IST-002057 **PalCom**

Palpable Computing:

A new perspective on

Ambient Computing

Deliverable 54 (2.2.3)

Open Architecture

Due date of deliverable: m 48
Actual submission date: m 46

Start date of project: 01.01.04
Duration: 4 years

Ecole Polytechnique Fédérale de Lausanne (EPFL)
Whitestein Technologies
University of Aarhus
Lund University
Siemens AG

Revision: 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	**PU**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Integrated Project

Information Society Technologies

Contents

List of Figures

List of Tables

Executive Summary

This document represents the PalCom External Report #69, Deliverable 54 (2.2.3), Month 48 for Work Package 2 – the PalCom Open Architecture.

Being the fourth edition of the PalCom Open Architecture specification, this document represents the final report with respect to the architectural aspects of the PalCom project.

This document provides an overview of the proposed PalCom Open Architecture for enabling the development of systems supporting palpable computing. The document is positioned as a main entry point for those requiring an understanding of the concepts and principles employed by the architecture. The document also provides definitions of the core architectural models and their interrelationships according to the overall system design.

This document describes the final stage of the architecture at two levels:

Firstly a general architectural description of architecture(s) that meet the specific architectural challenges identified for palpable systems.

Secondly a concrete proposal – The PalCom Open Architecture – of one such architecture, including further specifications, an overview of the prototype implementation of, and a definition of compliance levels with the PalCom Open Architecture provided by the PalCom project.

Preface

This document represents the forth and final version of the PalCom Open Architecture specification, and thus replaces the previous drafts [48, 57, 61].

The objectives of this edition of the architecture has been consolidation of the various design models and APIs previously developed in the project, as well as an alignment between the specifications and the reference implementation provided as part of the PalCom Open Source dissemination toolkit [41, 67].

This document serves as both a first contact point for those requiring an introduction to the architecture and as a specification of the PalCom Open Architecture itself. In the latter respect definitions are made of the core architectural concepts and their interrelationships according to the overall system design.

Being the "umbrella" for the architectural workpackages WP2-5, and to some degree WP6, this deliverable represents the overall result of all these workpackages, and attempts to present the consolidated efforts in a comprehensible way. This document mainly presents the final outcome, and does not discuss the many alternatives considered during the project time. For information about these, the reader is referred to the previous deliverables of WP2-6 [48, 45, 50, 54, 55, 56, 57, 58, 59, 60, 61, 63, 64, 65, 66], as well as numerous Working Notes from the project (not publically available).

Some of the material presented in this deliverable is replicated from these previous contributing writings - but updated according to the current state of the design and implementation - whereas some of the material will be new, also for readers of the previous writings.

This deliverable constitutes the result of one of the two main objectives of PalCom – An Open Architecture for Palpable Computing. The other main objective – A Conceptual Framework for Palpable Computing – is described in Deliverable 53 [70]. As such, this deliverable presents the more technical aspects of the project, whereas Deliverable 53 describes the motivation for as well as our current understanding of *palpability*. Naturally a strong dependency exists between these two deliverables.

The objectives and tasks of WP2 for the final reporting period is detailed in the Description of Work, version 6.5 update [42], summarized in the following Section .

Workpackage 2 Objectives

Objectives

To continue the development of a specification and a reference prototype of an open architecture for palpable computing with a focus on supporting the identified palpable properties. The PalCom Open Architecture specification and prototype will be used as the basis for specific application design and implementation.
The architecture will be open in the sense that it supports open-ended composition and development of palpable computing components and integration with non-palpable systems and components.
The architecture will support the development of ambient and pervasive computing systems by balancing quality requirements and challenges of ambient computing with the specific requirements and challenges of palpable computing. Overall, the objectives of this period will be to continue the process undertaken in the previous period covering the development of the aspects of the open architecture through:

- Technical considerations exemplified and verified through architectural prototypes.
- Adapting the architecture to the needs established through the application prototypes.
- Adapting the architecture to the concepts provided by the conceptual framework.
- Positioning the open architecture against related work.

These inputs are expected to provide a solid foundation for a mature version of the architecture, which will be reflected in its specification.

Deliverables

Month 48: Final Open Architecture Specification (2.2.3).

Tasks

Task 4. Input to Demo- & Dissemination Kits Maturing the architectural model and prototype of the Open Architecture Reference Implementation enough to provide a foundation for the Demo- & Dissemination Kits.

