated with not knowing if it is the child or a machine that behaves in an irregular way. The devices involved are the BioBelt that is attached to the child, a *saturimeter*, a display and a laptop. The saturime-

ter measures the child's oxygen saturation (the percentage of haemoglobin binding sites in the bloodstream that are occupied by oxygen), and also gives a heart rate value. The laptop runs the NICU Browser, which permits inspection of services (see Section 7.4), and it has an assembly manager where BioAssembly runs. The BioBelt and the saturimeter both have services for different medical measurements, but the BioAssembly only uses the service *HeartRate* on both devices. BioAssembly takes help from an unbound service *AlarmHRHR* for performing its functionality. When heart rate values arrive from one of the two instruments, they are forwarded to AlarmHRHR. AlarmHRHR compares the two values with each other, and with configured threshold values. If the values differ from each other by more than 15 %, an inconsistency alarm is sent to the assembly and shown on the display. That means one of the instruments is probably malfunctioning. If the values are sufficiently close, they are compared with the threshold values. If a value is outside the thresholds, an alarm about the child's health is triggered.

The Incubator setting has very heterogeneous devices. BioAssembly demonstrates useful functionality that is enabled by letting them offer PalCom services, and using the assembly mechanism for combining them. Using the assembly for the alarm handling is natural, because it is a concern that spans across several devices and cannot be implemented in one of them. Regarding the unbound service, its functionality is not much more advanced than what can be expressed directly in our current assembly descriptors. If a construct for conditional execution (`if`) was added, together with comparison expressions for comparing numerical values, the tests in AlarmHRHR could have been done directly in event handlers instead, which would have simplified the assembly.

9.4 Active Surfaces

Active Surfaces is another scenario investigated in cooperation with the 'Le Scotte' hospital in Siena [14, 45, 46]. The PalCom group at the University of Siena have built a prototype consisting of a set of floating tiles. The tiles are used in a swimming pool setting, as aids for physical-functional and cognitive rehabilitation of physically and mentally impaired children. In exercises designed by therapists, the children assemble the tiles into meaningful configurations.

Part of the background to this work is a traditional lack of integration between physical and cognitive rehabilitation. The cognitive tasks have usually been too static, and the children have easily lost attention. On the other hand, motor rehabilitation is very demanding at a physical level, and is based on repetitive sequences of actions: patients often perceive them as tiring and not engaging. Here, the Active Surfaces allow an in-

Figure 9.7: *The tiles hardware.*

tegration of these two therapeutic goals with the activity. In addition, the exercises are performed in the water, which is interesting from the activity perspective: water creates a safe context where impaired people can move autonomously, relying on the added support to the body, something they cannot do elsewhere.

Through interaction with PalCom assemblies running on the tiles, the therapist can inspect and change their configurations. This way, she can adapt the therapeutic activity in the middle of an exercise, and the visibility given by the assemblies helps her cope with unexpected breakdown situations.

9.4.1 The prototype

The floating tiles in the Active Surfaces prototype can be connected to each other to form a network (see Figure 9.7). Each of the tiles is a resource-constrained embedded system that communicates using only a low-bandwidth short-range infrared link. By having a simple composable physical appearance and multi-purpose programmable hardware, the tiles support multiple games or exercises. On each of a tile's four sides magnets are placed, to make the tiles "snap" together when they are in close vicinity. On the top of the tile is a replaceable plastic cover, also held in place by magnets. The image on the cover depends on the game. On each side of the tiles, light emitting diodes (LEDs) provide visual feedback to the user. There is also a special tile, called the *assembler tile*, which is used by the therapist for configuration.

Inside each tile, an embedded system uses the infrared light to communicate with other tiles, and detect their presence. Two tiles can only communicate if they are close to each other. The main computational unit is a UNC20 module, which is an ARM7-based embedded system running uClinux [111] at 55 MHz, with approximately 8 MB RAM. The UNC20

(a) Two tiles are placed correctly. The correct sides are lit.

(b) The upper-right tile has all neighbours correct.

(c) The complete puzzle is solved.

Figure 9.8: *The puzzle game.*

module runs the Pal-VM (see Section 8.3), and it communicates with a sideboard using a serial connection. The sideboard is responsible for controlling the infrared communication and the LEDs. The bandwidth of the infrared communication is approximately 600 bits per second.

9.4.2 Games

Several different games have been designed for the Active Surfaces. To be appropriate for the tiles, a game should support both physical and cognitive rehabilitation, while at the same time be implementable on the resource-constrained devices. Furthermore, to be able to help a wide range of patients, the set of games should be of varying difficulty, both on the physical and on the cognitive level. Finally, the games should be open-ended and configurable so that they can be adapted and customized to each rehabilitation session. To change the current game, the therapist connects the tile to a PDA running PalCom software. Since the PDA is not suited for a wet environment, this should be done prior to the training activity.

One of the designed games, *catch*, is meant to only require simple cognitive effort, but challenges the patient's reflexes, speed, and coordination. Here, the child holds one tile, and three or four other tiles are placed in a row, floating on the water. When one of the tiles lights up, the child should place his or her tile next to the lit one, as quickly as possible. Another game, *scrabble*, has the requirement that the child should be able to form words out of letters. There are letters on the tiles, and the child uses them to create words, while the LEDs provide feedback when correct words are formed. During the session, the therapist can change the faces of the tiles to different letters.

The third game, *puzzle*, is the one for which the prototype implementation has come furthest. It is a traditional puzzle game, in which an image is

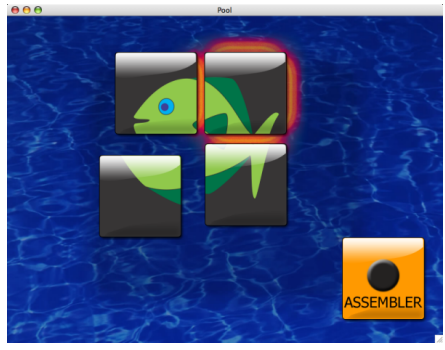


Figure 9.9: *A screenshot from the tiles simulator.*

created by assembling the tiles in a specific pattern. The face of each tile is part of a larger image, as can be seen in Figure 9.8. Initially, the tiles are spread in a random pattern, after which the child starts to solve the puzzle. As the game progresses, the child gets continuous feedback from the LEDs. When two tiles are connected correctly, the corresponding sides light up, as shown in Figure 9.8(a). When all of a tile's neighbours are correct, all sides of that tile light up (b), and finally when the puzzle is solved, the outline of the solution starts blinking (c).

During the session, the therapist can change the faces of the tiles to make a new puzzle. To reprogram the tiles, the special assembler tile is used. The assembler tile has the same physical appearance as the other tiles, but it also has a button. To make the tiles remember the new solution, they are arranged in the solution pattern, the assembler tile is put next to one of the tiles, and the button is pressed. After this, the tiles will remember the new solution and can be scattered randomly again. The LED feedback can be configured by the therapist to alter the difficulty level of the game. It is, e.g., easier to solve the puzzle if all the outer edges of the final solution will light up as the game is started.

9.4.3 A tiles simulator

In order to ease the development of game logic and software for the tiles, a simulation framework has been developed by Jeppe Brønsted at Aarhus University. This framework, as presented in the paper [14], can be used to experiment with the tiles on a standard PC. A screenshot is shown in Figure 9.9. Having the simulator available made it possible to develop tiles software and hardware in parallel, and high level tools that were not available for the embedded platform could be used for debugging and profiling. Furthermore, testing involving repeated rearrangement of the tiles was

much easier using a mouse in a graphical user interface, than physically moving the actual tiles around.

The simulator consists of a model of the swimming pool as a medium for infrared communication, and a graphical user interface for manipulating the physical location of the tiles. The user interface is connected with the pool model, so that when a tile is moved in the user interface, the pool model is updated accordingly. The software implementing the functionality of the tiles in the simulator is divided into services, and implemented using the Service Framework (see Chapter 8). This allows the services to be combined into PalCom assemblies.

For the simulated infrared communication, a MAL Layer media manager has been implemented. When a tile sends a message, the MAL Layer of the tile accesses the pool model to determine which tiles the tile is connected to (if any), and delivers the message accordingly. From an application developer's perspective, it is transparent whether the simulation framework or the physical hardware is used. The only part of the software on the tile that interacts with the simulation framework is in the MAL Layer, and therefore system behaviour experienced on the simulator should be similar on the embedded platform.

9.4.4 Groupcast tiles

The puzzle game, as described in Section 9.4.2, has been implemented for the tiles simulator, including configuration of the puzzle solution using the assembler tile. Similar functionality is also available in a later, experimental implementation by the author of this thesis. In that implementation, named *Groupcast tiles*, each tile is a simulated device (see Section 8.4), which communicates using UDP, instead of simulated IR communication.[2] The reason for building the Groupcast tiles implementation was to demonstrate use of groupcast communication between PalCom assemblies in the algorithm carried out during the puzzle game (see the following section). The mechanisms for groupcasts were not available when the tiles simulator was built, and the tiles simulator used lower-level broadcast communication directly between services. The structure of services and assemblies on the tiles is otherwise similar for Groupcast tiles and the tiles simulator.

9.4.5 Puzzle game logic

During the puzzle game, each tile changes between three states: *unhappy*, *locally happy*, and *globally happy*. The states correspond to the types of feed-

[2] We have concluded that tiles communicating using UDP is relevant for this demonstration, even if messages can reach tiles that are not "physically" connected to the sending tile. The service *Connectivity* on the tiles, which handles the relationships to neighbour tiles, works in the same way as for the tiles simulator.

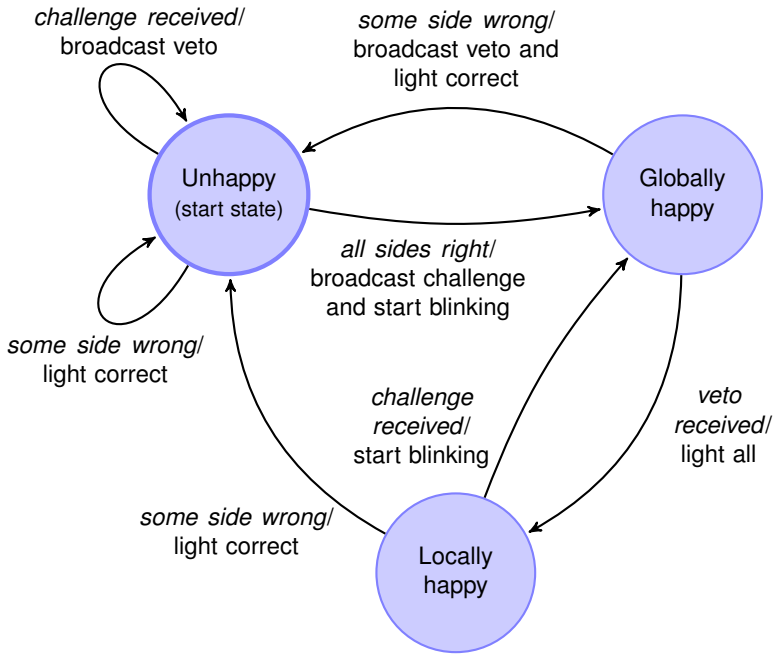


Figure 9.10: *State diagram for a tile in the puzzle game.*

back given by the tiles. In Figure 9.8, the top-right tile is unhappy in (a), locally happy in (b), and globally happy in (c). Three rules determine which state a tile is in:

1. A tile is **unhappy** if it has less than four correct sides.[3]
2. It is **locally happy** if it has four correct sides but at least one of the other tiles is unhappy. This means that the tile has found its place in the puzzle, but the complete puzzle is not solved.
3. If no tile is unhappy then all tiles are **globally happy**. The puzzle is solved.

As can be seen from these simple rules, the game has a notion of global state, namely whether there is at least one unhappy tile. This information is used by the tiles to distinguish whether the tile is locally happy or globally happy. If a tile has less than four correct sides, it does not need this information (because of rule 1).

[3]We define a correct side of a tile to be a side that has a correct neighbour, or has no neighbour and should have no neighbour according to the solution.

The state diagram in Figure 9.10 illustrates the algorithm followed by each tile in the puzzle game: the states in the diagram are states of an individual tile. The edges are marked with events that happen to the tile (in italic), and with actions taken by the tile (after the slash). Events come either when the set of neighbour tiles changes, or when a broadcast message is received.

The global state is maintained by handling two situations: The first situation occurs when an unhappy tile observes that it has four correct sides instead of three. This event is modelled as *all sides right* in the diagram. The tile then goes to the globally happy state, starts blinking its LEDs, and broadcasts a challenge to the other tiles (by using the groupcast mechanism), requiring any unhappy tiles to reply immediately (also using groupcast). If a response *veto* is received, the tile stops blinking and goes to the locally happy state. When a locally happy tile receives a challenge, it treats it as if it was originating from the tile itself: the tile goes to the globally happy state, and waits and sees if any veto responses come. It is assumed that the solution of the puzzle is connected and includes all tiles, and therefore it cannot be the case that no tiles are unhappy in a proper subset of all the tiles. Therefore, if there is an unhappy tile, there is an infrared communication path from that tile to the tile that initiated the challenge.

The second situation, inverse to the first one, occurs when a tile observes that it has three correct sides instead of four (*some side wrong* in Figure 9.10). The new state of the tile is now unhappy, and the LEDs of the sides that are correctly placed should be lit. If the tile was globally happy before, the other tiles are unaware that the tile is now unhappy, and therefore a message is broadcasted specifying so (*veto*). If the tile was locally happy before, it can assume that there is at least one unhappy tile in the graph it is connected to, and no broadcast is needed.

9.4.6 Services and assemblies

A set of PalCom native services encapsulates the basic hardware functionality of the tiles, and each game is implemented as an assembly and one or more unbound services, that can be connected to these services. Figure 9.11 shows the services and the assembly *TileAssembly* on a tile in the four-tile puzzle game, and the groupcast communication between the assemblies. The set of services and assemblies is similar on tiles 1, 2, 3 and 4.

The basic services of the tiles are a *LEDs* service controlling the LEDs, and a *Connectivity* service detecting the presence of neighbour tiles. In the real tiles implementation, the information about neighbours is available to the Connectivity service by inspecting the MAL Layer. The assembler tile also has a *Button* service receiving input from the button. The combination of the assembly and the *Puzzle* unbound service for the game logic can be

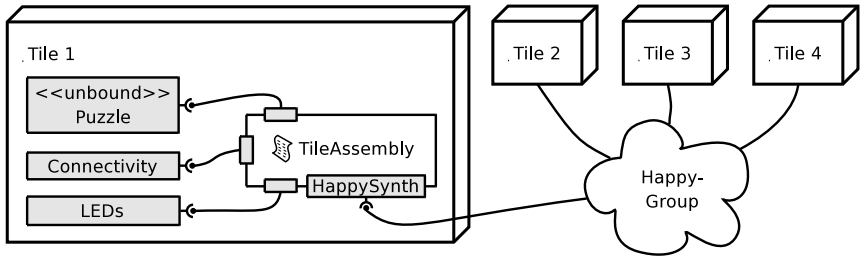


Figure 9.11: TileAssembly *and the unbound service* Puzzle *handle the puzzle game on one tile. Tiles 2, 3 and 4 have the same set of services as Tile 1, and a similar assembly.*

replaced when switching to another game. Figure 9.12 shows the assembly descriptor for TileAssembly:

- On lines 18–21, the command `connectivityUpdate` from Connectivity is forwarded to the unbound service Puzzle. Puzzle implements the state machine shown in Figure 9.10, and handles `connectivityUpdated` as one of the events *all sides right* or *some side wrong*.
- When Puzzle decides to control the LEDs, it sends out the command `lightLEDs`, which is handled on lines 22–24.
- The assembly has a synthesized service *HappySynth*, whose interface is defined on lines 38–41. The synthesized service uses groupcast communication, and participates in the group *HappyGroup*. As TileAssembly runs on all four tiles, there are four members in the group. The two commands used in the groupcast communication are `challenge` and `veto`. Those are, by the definition of groupcast communication, both out-going and in-going. The assembly can both send and receive them.
- On lines 25–30, groupcast commands are sent out when `challenge` or `veto` come from Puzzle, according to broadcasts shown in the state diagram in Figure 9.10.
- The event handlers on lines 31–36 handle receival of groupcast commands, by forwarding them to Puzzle.

In parallel with the four TileAssembly instances, a fifth assembly *ConfigureAssembly* runs on the assembler tile (see Figure 9.13). It handles configuration of the correct puzzle solution, by receiving `buttonPressed` commands from the Button service on the assembler tile, and sending `saveConfiguration` commands to the Puzzle unbound services on each tile.

```
1  assembly TileAssembly 1.0 released {
2    this = ServiceID;
3    devices {
4      Tile = DeviceID;
5    }
6    services {
7      Puzzle = ServiceInstanceID on Tile;
8      Connectivity = ServiceInstanceID on Tile;
9      LEDs = ServiceInstanceID on Tile;
10   }
11   connections {
12     Puzzle <-> this;
13     Connectivity <-> this;
14     LEDs <-> this;
15   }
16   script {
17     eventhandler {
18       when connectivityUpdate from Connectivity {
19         send connectivityUpdated(thisevent.neighbours)
20           to Puzzle;
21       }
22       when lightLEDs from Puzzle {
23         send setLEDs(thisevent.sides) to LEDs;
24       }
25       when challenge from Puzzle {
26         invoke challenge() on HappySynth;
27       }
28       when veto from Puzzle {
29         invoke veto() on HappySynth;
30       }
31       when challenge from HappySynth {
32         send challengeReceived() to Puzzle;
33       }
34       when veto from HappySynth {
35         send vetoReceived() to Puzzle;
36       }
37     }
38     synth HappySynth groupcast HappyGroup {
39       in out challenge();
40       in out veto();
41     }
42   }
43 }
```

Figure 9.12: *The TileAssembly descriptor.*

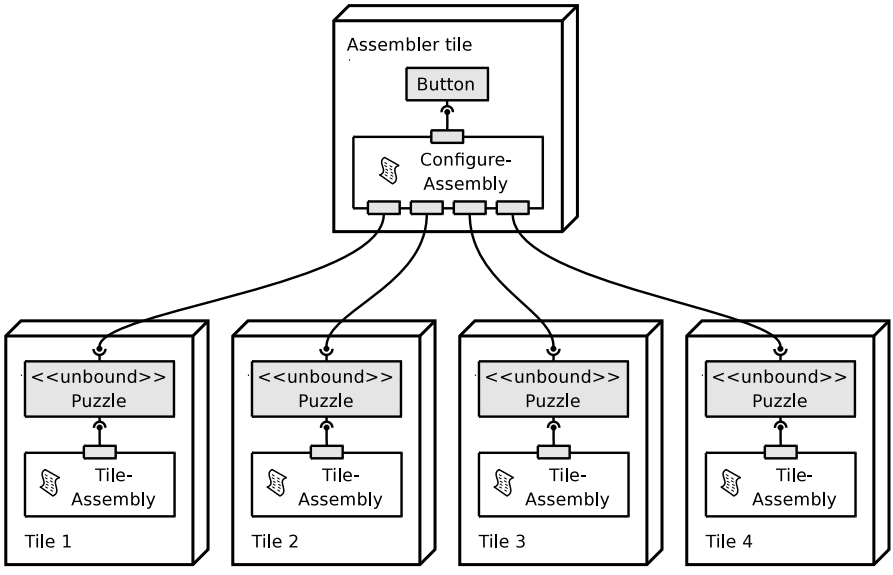


Figure 9.13: *The assembler tile runs* ConfigureAssembly. *It connects to the Puzzle services on the other tiles.*

In Puzzle, a `saveConfiguration` command is handled by saving the current set of neighbour tiles, as received from Connectivity, as the correct local solution, and updating the tile state accordingly.

9.5 PalCom on a Sun SPOT

Sun SPOT (Sun Small Programmable Object Technology [104]) is a small computer. It has wireless communication, an accelerometer, a temperature sensor, LEDs for output, buttons for input, and I/O pins for interfacing attached hardware. Sun SPOT software is implemented in Java, and there is a small Java Virtual Machine, called Squawk, that runs directly on the hardware, without an intermediate operating system. The Sun SPOT runs CLDC, a configuration of the Java Micro Edition [105].

Due to its form factor—a Sun SPOT device fits in the palm of a hand—and due to the wireless communication, the Sun SPOT is attractive as a platform for building pervasive computing prototypes. In a project in Lund, referred to as PalSpot, Jörgen Ellberg ported the PalCom middleware and Service Framework to the Sun SPOT platform, and demonstrated its use in a PalCom assembly. The CLDC libraries differ a little from the PalCom libraries, which are written in Pal-J (see Section 8.3). Therefore, the port was

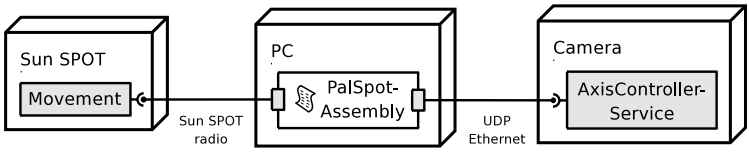


Figure 9.14: *The PalSpot assembly.*

not completely straightforward, and scripts had to be made that converted uses of certain library classes in the code.[4] Further, it was a challenge to make the software fit into the limited memory on the Sun SPOT. The third main part of the work was to create a new media manager for Sun SPOT radio communication in the MAL Layer (see Section 8.2.1).

The implemented assembly demonstrated use of the Sun SPOT as a nice and intuitive remote control for an Axis dome camera. In this case, the camera was used with the Major Incidents Overview prototype in the Tall Ships' Race scenario (see Section 9.7). The PalCom software on the camera was the same as for the car roof camera in the SiteTracker, presented above in Section 9.2. The remote control was implemented as a PalCom service *Movement* on the Sun SPOT, that captured the device's movements by interfacing the accelerometer, and that sent out commands reflecting the 3D rotational coordinates. An assembly connected the Sun SPOT to the camera, as shown in Figure 9.14. The assembly ran on a PC, which communicated with the camera in an Ethernet IP network. Through a Sun SPOT base station, attached to a USB port on the PC, the PC could communicate wirelessly with the Sun SPOT. By pressing a button on the Sun SPOT and tilting the device, the camera could be panned and tilted in a smooth way.

The PalSpot demonstration was an example of routing of PalCom communication between different network technologies (Sun SPOT radio and UDP over Ethernet, as shown in Figure 9.14). The Sun SPOT had a Sun SPOT media manager, as implemented in the PalSpot project, and the camera had a UDP media manager for IP communication. The PC, where the assembly executed, had both those media managers, and a routing manager in the Routing Layer that connected the two.

9.6 A bridge between PalCom and UPnP

In his master's thesis project [64], Johan Kristell implemented bridging software that made UPnP [114] devices and services available in PalCom

[4] With the scripts, the code in the central PalCom code base could be used for building the Sun SPOT PalCom libraries. This means that we did not have to branch off a complete source tree, but could continue to follow the development in the main tree.

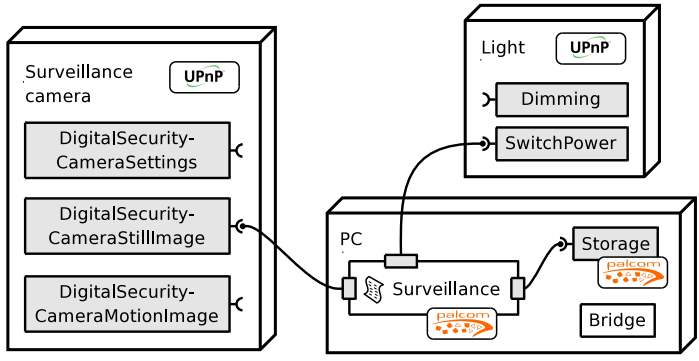


Figure 9.15: *A surveillance scenario with UPnP devices. The UPnP devices have been tagged with the UPnP logo, and the PalCom assembly and the PalCom service storage with the PalCom logo.*

networks, and vice versa. The main benefit of such a bridge is that the large base of available UPnP devices can be utilized in PalCom assemblies, and be combined with each other and with PalCom devices in a more flexible and light-weight way than with UPnP only.

The bridge was demonstrated in a surveillance demo scenario, as illustrated in Figure 9.15. It made use of the bridging of UPnP devices to PalCom. The set-up includes three devices, used for surveillance in a building. The camera is a UPnP device, adhering to the Digital Security Camera V1.0 standard (see [113] for UPnP Device Control Protocol standard documents). The light is also a UPnP device, implementing the Lighting Controls V1.0 standard. It is connected to a motion detector, that turns on the light if someone enters the building. The third device is a PC that runs the assembly *Surveillance*, the bridging software, and a PalCom storage service that can store images.