Task 5. Integration of design models Based on the provided models and APIs from the required Toolbox needs, these will be reviewed and integrated to make sure that the elements of the Open Architecture are coherent and compatible. The result of this work will be (1) the feedback to WP3-6 with possible Request for Changes, and (2) a subsequent model integration.
Task 6. Final Open Architecture Specification This deliverable follows the work from the reporting period, this effort will at the end result in the final Open Architecture Specification.

Contributors

The following people have contributed to this revision of the deliverable:

- Peter Andersen, University of Aarhus, *datpete* **at** *daimi.au.dk*
- David Svensson Fors, Lund University, *david* **at** *cs.lth.se*
- Thomas Forsström, Lund University, *thomasf* **at** *cs.lth.se*
- Tony Gjerlufsen, University of Aarhus, *tonz* **at** *daimi.au.dk*
- Roberto Ghizzioli, Whitestein Technologies, *rgh* **at** *whitestein.com*
- Dominic Greenwood, Whitestein Technologies, *dgr* **at** *whitestein.com*
- Görel Hedin, Lund University, *gorel* **at** *cs.lth.se*
- Mads Ingstrup, University of Aarhus, *ingstrup* **at** *daimi.au.dk*
- Boris Magnusson, Lund University, *boris* **at** *cs.lth.se*
- Emma Nyman, Lund University, *d02en* **at** *student.lth.se*
- Jesper Wolff Olsen, University of Aarhus, *jexper* **at** *daimi.au.dk*
- Sven Robertz, Lund University, *sven* **at** *cs.lth.se*
- Reiner Schmid, Siemens AG, *reiner.schmid* **at** *siemens.com*
- Jesper Honig Spring, Ecole Polytechnique Fédérale de Lausanne (EPFL), *jesper.spring* **at** *epfl.ch*
- Konrad Tollmar, Lund University, *konrad.tollmar* **at** *ics.lu.se*

Past contributors to documents this deliverable is based upon include:

- Aino Vonge Corry, University of Aarhus, *apaipi* **at** *daimi.au.dk*
- Erik Corry, University of Aarhus, *corry* **at** *daimi.au.dk*
- Frank Buschmann, Siemens AG, *frank.buschmann* **at** *siemens.com*
- Henrik Bærbak Christensen, University of Aarhus, *hbc* **at** *daimi.au.dk*
- Torbjörn Ekman, Lund University, *torbjorn* **at** *cs.lth.se*
- Erik Ernst, University of Aarhus, *eernst* **at** *daimi.au.dk*
- Jacob Frølund, University of Aarhus, *frolund* **at** *daimi.au.dk*
- Klaus Marius Hansen, University of Aarhus, *klaus.m.hansen* **at** *daimi.au.dk*
- Michael Lassen, University of Aarhus, *henryml* **at** *daimi.au.dk*
- Ole Lehrmann Madsen, University of Aarhus, *olm* **at** *daimi.au.dk*
- Nikolay Mihaylov, Ecole Polytechnique Fédérale de Lausanne (EPFL), *nikolay.mihaylov* **at** *epfl.ch*
- Ulrik Pagh Schultz, University of Aarhus, *ups* **at** *daimi.au.dk*
- Peter Ørbæk, University of Aarhus, *poe* **at** *daimi.au.dk*

Chapter 1

Introduction

The PalCom Open Architecture is a Service Oriented Architecture (SOA) where loosely coupled, independently communicating services on network-enabled devices can be assembled using a novel mechanism for composites, called PalCom Assemblies. Assemblies are dynamically (re-)bound at the application level from high-level resources (services/devices/actors/communication channels). Assemblies are specified with human-readable assembly descriptors (XML), and can be (de-/re-)constructed using visual browsers or purely programmatically. Like the aforementioned high-level resources, the assembly has an explicit runtime manifestation. The description of the bindings and dynamic behavior of an assembly is described using a simple scripting language.

The architecture is open in the sense that it has a public specification including formats and protocols and an open-source reference implementation. Furthermore the assembly mechanism allows for open-ended composition in terms of interoperation with non-PalCom devices and services, and for open-ended development strategies in the sense of supporting development for a dynamically changing application environment.