The bridge performs UPnP discovery, generates PalCom device and service descriptions for the surveillance camera and the light, and uses the Service Framework for announcing them. The assembly can interact with them as if they were PalCom devices. If someone moves in the building, the light is turned on and the service SwitchPower on the camera sends out a notification to the assembly. Then, the functionality of the assembly is that it asks the service DigitalSecurityCameraStillImage on the camera to take a picture. When the picture comes to the assembly, it is sent to the Storage service for storage, and future inspection.

In order to realize this scenario with UPnP only, a custom UPnP *control point* would have had to be implemented (control points are programs, typically implemented in C, C++ or Java, that use UPnP services). That is not as light-weight as the PalCom assembly, and cannot be modified as

easily, as discussed in Chapter 4. Using the bridge in the other direction, making use of PalCom devices and services from UPnP control points, is not immediately useful, because PalCom devices in general do not follow any of the 16 UPnP device types.

The characteristics of the UPnP standards influenced how the bridge could be implemented. Only the simplest device types, of which the Lighting Control is one example, could be bridged fully automatically. For the more complex types, the use of different conventions for different UPnP types made type-specific adaptation code necessary in the bridge. E.g., for transfer of large pieces of data, such as images, several UPnP types provide a URL for retrieving the data out-of-band, over HTTP (all UPnP devices run a Web server). There is, however, no established convention, for naming or otherwise, that can be used for detecting such a URL automatically, and presenting it as a command with an image parameter in a PalCom service description. Another example is that lists of values are formatted in different ways for different UPnP types (sometimes comma-separated, sometimes in other formats). Thus, adaptation code is needed in the bridge. Still, this is only needed to implement once for each of the relatively few UPnP device types.

9.7 Tall Ships' Race

The Tall Ships' Race demonstration, as introduced in Section 1.5.3, is the biggest and most complex demonstration of PalCom technology made during the project [83]. During four days in July 2007, the Major Incidents Overview prototype, developed by the PalCom team at the University of Aarhus, provided overview and supported communication for police, fire brigade and hospital staff in the Aarhus harbour area, where The Tall Ships' Race event took place. In a centrally placed command center, the Topos application [1] showed a 3D view of the area on a large screen. Inside the 3D model, the locations of key personnel were shown in real time, together with photos taken by their mobile phone cameras. Vessels taking part in the sailing competition could be followed in the model as well. Web cameras, and a remotely controllable Axis dome camera, were placed in strategic locations, and their video streams were displayed in the 3D view.

In the Tall Ships' Race set-up, devices communicated using 3G, directional Wi-Fi, Wi-Fi, Ethernet, and RS-232. Some of the communication was done in native formats, and converted to PalCom communication by wrapper services. There were two assemblies, which coordinated the communication:

- Photos taken by the cameras of the mobile phones, and coordinates from their GPSs, were retrieved over 3G, and collected by an assem-

bly. Both types of information were sent to PalCom services for storage and future reference. The assembly also sent the photos to Topos, with attached location information, so they could be shown at the right place in the 3D world.

- Another assembly provided GPS coordinates of the over 100 vessels, obtained from the Automatic Identification System (AIS), to the Topos application for 3D visualization.

The Axis dome camera, which streamed live video to the Topos application, was controlled using the same AxisControllerService as for the SiteTracker, presented in Section 9.2.

9.8 Summary

The examples in this chapter show that the Service Framework and middleware, as presented in Chapter 8,

- have been used by PalCom programmer teams for building prototypes, that have in turn been evaluated by professionals from the police and the fire brigade, by medical staff, and by landscape architects.
- have been used over different network technologies, such as Ethernet/IP, Bluetooth, IR, and short-range radio.
- have been bridged to other frameworks, such as UPnP [114].
- have been used on a resource-constrained platform, in the case of the Sun SPOT [104].
- allow specialized implementations of MAL Layer media managers, and routing between PalCom networks.

In all the examples, assemblies are used for combining and coordinating services, and in most cases also for adding some functionality. This is the case for the image tagging in GeoTagger, for the wind mill visualization in SiteTracker, for the alarm handling in the Incubator, and for the puzzle game in Active Surfaces. By letting the heterogeneous devices around the Incubator offer PalCom services, they could be integrated in a flexible manner, using assemblies. The software on the tiles is organized as PalCom services, which cooperate using assemblies. Unbound services are used for handling computation aspects in several of the scenarios. In Active Surfaces, the game could be changed by exchanging assemblies and unbound services. Some services, such as the AxisControllerService that controls the

dome camera, were re-used in several prototypes, by incorporating them in different assemblies. For the SiteTracker example, a way was shown of structuring systems of assemblies in a hierarchical way, using synthesized services.

The split between computation, expressed in unbound services, and coordination and configuration, expressed in assembly descriptors, has been useful and powerful in the example scenario implementations. Features that have emerged as candidates for extension of the assembly descriptor language are conditional execution, arithmetic and conditional expressions, and support for timers.

Chapter 10

Evaluation

In this chapter, we will evaluate the results of the work presented in this thesis. The evaluation will focus on

1. the support for non-preplanned interaction and ad-hoc combinations in the PalCom architecture,
2. the framework and the middleware, that has been implemented for the developer of PalCom software, and
3. the scalability of the protocols and the architecture.

The evaluation will be based on experiences from use of the software, in PalCom scenarios and in our lab environment, and on reasoning from cases and measurements. We should mention that our work has been part of the large PalCom project, and that this is not an evaluation of the complete project, but of our part in it.

10.1 Ad-hoc combinations and non-preplanned interaction

Our main evaluation method has been continuous use of the software, and demonstration of implemented systems. Different versions of the framework, middleware and browsers have been used throughout the PalCom project, and towards the end of the project they have been demonstrated in PalCom scenarios, as presented in Chapter 9. There have been PalCom devices running services on a JVM or the Pal-VM, and simulated devices running on laptops or desktop PCs, representing some hardware that was not physically available at the time. Some devices have been wrapped by

PalCom services running on connected devices, and there has been an Axis camera with Linux that implemented the protocols directly in C [72]. The different browsers, as presented in Chapter 7, have been the Handheld Browser, the PalCom Developer's Browser, the PalCom Overview Browser and the NICU Browser. The middleware and the framework, supporting assemblies and construction of services, as well as the browsers, are all now part of the open-source PalCom reference implementation, and the main design is stable and working.

Looking at it from a certain angle, it is of course difficult to test some aspects of the ad-hoc combinations and non-preplanned interaction through long-term use by people who, after some time, get quite accustomed to the structure of services and assemblies. In such a setting, is not easy to repeatedly capture the moment where the opportunity emerges for combined use of a set of services, or for interaction with a new service. We have, however, noted on a number of occasions that the assembly mechanism allows much more light-weight—and ad-hoc—compositions, than would have been possible without such a mechanism. Examples of this are where services developed by different groups have been combined in assemblies, such as with the AxisControllerService, which was used in SiteTracker, in the Tall Ships' Race and in a demonstration with the Sun SPOT, and with the ExtendedGeoTagger assembly (Section 9.1), which was constructed at a later time and integrated with the existing, unchanged functionality of the GeoTagger. It is thus possible to combine services that were not created together. In the surveillance scenario with the UPnP bridge (Section 9.6), an assembly was created for combining PalCom devices with devices whose service interfaces were generated automatically, from existing UPnP services.

We have noted some particularly important factors behind this. One is that the assembly is external to the services—that it separates computation from coordination and configuration—and another is that the Discovery Protocol is defined as a standard protocol, known by all PalCom devices. Without such a standard protocol, ad-hoc combinations would not be possible. Other important factors are the possibility to make adaptations of service interfaces in synthesized services, and the possibility to create hierarchies of assemblies, as shown for the SiteTracker in Section 9.2.

We have strived to keep our designed assembly descriptor language as simple as possible. One, perhaps slightly subtle, language feature is the possibility to adapt command flows between services in event handlers, by selecting a subset of parameters, and forwarding them in a differently named command. The less complex alternative, having only the possibility to forward unchanged commands, would effectively put us in the problematic standardization situation discussed in Section 1.2, or would require that the set of services were created together. Examples of command changes during forwarding can be found in most of the implemented Pal-

Com scenarios we have seen, including the ones in Chapter 9, and we see this as a necessary feature of a useful assembly mechanism. Whether our set of assembly descriptor language features is also optimal, giving a sufficiently powerful mechanism, will be discussed further in Chapter 11.

10.1.1 Usability

Of crucial importance, for non-preplanned interaction and ad-hoc combinations, are also the design of the language for service descriptions, and the functionality in browsers for presenting the service descriptions to users. Here, we have the ambition of making the services self-describing through their rendered user interfaces, as a support for experimentation with the services while combing them in assemblies. Another important factor is the possibility to view the set of currently established connections in browsers, for getting a view of the current communication situation, as supported by the Discovery Protocol. These factors are tightly connected with issues about the usability of available tools, in the form of browsers.

As discussed in Section 7.5, the different browsers implemented in the project focus on different user groups, and provide different functionality. The nicest overview of current connections is given by the Overview Browser, which shows them graphically. The Developer's Browser shows connections in a simple list, but has functionality for editing assemblies, and for rendering user interfaces from service descriptions. The rendering, which works the same way as for the Handheld Browser, gives user interfaces that are more or less appealing, depending on the structure of a particular service description and the size of the available screen area. Here, we have discussed giving the service provider the possibility to provide some more layout hints than what can currently be specified for a PalCom service. One simple and very useful related feature, that has been incorporated in the protocols and utilized by the browsers in tool-tip texts, is human-readable help texts, as shown in Figure 5.9. The design of a more sophisticated scheme for layout hints is part of future work.

Another mechanism in browsers, that we have seen as potentially useful, and perhaps necessary for large-scale use, is filtering of the set of discovered devices, services and connections that are shown to the user. Our current basic bordering mechanism is the local network, which may be too large in many cases. One mechanism, which has been used in some versions of the Overview Browser, is to focus on the services and connections that are included in one or a few assemblies, and hide the rest. For extending the set of discovered entities beyond the local network, we have the PalCom tunnels, as discussed in Section 1.6.6.

10.1.2 Palpable challenges

The palpable challenges, as listed in Section 1.4, are related to ad-hoc combinations and non-preplanned interaction, and to the usability issues. They were formulated at the beginning of the PalCom project, and we have used them as a motivation and inspiration for our work. The challenges are very rich, however, and finding solutions for balancing the six pairs of complementing properties has not been the primary goal of the work presented in this thesis. Nevertheless, it is interesting to see that the work done on assemblies and browsers does indeed address all of the challenges in some way.

Visibility and invisibility

Ubiquitous computing brings a degree of invisibility to computing systems, in that the systems blend into the environment. PalCom highlights the need for balancing this with an appropriate degree of visibility, so that the systems remain understandable. Visibility is supported by announcing descriptions of services, including synthesized services of assemblies, and by involving the user in the initial process of setting up a set of connections, that can later be turned into an assembly. Further, it is supported by giving a view of the current communication in announced connections, and by the device awareness, as discussed in Chapter 5. Invisibility is supported by assemblies that work automatically in the background, establishing connections and governing communication between services, and also by the notion of synthesized services, that can hide the inner workings of an assembly and present only an interface for use from the outside.

Construction and de-construction

For construction and de-construction, the notion of the assembly is central in PalCom. We enable construction of assemblies by end users, while requiring only limited pre-defined knowledge of service interfaces. Deconstruction is supported by the interactive browsers, where existing assemblies can be taken apart and changed. This is true for hierarchies of assemblies, constructed by means of synthesized services, and for individual assemblies that combine a set of native services.

Heterogeneity and coherence

The approach to heterogeneity lies in that assemblies can combine services that were not created according to a pre-defined standard, and adapt to the actual service protocols using scripted logic. There is no standardized protocol at the domain level, and the heterogeneity is not restrained. At the

same time, the common, domain-independent Wire and Discovery protocols allow services to be inspected and interacted with in a coherent way. A synthesized service can be used for providing an end user with a coherent view of a set of services.

Change and stability

The high degree of change in pervasive computing systems is moderated in PalCom by mechanisms for increased stability, including assemblies with dynamic bindings that adapt to a fluctuating set of available devices. The assembly remembers its connections and automatically connects its assembled services once they become available. The support for transient devices in the Pacemaker Protocol, presented in Chapter 5, also works for increased stability. Regarding the support for change, the assembly can be adapted by the user for handling changes in the environment, e.g. by rebinding a connection from one device to another.

Scalability and understandability

Scalability is supported at the assembly level, where complex scenarios can be handled by using synthesized services in order to build larger systems from simpler ones. Assemblies can be scaled to compose non-local services, for example over the Internet, through the use of tunnels. The potential need for scoping mechanisms on the network, as discussed above, also belongs here. Understandability is supported by the possibility to group services as composite services in assemblies, and for grouping services in a tree structure on a device. The structure of the assembly mechanism is also sufficiently simple to promote understandability: an assembly can be inspected directly in a browser.

Sense-making and negotiation, user control and deference

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. The rules for the autonomous behaviour of an assembly are defined by the user in the assembly descriptor, and services can be explored directly by the user before building an assembly, so problems can be identified. As a kind of sense-making and negotiation, an assembly can automatically replace a failing main service with a backup service using alternative services, but still under control of the creator of the assembly.

10.2 PalCom for developers

The middleware and the Service Framework have been created as support for developers of PalCom software. Our principal evaluation method for these has been to observe how programmers have worked with them, both in our group in Lund and in other PalCom groups. Our main observation is that the middleware and framework have been used in the implementation of all four browsers, and in the implementation of services and assemblies in all PalCom scenarios presented during the final year of the project, including the ones presented in Chapter 9. Developers in groups outside Lund have worked in the project's shared code repository, and sometimes in local code bases that have linked to the middleware and framework classes in the reference implementation.

Thus, we have some evidence that the middleware and framework is useful for developers. In order to get a more precise view, we will try to sort out how different development tasks can be performed:

- In order to implement a simple service, the normal way is to subclass the class AbstractService in the Service Framework. A common practice has been to start from one of the simple example services in the reference implementation (often a service named EchoService that simply echoes received commands back to the sender).
- For implementing a small device, subclassing of the classes AbstractDevice and AbstractService can be used, if the device can host a JVM or a Pal-VM. This was done for the Sun SPOT, which has a JVM, where the built-in sensors and actuators of the device were exposed as PalCom services (see Section 9.5). If there is no compatible virtual machine, it is also possible to implement support for the PalCom protocols directly in C or some other language, as was done in the project [72], but that is a substantially more complex task.
- Adding PalCom browsing capability to a device is straightforward for a device that supports desktop Java, where one of the available browser applications can be installed. If the device supports Java Swing [106], the Handheld Browser is also available in a Swing component that can be included in the graphical user interface of another application. Implementing a browser from scratch, on top of the Service Framework, is also possible but a bigger task.
- An assembly can be added to any device that hosts an assembly manager, as available in the reference implementation. Once the assembly manager is running, assemblies can be uploaded to the manager remotely, and the device does not need to have an assembly editor of its own, or even persistent storage. For editing assemblies, the PalCom Developer's Browser can be run on a laptop or a desktop computer.

- Finally, a task that requires deeper knowledge of the middleware is to add a new media manager in the Media Abstraction Layer, as was done for the tiles simulator (Section 9.4.3), and for the Sun SPOT. That can be done by subclassing the class *AbstractMediaManager*, but it also involves understanding of the PalcomThreads library (Section 8.3), and of base libraries below the middleware.

To conclude, the more common tasks of a PalCom developer have direct support. For the more uncommon tasks deeper knowledge is needed, but there are examples to look at in the reference implementation.

10.3 Scalability

Our evaluation of scalability considers several aspects of the PalCom protocols and architecture:

1. networks with large numbers of devices,
2. response times in the system, which have to be sufficient for interactive work, and
3. the support for PalCom on resource-constrained devices.

All of these are important to demonstrate for the architecture and the mechanisms proposed in the thesis.

10.3.1 Large numbers of devices

In order to show that the architecture scales up to large numbers of devices, we have to show that the load on the network is kept down, while the requirements of support for transient devices are kept (devices frequently joining and leaving networks, see Chapter 5), and while the response times in the system are sufficient for interactive work. PalCom is primarily designed for systems where a human is in the loop, and not for hard real-time requirements. The protocols 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.

For the load on the network given by a PalCom system, the heartbeat mechanism in the Pacemaker Protocol is central. The heartbeat communication takes place regardless of any interaction between services, and thus has to be kept compact in order for the protocols to scale. The approach of Zeroconf [99], which has no periodic idle packets at all, is not sufficient for the PalCom protocols, because the user has to become aware when a device disappears.

There are different timing requirements for discovery of appearing and disappearing devices in PalCom:

- When a new device enters the network, the other devices have to become aware of it within a *short* time, and it has to become aware of them.
- When a device leaves the network, the remaining devices have to become aware that it has left within a *known* time, even if the device leaves without announcement (it may crash or just leave the network range).

The differences in timing requirements come from differences at the user level. When a device enters a network, a user is often involved, inspecting devices and services in a browser or otherwise. Therefore, discovery has to be more or less instant. When devices disappear, on the other hand, they are usually not in direct use. The requirement, as we have seen in the PalCom scenarios, is instead that the time of discovery of device disappearances has to be under control of the involved devices, and possible to vary depending on the application, as with the heartbeat frequency that is controlled by the most eager device.

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 the network range.

In the Discovery Protocol, there will be direct notification to all other devices in cases 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 heartbeat 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 heartbeat procedure. This time is thus determined by the applications, and not by the Discovery Protocol itself. It can, e.g., 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. In summary, for all the five cases, the response time requirements are met for the heartbeat mechanism.

Regarding the requirement of keeping the load on the network down, it is a PalCom requirement that there is no central server (requirement 14 on page 76). 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 means at least 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, n^2 messages would have been needed.

For distributing more detailed information than the presence information distributed through the heartbeats, two basic design options for the discovery protocol are broadcast and unicast messages. We have chosen to use a unicast request-reply scheme, combined with caching, as discussed in Chapter 5. Devices send unicast requests for descriptors they are interested in, and unicast replies are sent back. These, potentially extensive, pieces of information go only to those devices that are explicitly interested. On a shared-medium local network, the costs of broadcast and unicasts are the same at the very lowest level: both occupy the medium. But unicasts are filtered out, by hardware or by low-level network software, for all but the intended receiver, and only processed further there. We concluded that the potential advantage of listening in on broadcasts for the more detailed information, as is done for the heartbeats, would be eliminated by a large amount of unwanted processing of messages about services that some devices, typically small ones, are not interested in. Therefore, the detailed information is obtained through unicast request messages. The HeartBeat message has been kept as compact as possible, with a special message node, as illustrated in Figure 5.3. For very constrained environments, the broadcasting of heartbeats can be tailored further.

In this context, we should also mention that, as for the ad-hoc combinations and non-preplanned interaction discussed in Section 10.1, our continuous use of the software has also given us hints about the scalability. One example is the PalCom project reviews in March and December 2007, where many PalCom devices and services, implemented for different demonstration scenarios, executed in the same local network. On the latter occasion, the implementation of the caching mechanism in the protocols had been finished and stabilized, which gave a remarkable increase in the stability of discovered devices and services in browsers.

10.3.2 Execution on resource-constrained devices

In order to evaluate the support for execution on small devices, our first observations are that in the PalSpot project, a Sun SPOT worked as a Pal-

Com device, running the built-in JVM (see Section 9.5), and in the project [72], PalCom services were implemented on an Axis network camera. Both of those devices are fairly small and resource-constrained, and worked together with PalCom browsers and services in assemblies.

When looking at such examples of implemented scenarios, we evaluate the architecture and protocols by evaluating our implementation of them. This is also the philosophy behind a number of measurements that we have made: we measured execution times, processor load and response times for communication between PalCom services and assemblies, implemented using the middleware and Service Framework. The services and assemblies ran on Linux PCs in a 100 Mbit/s Ethernet LAN.[1]

We relate the measurements to times for "minimal" programs, written in C and Java, that send similar amounts of data over the network. The minimal programs do no processing of data, either at the sender or the receiver side, while PalCom services pack and unpack commands according to the protocols described in Chapter 5, among other things. Thus, the minimal programs are not relevant to use as direct references for wanted performance. Instead, we use them to relate the times of PalCom processing to the times for "pure communication" on the hardware used.

Sending commands directly between services

The first measurement set-up was according to Figure 10.1, with two computers running one PalCom service each. In this test, and in the following tests with PalCom services, the services were implemented using the Service Framework and executed on a JVM. The service Sender sent commands to the service Receiver with different periods, i.e. with different delays between consecutive commands. The UDP packets sent, containing the commands, were approximately 244 bytes each. Figure 10.2 shows the results for CPU load and packet counts (command counts). Period 0 was tested by doing a *yield* between each command, letting other PalcomThreads execute, but without any other waiting. With that maximum sending rate, around 4500 commands could be transported per second. The CPU loads are plotted in Figure 10.3. We see that periods of 1 and 2 ms gave some load on the receiver, while for 0 ms both the sender and the receiver had 100 % load. For a reasonable CPU load of around 20 %, the PalCom services could transfer about 1000 commands per second.

In order to get an idea of the capacity of the network and the lower networking layers on the computers, we made measurements with two minimal C programs, one which sent out UDP packets at maximum speed, and one which received the packets. The results are shown in Figure 10.4, and

[1] AMD Athlon 64 3800+ X2 computers with 2 GHz processors and 1 Gb memory, running Debian Linux.

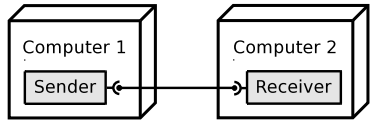


Figure 10.1: *Set-up for the test of command sending from one service to another.*

Period (ms)	CPU Sender (%)	CPU Rec. (%)	t (s)	Pkt sent ($\cdot 10^3$)	Pkt rec. ($\cdot 10^3$)	Rec. rate (pkt/s)
100	0	0	100	1.00	1.00	10
10	0	0	100	10.00	10.00	100
3	0	0	99	33.00	33.00	333
2	0	2	99	50.00	49.51	500
1	0	20	99	100.00	99.16	1002
0	100	100	22	100.00	99.52	4524

Figure 10.2: *Results for the command sending test.*

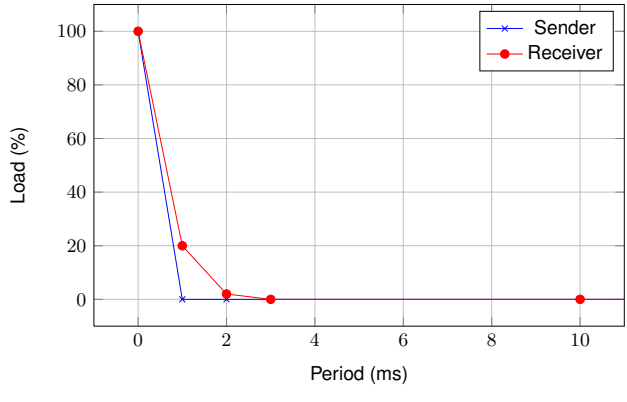


Figure 10.3: *CPU loads in the command sending test.*

Pkt size (bytes)	**CPU sender (%)**	**CPU rec. (%)**	**Pkt sent ($\cdot 10^6$)**	**Pkt rec. ($\cdot 10^6$)**	**t (s)**	**Rec. rate ($\cdot 10^3$ pkt/s)**
1500	5	2	0.50	0.50	65	7.69
1000	5	3	0.50	0.50	42	11.90
500	8	4	0.50	0.50	23	21.73
400	10	5	1.00	1.00	37	27.00
300	13	5	1.00	1.00	29	34.45
200	19	8	1.00	1.00	21	47.53
100	28	12	2.00	1.99	26	76.72
50	50	18	2.00	1.99	19	104.80
30	80	50	2.00	1.99	15	132.79
20	99	65	3.00	2.92	21	138.84
10	99	70	3.00	2.96	22	134.48
1	99	70	3.00	2.93	21	139.29

Figure 10.4: *Values for packet sending test with minimal C programs.*

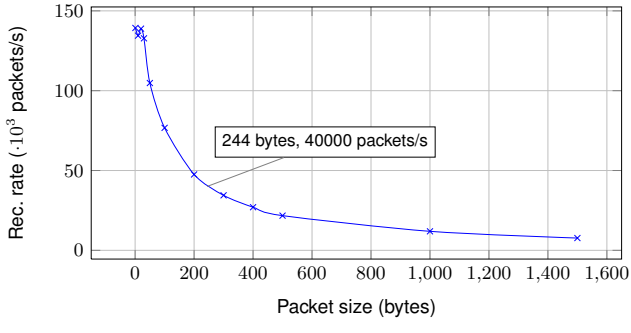


Figure 10.5: *Packet receival rates for the C packet sending test.*

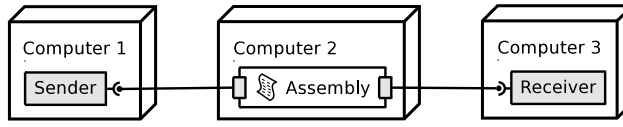


Figure 10.6: *Set-up for the assembly test.*

packet receival rates for different packet sizes are plotted in Figure 10.5. At the packet size used in the PalCom command sending test, 244 bytes, approximately 40000 packets could be transferred per second. From this we conclude that with the PalCom services, around $4500/40000 \approx 11\%$ of the time was spent on "pure communication", and the remaining 89% was spent on PalCom processing. Within those 89%, we have the possibility to optimize the processing in order to increase throughput.

Sending commands via an assembly

The next test was done for measuring the performance of a system with a PalCom assembly. The set-up was as shown in Figure 10.6, with three computers, where two had one service each and one executed an assembly. The assembly forwarded commands from the Sender service to the Receiver service. The Sender sent with different periods, and the 0 ms period was handled the same way as above, using a *yield*. The UDP packets had a size of approximately 244 bytes. The measurement values are shown in Figure 10.7, with the CPU load values plotted in Figure 10.8. We see that the assembly manager on Computer 2 gets a higher load than both the sender and the receiver. At the period of 0 ms it reaches 100 %, and loses the majority of the packets. Down to 1 ms, the values are similar as for the results with only services in Figure 10.2. It was not possible to test with a period between 0 and 1 ms, because the time resolution of waits in the framework is 1 ms. The reason that the assembly manager has to work harder is probably simply that it has to perform both a receive and a send for each command. Looking at the CPU loads, we see that it is reasonable to use the current implementation for systems where assemblies transfer up to about 500 commands per second (20 % load on the assembly manager).

Response times

In addition to receival rate, another interesting measurement is response times, i.e. the round-trip time from a command is sent, until a reply command comes back. This was measured using the three set-ups shown in Figures 10.9 through 10.11. In set-up A, shown in Figure 10.9, the service Tester, which sends out the first command and measures the response time,

Period (ms)	CPU Sender (%)	CPU Rec. (%)	CPU Ass. (%)	t (s)	Pkt sent ($\cdot 10^3$)	Pkt rec. ($\cdot 10^3$)	Rec. rate (pkt/s)	Pkt lost (%)
100	0	0	0	50	0.50	0.50	10	0
10	0	1	1	50	5.00	5.00	100	0
5	0	0	3	150	30.00	30.00	200	0
3	0	1	13	300	100.00	100.00	333	0
2	0	1	25	148	75.00	74.01	500	1.32
1	0	15	60	150	150.00	149.47	996	0.35
0	90	3	100	12	100.00	15.90	1325	84.1

Figure 10.7: *Values for the assembly test.*

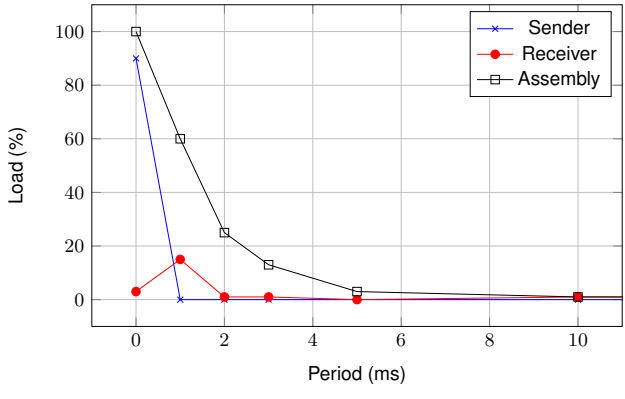


Figure 10.8: *CPU loads for the assembly test.*

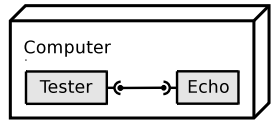


Figure 10.9: *Response times test set-up A.*

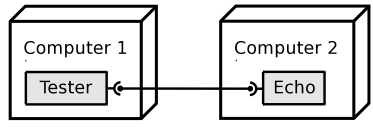


Figure 10.10: *Response times test set-up B.*

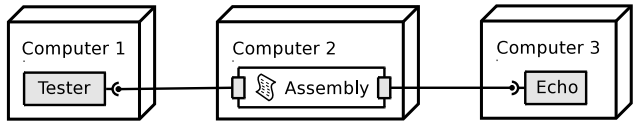


Figure 10.11: *Response times test set-up C.*

and the service Echo, which replies as fast as it can, ran on the same computer. In set-up B they ran on different computers (Figure 10.10), and in set-up C an assembly on a computer in the middle forwarded both the first command and the reply (Figure 10.11). The results are shown in Figure 10.12. The average response times were approximately 1400 μs for set-up A, 1100 μs for set-up B, and 2800 μs for set-up C (it was thus not faster to run both services on the same computer).

Response times for minimal C and Java programs, used in set-ups corresponding to set-up B, are shown in Figures 10.13 and 10.14, and plotted in Figure 10.15. We see that for the packet size of 244 bytes, as used with the PalCom services, the minimal programs have a response time of about 135 μs. As pointed out above, this cannot be directly compared with the 1100 μs for set-up B with PalCom services, because the PalCom services do processing that the minimal programs do not. Still, it serves to give an estimate of the time used for "pure communication". Figure 10.15 shows that the difference between C and Java is very small, so the fact that the PalCom services run on a JVM, and are not implemented directly in C, is not an important factor for the performance at this level.

Set-up	Runs	Pkt size (bytes)	Average resp. time (μs)	CPU Tester (%)	CPU Echo (%)	CPU Ass. (%)
A	10000	224	1427	100	100	-
B	50000	224	1051	50	45	-
C	50000	224	2767	19	17	60

Figure 10.12: *Values for response times test with PalCom services and assemblies.*

Runs	Pkt size (bytes)	Average resp. time (μs)	CPU tester (%)	CPU echo (%)
100	500	218	5	3
100	244	134	8	4
200	100	88	10	7
1000	10	61	16	10
1000	1	60	16	8

Figure 10.13: *Values for response times test with minimal C programs.*

Runs	Pkt size (bytes)	Average resp. time (μs)	CPU tester (%)	CPU echo (%)
100	500	223	7	4
100	244	137	9	5
200	100	90	15	5
1000	10	65	18	12
1000	1	64	19	12

Figure 10.14: *Values for response times test with minimal Java programs.*

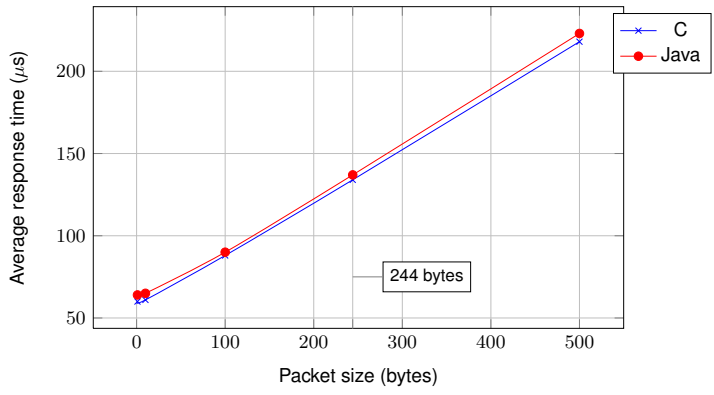


Figure 10.15: *Response times for minimal C and Java programs.*

				Average per run (μs)		
Program	**Runs**	**Pkt size**	**CPU (%)**	**User**	**System**	**Total**
B	$5.0 \cdot 10^4$	244	43	436	244	1551
MinJava	$1.0 \cdot 10^6$	244	10	3	11	140

Figure 10.16: *Values for measurements with the Unix* `time` *command.*

Execution times

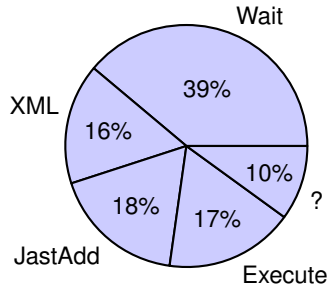
In order to get an idea of how the time for processing is spent in a PalCom service, we measured a number of executions of set-up B with the Unix `time` command, and compared with measurements for the minimal Java program. The results, as shown in Figure 10.16, show that we got 1551 μs total time per run for set-up B in that test, and 140 μs for the minimal Java program. The column System shows the time used by the operating system. The extra System time of $244 - 11 = 233$ μs for set-up B, compared to the minimal Java program, is mainly used for context switches between threads. Those 233 μs are a substantial time, compared to the User time of 436 μs for set-up B, where the service executes at the Java level.[2]

In order to investigate the execution times more closely, time logging was inserted at certain points of the Tester service and Service Framework code that executed during the handling of a received reply, until it was delivered at the level of service code. The execution intervals between the logging points were categorized according to the main activity in that interval. The results for one run are shown in Figure 10.17. The total time for

[2]The part of the total 1551 μs that is not covered by User time or System time is spent waiting for responses over the network.

Action	Time (μs)
Wait	122
XML	50
JastAdd	55
Execute	54
?	31
Total	312

(a)



(b)

Figure 10.17: *Times spent for different activities during the processing of a received command (a), and the percentages (b).*

Thread to follow	JCS1 (μs)	Sch (μs)	JCS2 (μs)	Total (μs)
mediaMgr	5	6	8	19
routingMgr	7	5	7	19
commMgr	5	6	7	18
connMgr	5	5	7	17
abstractService	6	5	6	17
tester	8	7	8	23
Total				113
Average				18.83

Figure 10.18: *Execution times during context switches in the processing of a received command.*

the handling of the received reply was 312 μs. The intervals were tagged as *Wait*, which means context switching between PalcomThreads, *XML*, which means XML parsing, *JastAdd*, which means creating of a JastAdd AST from the received command (see Section 8.2.1), and *Execute*, which means "other execution". One interval, where we could not determine the main activity, was tagged with a question mark. We see, again, that the waiting for context switches takes a large part of time, 39 %.

Figure 10.18 shows detailed measurements for the six context switches between PalcomThreads that took place during the handling of a received command in one run. The first four of the threads handle different layers in the communication stack, the thread *abstractService* is in the Service Framework, and the thread *tester* is in the Tester program. The total time of 113 μs in this run corresponds to 122 μs for *Wait* in Figure 10.17. The context switches between PalcomThreads are carried out as follows: each PalcomThread executes in a coroutine, which is supported natively on the Pal-

VM, and which is executed by a dedicated Java thread on the JVM (the JVM was used in this test). In the context switch, the coroutine hands over to the PalcomScheduler, which is a Java thread on the JVM, and which chooses a new coroutine (Java thread) and schedules it. I.e., in each PalcomThread context switch there are two Java thread context switches, shown as *JCS1* and *JCS2* in Figure 10.18, and execution by the PalcomScheduler, shown as *Sch*. It can be seen that the average time for a context switch between PalcomThreads on the JVM is 19 μs, and with six threads that adds up to 113 μs during the handling of a received command.

10.3.3 Conclusion

As a conclusion of the treatment of the measurement results, we see that in the current reference implementation we have a maximum throughput of about 4500 commands per second on the computers used in the tests, and response times around 1 ms directly between services (Figure 10.12). For reasonable CPU loads, around 20 %, we got a throughput of about 1000 commands per second. With an assembly in the middle we got response times around 2.8 ms, and could transfer about 500 commands per second. Hence, we see the reference implementation as reasonable to use in embedded settings, with some optimizations. This view is supported by the implementation demonstrated on the Sun SPOT. Regarding the protocols as such, the C implementation on the Axis camera also supports this view.

The pie chart in Figure 10.17 (b) gives a clue about where the bottlenecks are, that should be optimized first. Based on the structure of the system, we believe that the handling of a received command is representative for the execution in general, during such execution that involves sending and receiving of commands. The PalcomThread context switches are expensive on the JVM, largely because of the coroutine emulation solution with dedicated Java threads, and the relatively high cost of Java thread context switches. Here, one option would be to try to optimize the PalcomThreads for execution on the JVM, perhaps by using more of the native Java scheduling mechanisms. Another option, more directly targeted at this problem, would be to reduce the number of PalcomThreads used in the middleware and Service Framework—thus reducing the number of context switches—and use more callbacks rather than event communication between threads in the implementation. Two other large pieces of the execution are the XML parsing, which is done by a third-party XML parser, and the JastAdd handling. Both of those could be optimized by implementing a more specialized solution, tailored to the grammar for PalCom messages.

10.4 Summary

In this chapter, we first evaluated the proposed architecture from the perspective of ad-hoc combinations and non-preplanned interaction, as discussed in Chapter 1. The overall conclusion is that the assembly has been very useful as a light-weight mechanism for combining services. This could be seen in a number of usage examples from the PalCom project. Important factors are that the assembly can adapt command flows between services, and that the Discovery Protocol is defined as a standard, domain-independent protocol, known by all PalCom devices.

PalCom services are self-describing, as a basis for experimentation when constructing and adapting assemblies. The usability of available browsers is an important factor in order to support this, and we have discussed possibilities for additional layout hints for service descriptions. In this context, the visualization of established connections, that give a view of the current communication situation, is also important.

Further, we evaluated the implemented Service Framework and middleware from a developer perspective. Based on usage by ourselves and by other PalCom groups, we concluded that they have supported the development of many useful PalCom services, with the most elaborate support for the most common tasks.

Finally, we looked at performance issues of the protocols and the implementation, and identified areas suited for optimization. For the heartbeat mechanism in the Pacemaker Protocol, whose performance is important because of its use of idle packets, we saw that the load on the network is minimal, given the need for every device to supply information every heartbeat period, and to discover new devices instantly. Only n messages are sent per heartbeat period, in a network with n devices. The communication between services and assemblies was also measured, and we concluded that performance is sufficient for use on embedded devices. On the hardware used for testing, the current implementation has the following practical limits, with reasonable CPU loads:

- For commands sent directly between services, throughput is about 1000 commands per second, and we get a response time of 1 ms.
- For commands sent via an assembly, we get about 500 commands per second, and 2.8 ms response times.

Chapter 11

Conclusions and future work

This thesis has presented ideas about non-preplanned interaction and ad-hoc combinations in pervasive computing environments. The goals have been to enable direct use of services when discovering new devices, and to have a lightweight service composition mechanism that avoids dependence on domain-specific standards. The usefulness of the ideas has been demonstrated

- by designing the notions of service and assembly in PalCom, where an assembly combines a set of services on devices,
- by defining domain-independent discovery and communication protocols that support services and assemblies,
- by designing a language for assembly descriptors, that is edited in interactive browsers and executed by assembly managers,
- by implementing support for development of services, by means of a Service Framework and middleware, and
- by evaluating the implementation through example scenarios that have been built in the PalCom project.

The evaluation has shown the potential of assemblies as a mechanism for ad-hoc combinations. In the scenarios, of which several were developed in cooperation with prospective end users, assemblies were used for combining and coordinating services by adapting command flows, for adding functionality to what was given by the individual services themselves, and for combining heterogeneous devices. Some services were re-used in

several assemblies, and unbound services were used for adding computational power.

Further, the evaluation showed that the implemented Service Framework and middleware have been useful for programmers, and that the architecture is useful in practice, in that it scales to large networks and small devices. The latter was shown by analyzing the protocols and the reference implementation.

11.1 Summary of the architecture

In the proposed architecture, services execute on devices. The services describe themselves in textual descriptions, where no domain-level standardization or ontology is used. This makes the services self-describing, and supports end-users in exploring and experimenting with them. There is a generic mechanism for rendering of user interfaces in browsers, using the service descriptions as blueprints, and browsers are also used for combining services in assemblies. When constructing an assembly, the service description is used in a dual role as a programmatic interface.

An assembly defines a particular configuration of services on devices, and connections between them. It builds on the property that connections are first-class. They are announced explicitly, and can be established between two services from a third device. Further, the assembly can specify dynamic bindings to services, that are updated automatically during the execution of the assembly, and script logic that governs coordination of the services by forwarding commands.

The assembly is external to the included services, and separates *configuration* and *coordination* from the *computation* in the services. This enables end users in the loop, without detailed programming needed for configuration and coordination. It gives loose coupling, and supports handling of changes in service interfaces. The script can be used to mediate between services that were not designed to work together, thus enabling ad-hoc combinations.

Assemblies can be edited in interactive assembly editors in browsers. They can specify new services, so called synthesized services, and can be further aggregated by other assemblies in a hierarchical way. The assembly script defines event handlers that react upon received commands, and forward them or save parameter values in variables. An event handler is atomic, and the whole behaviour of the assembly is non-blocking. This fits with the behaviour of services, which communicate asynchronously in a peer-to-peer fashion. Where involved computations are needed, unbound services can be incorporated for specific computations. This helps keeping the script language simple, and computation separated from configuration and coordination.

The architecture defines protocols that let PalCom devices discover each other and communicate. The protocols support communication across different networking technologies, by defining a common wire format, abstractions from concrete network addresses, and routing between technologies. The use of DeviceIDs, that address devices and not their individual network interfaces, makes it possible to support routing and mobility in the Routing Layer, without affecting the Service Layer. Distant PalCom networks can be connected by the use of tunnels. There is support for connections with different distribution schemes (one-to-one, one-to-many and many-to-many), for reliable communication, and for sending of large messages.

The Discovery Protocol is defined as a standard protocol, known by all PalCom devices, which is a prerequisite for ad-hoc combinations. It distributes name, addressing and versioning information, and descriptions of devices, services and connections in a lightweight XML format. Each device and service has a unique identifier, which contains versioning information. This is a cornerstone of the strict versioning scheme, which supports controlled updates of services and assemblies.

At the centre of the Discovery Protocol is a heartbeat mechanism, known as the Pacemaker Protocol, which puts minimal load on the network, given our requirements of support for transient devices, and given that there is no central directory. The heartbeats are combined with caching at upper levels, for keeping the load on the network down. These two techniques are the base of our claim that PalCom support can be implemented on small devices. Above the Discovery Protocol, there is also a default protocol for Service Interaction, defining how commands are packaged and sent over connections.

11.2 Future work

Our planned and ongoing continued work on the architecture and the implementation concerns many different aspects. Optimizations of the framework code focuses on thread handling and on the parsing of messages, as indicated by the measurements in Section 10.3.2. We plan to implement additional browsers, where editing of assemblies can be done in a more graphical way than with the PalCom Developer's Browser, and browsers for handheld devices that are more powerful than the Handheld Browser. Related to the usability issues, we also plan to design a language for layout descriptions, coupled to the service descriptions, that gives hints for the rendering of user interfaces in browsers.

We plan to utilize the support for versioning in the protocols, as presented in Chapter 5, for developing more elaborate support for updates of services and assemblies at the service manager, assembly manager and browser lev-

els. In particular, this regards support for epidemic updates of assemblies, that spread between users when they come in contact with each other. In the protocols used between service managers and assembly managers, we intend to make further use of radiocast and groupcast communication, as explored for the Active Surfaces prototype in the Groupcast tiles experiment (Section 9.4.4). Potentially, this will decrease the bandwidth usage, and also make the interaction more flexible.

For the discovery and communication protocols, we see the most interesting issues in the areas of filtering mechanisms for discovered devices and services, based on other scoping principles than the edges of the local network, and refined support for discovery in aggregated networks, connected by tunnels. In such networks, central directories may be useful for keeping the amount of broadcasted traffic down.

At the level of service descriptions, ideas have emerged for letting the services describe more about their behaviour, e.g. how different commands are related. Some pairs of commands may form request-reply pairs, where one in-going command leads to another out-going command, some outgoing commands may be sent out spontaneously, etc. If the services could specify such relations in their descriptions, that would be helpful for humans that use the services or combine them in assemblies, and possibly also for tools that perform automated analyses of systems, based on descriptions of services and assemblies.

11.2.1 The assembly descriptor language

A central issue in our continued work is the development of the assembly descriptor language. Finding out what constructs it should support, and how sophisticated they should be, is crucial in order to support ad-hoc combinations at the right level. This depends on what kinds of scenarios and functionality we will target. Features that have emerged in the PalCom scenarios, as candidates for extension of the assembly descriptor language, are algorithmic constructs such as conditional execution, arithmetic and conditional expressions, and support for timers. A higher degree of sophistication gives more power, but may make it more difficult for end users to work with assemblies. In connection to this, there is the balance of what can and should be delegated to unbound services. We want to continue to be able to express the necessary coordination in the assembly descriptor.

Assemblies are transparent to the PalCom system as a whole, in the sense that services being part of an assembly may still be discovered and used by other services and assemblies. There may be situations, however, where services are constructed specifically to be part of a assembly. This is often the case, e.g., for unbound services. It is future work to put mechanisms in place in the language, which let the assembly delimit access and discovery of the participating services. In this way, assemblies can provide scoping

and encapsulation, further supporting a scalable programming model. A useful feature would also be to allow dynamic sets of devices and services in assemblies, where the number of participants is not known at assembly design time. One approach to this is proposed for PalCom assemblies by Brønsted et al. [15]. That paper also presents a mechanism for decentralized execution of assemblies, where different parts of an assembly descriptor are handled by assembly managers on different devices. That is a more automated process than the hierarchies of assemblies with synthesized services that we have presented here, and aims at reducing the network traffic by avoiding a central device where all commands pass. A similar solution has been implemented for Web Services by Chafle et al. [17].

In the current assembly descriptor language, the interface between services and assemblies consists solely of asynchronous commands. In some cases this gives increased complexity, when event handlers have to be added for handling replies from unbound services that belong to request-reply sequences. An interesting issue to look at would be to investigate support for some synchronous interaction with unbound services on the same device, where the communication does not go over the network.

Another interesting area of exploration would be to add execution state to assemblies, as used, e.g., in the BPEL language [80]. That could make the event flow clearer, and would let the assembly react to commands in different ways depending on its state. If that is added, the assembly must not risk to end up in a blocked state, because of missed commands or similar. This is connected to added support for error handling, and to the use of timeouts when waiting for received commands. One option in this direction would be to adopt parts of the Statecharts language [49], which has support for substates.

Appendix A

Message node types

The PalCom Wire Protocol defines a message format based on nodes, as described in Chapter 5. This appendix briefly presents the different types of nodes, which can be header nodes and data nodes. Each message has one or more header nodes and an optional data node:

- `Message ::= HeaderNode+ [DataNode]`

The fundamental layout of a message node is as follows, with three parts separated by semicolons:

- `MessageNode ::= <F>;<L>;<data>`
  - *<F>* is a one-byte format identifier.
  - *<L>* is the length of *<data>* (using ASCII characters).
  - *<data>* is the contents of the node, which depends on the *<F>*.

A.1 Data nodes

There are two types of data nodes, for single messages and multi-part messages. These have format bytes 'd' and '+' (as shown in bold below):

- `DataNode ::= SingleMsg | MultiPart`
- `SingleMsg ::=` **`d`**`;<L>;<data>`
  - *<data>* is *<L>* bytes of application data.
- `MultiPart ::=` **`+`**`;<L>;<data>`
  - *<data>* is a sequence of bytes, containing zero or more messages, concatenated directly after each other.

A.2 Header nodes

Each message contains one or more header nodes. They can be of the following types:

- `HeartBeat ::= h;<L>;<CacheNbr>;<DeviceStatus>`

  A heartbeat in the Pacemaker Protocol (Section 5.4.3).

  - `<CacheNbr>` is the device cache number.
  - `<DeviceStatus>` is the device status (Section 5.5.3).

- `HeartBeatAck ::= H;<L>;<CacheNbr>;<DeviceStatus>`

  A reply to a HeartBeat. Contains the same fields as HeartBeat.

- `HBInfoRequest ::= i;0;`

  A unicast request for obtaining the DeviceID of a device (Section 5.4.3).

- `HBInfoReply ::= I;<L>;<DeviceID>;<DiscoverySel>`

  A reply to an HBInfoRequest.

  - `<DeviceID>` is the DeviceID.
  - `<DiscoverySel>` is the selector used by the device for sending and receiving Discovery Protocol messages.

- `HeartAttack ::= X;0;`

  Broadcasted when a device is about to make an orderly shut-down (Section 5.4.3).

- `FormatVersion ::= v;<L>;<WPVersion>;<DPVersion>`

  Protocol version information sent together with an HBInfoReply during discovery. If the receiving device does not support the given protocol versions, it will not communicate further with that device.

  - `<WPVersion>` is the Wire Protocol version (currently “W2”).
  - `<DPVersion>` is the Discovery Protocol version (currently “D2”).

- `RoutingS ::= s;<L>;<ShortID>`

  Node added to routed messages by PalCom routers, pointing out the sender of the message (Section 5.4.4).

  - `<ShortID>` is the ShortID assigned to the sender.

- `RoutingR ::= r;<L>;<ShortID>`

  Node added to routed messages by PalCom routers, pointing out the receiver of the message (Section 5.4.4). `<ShortID>` as for RoutingS.

- `Connection ::= c;<L>;<f>;...`

  Node used in messages sent over connections (Section 5.4.5). There are different variants, depending on the *<f>* tag (shown in bold):

  - `Open ::= c;<L>;o;<SelR>;<SelS>`

    A request for opening a unicast connection.

    * *<SelR>* is the selector where the receiving device listens for connection requests.
    * *<SelS>* is the selector to be used by the sending device for this connection.

  - `OpenReply ::= c;<L>;p;<SelS>;<SelN>`

    A reply to an Open.

    * *<SelS>* is the selector used by the requester (same as *<SelS>* for the corresponding Open).
    * *<SelN>* is a newly allocated selector to be used by the sending device for this connection.

  - `Close ::= c;<L>;c;<SelR>`

    A request that the connection should be closed.

    * *<SelR>* is the selector associated with the connection on the sending device.

  - `Unicast ::= c;<L>;m;<SelN>`

    Used in a message over a unicast connection. The message contains a data node with message data.

    * *<SelN>* is the selector identifying the receiver.

  - `Radiocast ::= c;<L>;b;<SelS>`

    Used in a radiocasted message. The message contains a data node.

    * *<SelS>* is the used selector on the sending device.

  - `ReOpen ::= c;<L>;r;<SelR>`

    Used in a response to a Unicast, informing that the connection associated with *<SelR>* is no longer available.

- `Reliable ::= R;<L>;<Seq>`

  Node used in a message sent over a reliable connection (Section 5.4.7).

  - *<Seq>* is a sequence number.

- `AckMessage ::= A;<L>;<Seq>`

  An acknowledgment that message number *<Seq>* has been received over a reliable connection (Section 5.4.7).

- `ResendMessage ::= B;<L>;<Seq>`

  A request to resend message number *<Seq>* over a reliable connection (Section 5.4.7).

- `Chopped ::= -;<L>;<MessageID>;<Part>;<Parts>`

  Node used in a message that is part of a larger message (Section 5.4.7).

  - *<MessageID>* identifies the large message.
  - *<Part>* is the index of this part.
  - *<Parts>* is the total number of parts.

- `SingleShot ::= S;<L>;<SelR>;<SelS>`

  Node used in a single-shot message (not sent over a connection).

  - *<SelR>* is the selector on the receiving device.
  - *<SelS>* is the selector on the sending device.

- `LocalGroupMessage ::= P;<L>;<GroupID>`

  Node used in a groupcast message, for groupcast connections that are local to one network (Section 5.4.5).

  - *<GroupID>* is the GroupID.

- `GroupMessage ::= G;<L>;<GroupID>`

  Node used in a groupcast message, for groupcast connections that work across routers (Section 5.4.5).

  - *<GroupID>* is the GroupID.

- `GroupJoin ::= g;<L>;<GroupID>`

  Used for joining a groupcast group, for groupcast connections that work across routers (Section 5.4.5).

- `GroupLeave ::= q;<L>;<GroupID>`

  Used for leaving a groupcast group that has been joined using a GroupJoin.

Appendix B

Descriptor grammars

This appendix contains grammars for the descriptors presented in Chapters 5 and 6. They are abstract grammars, as used by the JastAdd tool [36]. JastAdd is used in the PalCom reference implementation, and the grammars describe ASTs that are built at run-time. There are factory methods that parse XML and build the ASTs, and a JastAdd modular aspect that unparses the ASTs to XML. For the assembly descriptors, there is also an unparser to the concrete syntax used in this thesis.

B.1 Devices, services and connections

The following is the grammar for descriptors of devices, services and connections, and for request and reply messages in the Discovery Protocol:

```
/*** Device and service descriptors (Section 5.5.2) *******/
PRDDevice ::=
  [DeviceID] <Name:String> <DeviceVersion:String>
  <DeviceStatus:byte> DiscoverySelector:Selector
  RemoteConnectSelector:Selector;

PRDServiceList ::=
  ParentLocalSID:LocalSID PRDServiceListItem*;

abstract PRDServiceListItem ::= <Name:String>;

PRDService: PRDServiceListItem ::=
  LocalSID <Distribution:byte> <HasDesc:boolean>
  <RemoteConnect:boolean> <Protocol:String>
  <Reliable:boolean> <VersionName:String>
  <ServiceHelpText:String> [GroupID] [Topic];
```

```
PRDSubList: PRDServiceListItem ::=
  ListLocalSID:LocalSID <Kind:byte>;

/*** Connection descriptors (Section 5.5.2) ***************/
PRDConnectionList ::= [DeviceID] PRDConnection*;

abstract PRDConnection ::= ;
PRDUnicastConnection: PRDConnection ::=
  LocalSID1:LocalSID LocalSID2:LocalSID;
PRDRadiocastConnection: PRDConnection ::=
  SenderLocalSID:LocalSID ListenerLocalSID:LocalSID;
PRDGroupcastConnection: PRDConnection ::=
  LocalSID <GroupID:String>;
PRDBroadcastConnection: PRDConnection ::=
  LocalSID <Topic:String>;

/*** Service descriptions (Section 5.5.2) *****************/
ServiceDescription: GroupInfo ::= LocalSID ControlInfo*;
abstract ControlInfo ::= <ID:String> <Help:String>;
GroupInfo: ControlInfo ::= ControlInfo*;
CommandInfo: ControlInfo ::=
  <Direction:String> <Name:String> ParamInfo*;
ParamInfo: ControlInfo ::=
  <Type:String> <Name:String> <DataRef:int>;
StreamInfo: ControlInfo ::=
  <Direction:String> <StreamType:String>;

/*** Service proxies (Section 8.1) ************************/
ServiceProxy: Group ::= ;
abstract ControlItem ::= ;
Group: ControlItem ::= Info:GroupInfo ControlItem*;
Command: ControlItem ::= Info:CommandInfo Param*;
Param: ControlItem ::= Info:ParamInfo;
Stream: ControlItem ::= Info:StreamInfo;

/*** Wire Protocol addressing (Section 5.4) ***************/
DeviceID ::= <String>;
Selector ::= <String>;
GroupID ::= <String>;
Topic ::= <String>;

/*** Service addressing (Section 5.5.1) *******************/
abstract AddressPart ::= DeviceID <ServicePartID:String>;
LocalSID: AddressPart ::=;
VersionPart: AddressPart ::=;
abstract PRDVersion ::=
  CreatingVersionPart:VersionPart
  UpdatingVersionPart:VersionPart
  PreviousVersionPart:VersionPart
```

```
  MergedFromVersionPart:VersionPart;
ServiceID: PRDVersion ::= ;
AssemblyID: PRDVersion ::= <LogicalVersion:String>;
ServiceInstanceID ::=
  [DeviceID] ServiceID:ServiceID <InstanceNumber:String>;

/*** Discovery Protocol messages (Section 5.5.2) **********/
abstract Request ::= ;
abstract Reply ::= ;

PRDDeviceRequest: Request ::= ;
PRDDeviceReply: Reply ::=
  <ServiceCache:String> <ConnectionCache:String>
  <AssemblyCache:String> <SelectorCache:String>
  <ServiceStatusCache:String> PRDDevice;

PRDServiceListRequest: Request ::= ParentLocalSID:LocalSID;
PRDServiceListReply: Reply ::=
  <CacheNo:String> PRDServiceList;
PRDServiceRequest: Request ::= LocalSID;

ServiceInstanceIDRequest: Request ::= LocalSID;
ServiceInstanceIDReply: Reply ::=
  LocalSID ServiceInstanceID;

LocalServiceIDRequest: Request ::= ServiceInstanceID;
LocalServiceIDReply: Reply ::= ServiceInstanceID LocalSID;

SelectorRequest: Request ::= LocalSID;
SelectorReply: Reply ::= LocalSID Selector;

ServiceDescriptionRequest: Request ::= LocalSID;
ServiceDescriptionReply: Reply ::= ServiceDescription;

ServiceStatusRequest: Request ::= LocalSID;
ServiceStatusReply: Reply ::=
  LocalSID <ServiceStatus:byte> <StatusHelpText:String>;

PRDConnectionListRequest: Request ::=;
PRDConnectionListReply: Reply ::= PRDConnectionList;
```

B.2 Assemblies

The language of assembly descriptors was presented in Chapter 6. Its grammar builds on the grammar for device and service descriptors (Section B.1):

```
/*** Multiple versions of one assembly ********************/
PRDAssemblyD ::=
  <Format:String> <Name:String> BaseVersion:VersionPart
  PRDAssemblyVer*;

/*** One version of an assembly ***************************/
PRDAssemblyVer ::=
  <Format:String> <Name:String> Version:AssemblyID
  <Released:boolean>
  Devices:DeviceDeclList
  Services:ServiceDeclList
  Connections:ConnectionDeclList
  [EventHandlerScript]
  SynthesizedServices:SynthesizedServiceList;

/*** Device declarations **********************************/
DeviceDeclList ::= DeviceDecl*;
DeviceDecl ::= NameID:Identifier DeviceAddress;
UnboundDeviceAddress: DeviceAddress ::= ;

/*** Service declarations *********************************/
ServiceDeclList ::= ServiceDecl*;
ServiceDecl ::=
  LocalName:Identifier Decl:AbstractServiceDecl;
abstract AbstractServiceDecl ::= ;
SingleServiceDecl: AbstractServiceDecl ::=
  ServiceName:Identifier DeviceUse ServiceID
  <InstanceString:String>;
AltServiceDeclList: AbstractServiceDecl ::=
  ServiceDecl:AltServiceDecl*;
AltServiceDecl: SingleServiceDecl ::= <Prio:String>;

abstract ServiceExp;
ServiceUse: ServiceExp ::= Identifier;
ThisService: ServiceExp;
SynthesizedServiceUse: ServiceExp ::= Identifier;

/*** Connection declarations ******************************/
ConnectionDeclList ::= ConnectionDecl*;
ConnectionDecl ::= Provider:ServiceExp Customer:ServiceExp;

/*** Event handlers ***************************************/
EventHandlerScript ::=
  Variables:VariableList EventHandlers:EventHandlerList;
```

```
VariableList ::= VariableDecl*;
VariableDecl ::= VariableType Identifier;
abstract VariableType;
MimeType: VariableType ::= <TypeName:String>;

EventHandlerList ::= EventHandlerClause*;
EventHandlerClause ::=
  <CommandName:String> ServiceExp [CommandInfo] Action*;

abstract Action ::= ;
AssignAction: Action ::= VariableUse ParamUse;
abstract ActionWithParams: Action ::=
  <Command:String> ParamValue:Use*;
SendMessageAction: ActionWithParams ::= ServiceExp;
InvokeAction: ActionWithParams ::=
  SynthesizedServiceUse <AddressingType:String>
  [ServiceInstanceID] [AddressUse];
SelfTestAction: ActionWithParams ::= ;

/*** Declarations of synthesized services ****************/
SynthesizedServiceList ::= SynthesizedService*;
SynthesizedService ::=
  <Distribution:byte> [GroupID] [Topic] ServiceDescription
  <RemoteConnect:boolean>;

/*** Basic grammar elements ******************************/
Identifier ::= <ID:String>;
DeviceUse ::= Identifier;

abstract Use ::= ;
VariableUse: Use ::= <Name:String>;
ParamUse: Use ::= <Name:String>;
ConstantUse: Use ::= Identifier;
MissingUse: Use;
AddressUse: Use ::= <Name:String>;
```

Bibliography

[1] 43D ApS. Topos. `http://www.43d.dk/topos.php`.

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

[3] Gregory D. Abowd, Elizabeth D. Mynatt, and Tom Rodden. The Human Experience. *IEEE Pervasive Computing*, 1(1):48–57, 2002.

[4] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: an appliance-independent XML user interface language. *Comput. Netw.*, 31(11-16):1695–1708, 1999.

[5] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language Support for Connector Abstractions. In L. Cardelli, editor, *ECOOP 2003*, pages 74–102, 2003.

[6] OSGi Alliance. *OSGi Service Platform Core Specification*, 2005. Release 4, Version 4.1.

[7] Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. iStuff: A Physical User Interface Toolkit for Ubiquitous Computing Environments. In *Proceedings of the ACM CHI 2003 Conference on Human Factors in Computing Systems*, pages 537–544, Ft. Lauderdale, Florida, USA, April 2003.

[8] 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.

[9] Michael Beigl and Hans Gellersen. Smart-Its: An Embedded Platform for Smart Objects. In *Smart Objects Conference, SOC2003*, May 2003.

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

[11] Bluetooth.com. Specification Documents. `http://bluetooth.com/Bluetooth/Technology/Building/Specifications/` Accessed August 27, 2008.

[12] Bluetooth.com. The Official Bluetooth® Wireless Info Site. `http://www.bluetooth.com/` Accessed October 16, 2008.

[13] Jeppe Brønsted, Klaus Marius Hansen, and Mads Ingstrup. A Survey of Service Composition Mechanisms in Ubiquitous Computing. In *UbiComp 2007 Workshop Proceedings*, pages 87–92, 2007. Second Workshop on Requirements and Solutions for Pervasive Software Infrastructures (RSPSI).

[14] Jeppe Brønsted, Erik Grönvall, and David Fors. Palpability Support Demonstrated. In *Embedded and Ubiquitous Computing*, volume 4808/2007 of *Lecture Notes in Computer Science*, pages 294–308. Springer Berlin/Heidelberg, 2007.

[15] Jeppe Brønsted and Klaus Marius Hansen. Handling membership dynamicity in service composition for ubiquitous computing. In *International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, UBICOMM'08*, 2008. To appear.

[16] N. Carreiro and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4), 1989.

[17] Girish Chafle, Sunil Chandra, Vijay Mann, and Mangala Gowri Nanda. Decentralized orchestration of composite web services. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *WWW (Alternate Track Papers & Posters)*, pages 134–143. ACM, 2004.

[18] Dipanjan Chakraborty, Anupam Joshi, Tim Finin, and Yelena Yesha. GSD: A Novel Group-based Service Discovery Protocol for MANETs. In *4th IEEE Conference on Mobile and Wireless Communications Networks (MWCN)*, Stockholm. Sweden, September 2002. IEEE.

[19] Dipanjan Chakraborty, Filip Perich, Sasikanth Avancha, and Anupam Joshi. DReggie: Semantic Service Discovery for M-Commerce Applications. In *Workshop on Reliable and Secure Applications in Mobile Environment, In Conjunction with 20th Symposium on Reliable Distributed Systems (SRDS)*, 2001.

[20] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. Technical report, Apple Computer, Inc., August 2006. Internet-Draft (work in progress), `http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd-04.txt`.

[21] S. Cheshire and M. Krochmal. Multicast DNS. Technical report, Apple Computer, Inc., August 2006. Internet-Draft (work in progress), http://files.multicastdns.org/draft-cheshire-dnsext-multicastdns-06.txt.

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

[23] CORDIS. ISTweb. http://cordis.europa.eu/ist/ Accessed October 16, 2008.

[24] Aino Vonge Corry, Klaus Marius Hansen, and David Svensson. Traveling Architects – A New Way of Herding Cats. In *Quality of Software Architectures*, volume 4214/2006 of *Lecture Notes in Computer Science*, pages 111–126. Springer Berlin/Heidelberg, 2006.

[25] Nigel Davies and Hans-Werner Gellersen. Beyond Prototypes: Challenges in Deploying Ubiquitous Systems. *IEEE Pervasive Computing*, 1(1):26–35, 2002.

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

[27] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-Oriented Programming in AmbientTalk. In *20th European Conference on Object-Oriented Programming*, pages 230–254, 2006.

[28] Dns-sd.org. DNS SRV (RFC 2782) Service Types. http://www.dns-sd.org/ServiceTypes.html. Accessed August 27, 2008.

[29] Eclipse.org. Eclipse.org home. http://www.eclipse.org/.

[30] Eclipse.org. SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/ Accessed October 16, 2008.

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

[32] 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.

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

[34] W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy, and Trevor F. Smith. Supporting serendipitous integration in mobile computing environments. *Int. J. Hum.-Comput. Stud.*, 60(5–6):666–700, 2004.

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

[36] Torbjörn Ekman and Görel Hedin. The JastAdd system — modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[37] Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[38] Thomas Forsström and David Raimosson. External ad-hoc communication for PalCom. Master's thesis, Department of Computer Science, Lund University, 2007. LU-CS-EX: 2007-11.

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

[40] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

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

[42] Google. Google Earth. `http://earth.google.com/`.

[43] Robert Grimm. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[44] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[45] Erik Grönvall, Patrizia Marti, Alessandro Pollini, and Alessia Rullo. Active surfaces: a novel concept for end-user composition. In *NordiCHI '06: Proceedings of the 4th Nordic conference on Human-computer interaction*, pages 96–104. ACM Press, 2006.

[46] Erik Grönvall, Alessandro Pollini, Alessia Rullo, and David Svensson. Designing game logics for dynamic Active Surfaces. MUIA 2006: third international workshop on mobile and ubiquitous information access. Espoo, Finland, September 2006.

[47] Erik Grönvall, Luca Piccini, Alessandro Pollini, Alessia Rullo, and Giuseppe Andreoni. Assemblies of Heterogeneous Technologies at the Neonatal Intensive Care Unit. In *Ambient Intelligence*, volume 4794/2007 of *Lecture Notes in Computer Science*, pages 340–357. Springer Berlin/Heidelberg, 2007.

[48] Martin Gudgin et al. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*, April 2007. `http://www.w3.org/TR/2007/REC-soap12-part1-20070427/`.

[49] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programmming*, 8(3):231–274, 1987.

[50] Stephan Hartwig, Jan-Peter Strömann, and Peter Resch. Wireless Microservers. *IEEE Pervasive Computing*, 1(2):58–66, 2002.

[51] Todd D. Hodes and Randy H. Katz. A document-based framework for internet application control. In *USITS'99: Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems*, pages 6–6, Berkeley, CA, USA, 1999. USENIX Association.

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

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

[54] Valerie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Francoise Sailhan, Rafik Chibout, Nicole Levy, and Angel Talamona. Developing Ambient Intelligence Systems: A Solution based on Web Services. *Automated Software Engineering*, 12(1):101–137, January 2005.

[55] 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.

[56] Jini.org. Jini Specifications. `http://www.jini.org/w/index.php?title=Category:Jini_Specifications&oldid=2240`. Accessed August 27, 2008. Permanent link.

[57] Brad Johanson, Armando Fox, and Terry Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing*, 1(2):67–74, 2002.

[58] 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/`.

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

[60] 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.

[61] 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.

[62] Tim Kindberg and Armando Fox. System Software for Ubiquitous Computing. *IEEE Pervasive Computing*, 1(1):70–81, 2002.

[63] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[64] Johan Kristell. Interoperability between PalCom and external protocols. Master's thesis, Department of Computer Science, Lund University, 2008. LU-CS-EX: 2008-18.

[65] 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.

[66] Mohan Kumar, Behrooz A. Shirazi, Sajal K. Das, Byung Y. Sung, and David Levine. PICO: A Middleware Framework for Pervasive Computing. *IEEE Pervasive Computing*, 2(3):72–79, 2003.

[67] Jon Lathem, Karthik Gomadam, and Amit P. Sheth. SA-REST and (S)mashups : Adding Semantics to RESTful Services. *Semantic Computing, 2007. ICSC 2007. International Conference on*, pages 469–476, Sept. 2007.

[68] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[69] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[70] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall PTR, 2nd edition, April 1999.

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

[72] Boel Mattsson and Brice Jaglin. Implementing the PalCom protocol in an Axis network camera. Master's thesis, Department of Computer Science, Lund University, 2007. LU-CS-EX: 2007-02.

[73] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26:70–93, 2000.

[74] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 178–187, New York, NY, USA, 2000. ACM.

[75] 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.

[76] S. Motegi, K. Tasaka, A. Idoue, and H. Horiuchi. Proposal on Wide Area DLNA Communication System. *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 233–237, Jan. 2008.

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

[78] Emma Nilsson-Nyman. Software Integration in Health Care. Master's thesis, Department of Computer Science, Lund University, 2007.

[79] OASIS. *UDDI Version 3.0.2*. UDDI Spec Technical Committee Draft, Dated 20041019.

[80] OASIS. *Web Services Business Process Execution Language Version 2.0*, April 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

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

[82] PalCom Project. Cyclic development. `http://www.ist-palcom.org/approach/cyclic-development/`.

[83] PalCom Project. Tall Ships' Race Aarhus. `http://www.ist-palcom.org/activities/tall-ships-race-aarhus/`.

[84] PalCom Project. Try It! `http://www.ist-palcom.org/try-it/`.

[85] PalCom Project. PalCom External Report 53: Deliverable 38 (2.14.2): Dissemination. Technical report, PalCom Project IST-002057, November 2006. http://www.ist-palcom.org/publications/deliverables/Deliverable-38-[2.14.2]-dissemination.pdf.

[86] PalCom Project. PalCom External Report 69: Deliverable 54 (2.2.3): Open Architecture. Technical report, PalCom Project IST-002057, December 2007. http://www.ist-palcom.org/publications/deliverables/Deliverable-54-[2.2.3]-open-architecture.pdf.

[87] PalCom Project. PalCom External Report 70: Developer's Companion. Technical report, PalCom Project IST-002057, April 2008. http://svn.ist-palcom.org/svn/palcom/trunk/doc/tex/companion.pdf Accessed October 16, 2008.

[88] PalCom web site. Palpable Computing—a new perspective on Ambient Computing. http://www.ist-palcom.org/.

[89] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct. 2003.

[90] Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing*, pages 56–75, London, UK, 2001. Springer-Verlag.

[91] 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.

[92] 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.

[93] Tom Rodden, Andy Crabtree, Terry Hemmings, Boriana Koleva, Jan Humble, Karl-Petter Åkesson, and Pär Hansson. Between the dazzle of a new building and its eventual corpse: assembling the ubiquitous home. In *DIS '04: Proceedings of the 5th conference on Designing interactive systems*, pages 71–80, New York, NY, USA, 2004. ACM.

[94] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A Middleware

Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[95] Marwan Sabbouh, Jeff Higginson, Salim Semy, and Danny Gagne. Web mashup scripting language. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 1305–1306, New York, NY, USA, 2007. ACM.

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

[97] A. Shenoi, Y. Yesha, Y. Yesha, and A Joshi. A framework for specification and performance evaluation of service discovery protocols in mobile ad-hoc networks. *Ad-hoc Networks*, 4(1):1–23, 2006.

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

[99] Daniel Steinberg and Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[100] Jing Su, James Scott, Pan Hui, Jon Crowcroft, Eyal de Lara, Christophe Diot, Ashvin Goel, Meng How Lim, and Eben Upton. Haggle: Seamless Networking for Mobile Applications. In *UbiComp 2007: Ubiquitous Computing*, volume 4717/2007 of *Lecture Notes in Computer Science*, pages 391–408. Springer Berlin/Heidelberg, 2007.

[101] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Trans. Softw. Eng. Methodol.*, 1(3):229–268, 1992.

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

[103] Sun. Mobile Information Device Profile. `http://java.sun.com/products/midp/` Accessed October 16, 2008.

[104] Sun. SunSpotWorld - Home of Project Sun SPOT. `http://www.sunspotworld.com/` Accessed October 16, 2008.

[105] Sun. The Java ME Platform. `http://java.sun.com/javame/` Accessed October 16, 2008.

[106] Sun. The Swing Tutorial. `http://java.sun.com/docs/books/tutorial/uiswing/index.html` Accessed October 16, 2008.

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

[108] David Svensson, Görel Hedin, and Boris Magnusson. Pervasive applications through scripted assemblies of services. *Pervasive Services, IEEE International Conference on*, pages 301–307, July 2007.

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

[110] 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.

[111] uClinux. uClinux – Embedded Linux Microcontroller Project. `http://www.uclinux.org/` Accessed October 16, 2008.

[112] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, 1987.

[113] UPnP™ Forum. UPnP™ Standards. `http://www.upnp.org/standardizeddcps/` Accessed August 27, 2008.

[114] UPnP™ Forum. UPnP™ Device Architecture 1.0. Technical report, `http://www.upnp.org/`, December 2003. Version 1.0.1.

[115] Mathieu Vallée, Fano Ramparany, and Laurent Vercouter. Flexible Composition of Smart Device Services. In *Advances in Pervasive Computing, Adjunct Proceedings of Pervasive 2006*, May 2006.

[116] N. Venkitaraman. Wide-Area Media Sharing with UPnP/DLNA. *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 294–298, Jan. 2008.

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

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

[119] 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.

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

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

[122] M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research at PARC in the late 1980s. *IBM Syst. J.*, 38(4):693–696, 1999.

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

[124] Michel Wermelinger, José Luiz Fiadeiro, Luís Andrade, Georgios Koutsoukos, and João Gouveia. Separation of Core Concerns: Computation, Coordination, and Configuration. In *Workshop on Advanced Separation of Concerns, OOPSLA'01*, 2001.

[125] Feng Zhu and Matt W. Mutka. Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, 4(4):81–90, October–December 2005.