To support sense-making by the user, the architecture provides generic support for making services and assemblies inspectable. This is done using the Hierarchical Graph (H-Graph) datastructure to organize access to manifestations of resources. H-Graphs provide uniform access to all data, and thus challenge the traditional notion of encapsulation. The H-Graph is made remotely accessible through a service on each node which thus provides an open data representation across network nodes.

PalCom Services are instantiated from components into discoverable entities executing on a platform independent runtime environment providing basic infrastructure functionality executing on a runtimeengine. The runtime engine is realized as a virtual machine: either the standard Java VM or the novel PalCom Virtual Machine (Pal-VM) which supports multi-lingual development and has mechanisms for monitoring low-level resources.

The PalCom Services communicate via open communication protocols using a human-readable, yet performant representation. Pluggable media managers can adapt communication to any low-level protocol to support transparency across multiple bearers. Adaptive selection of communication channels according to situation and user needs allows communication via publish/subscribe (multicast), point-to-point messaging (one-shot), or point-to-point connections (e.g. streaming).

The architecture provides three middleware managers. The Resource Manager provides discovery mechanisms for low- and high-level resources used to construct assemblies. The Assembly Manager establishes and maintains dynamic behavior of bindings in assemblies. In cooperation with these two managers the Contingency Manager supports resilience through automatic reactive and proactive, as well as manual problem compensation.

A slightly more detailed, itemized, version of this short introduction of the PalCom Open Architecture can be found in Appendix A, “PalCom Open Architecture Highlights”.

1.1 Organization

The rest of this document is structured as follows:

First in Chapter 2 we discuss the original PalCom Challenges (from [40]), and how these can be addressed by a number of architectural qualities.

With this as background follows the first of the two major parts of this deliverable: discuss the concepts of a *general palpable architecture* from a purely ontological viewpoint. Part II details of the *specification and reference implementation of the PalCom Open Architecture*, being one proposal for reifying a palpable architecture as defined in Part I.

When reading this report, it is important to remember this distinction between the two parts: Part I describes palpable architectures in general, relating to, e.g., the architectural qualities argued for in Chapter 2, whereas Part II details our current proposal for a concrete architecture and implementation thereof concretizing a palpable architecture. Part II, Chapter 10, is concluded with a chapter defining various *compliance levels* with *The PalCom Open Architecture*. Again it is important to stress that these compliance levels relate to our concrete proposal for a palpable architecture, and that other concrete architctures and implementations may fully qualify as palpable architectures even with little or no compliance with the PalCom architecture.

In this report we reserve the term “PalCom (Open) Architecture” for our concrete architecture and implementation thereof. When talking about architecture supporting palpability in general, we use the term “palpable architecture”.

Below is a more detailed survey of the structure of the rest of this document:

Part I is structured as one in Section 3 by giving a tabular overview of the concepts of a palpable architecture. Next follows 16 sections in Section 3.1 each detailing one of the normative ontology concepts for palpable architectures. Each concept is explained in detail, including it’s relations to the other concepts. Naturally Part I has a very strong relationship with the PalCom Conceptual Framework [70].

Part II details the specification of and parts of the reference implementation of the PalCom Open Architecture, which constitutes our proposal for a specific architecture having the qualities described in Part I.

Part II has many chapters: First in Chapter 4 we give a schematic overview of the PalCom architecture and the reference implementation, as well as explanation as to why this constitutes a reification of the ontological concepts of Part I. Based on this schematic overview, the rest of Part II details each of the main elements of the reified architecture: Resource frameworks, including Service, Assembly and Device, are presented in Chapter 5. Next our different Middleware Managers, i.e. Resource-, Assembly-, and Contingency-Manager, are presented in Chapter 6. Then follows a long chapter, Chapter 7, which presents a number of aspects of the PalCom Runtime Environment. These include the Communication Model in Section 7.1, Resource Descriptors in Section 7.2, Inspection through the use of HGraphs in Section 7.3, processes and threads in Section 7.4, and finally the two different Runtime Engines used by the reference implementation – JVM and Pal-VM – are discussed in Section 7.5 For the Pal-VM, the main characteristics of the design of this novel virtual machine is also presented. Next in Part II follows two smaller chapters: Chapter 8 discuss which hardware and operating system requirements the PalCom reference implementation has, and Chapter 9 discuss some additional functional elements of the PalCom reference implementation providing Display and Storage capabilities.

Part II concludes by specifying four compliance levels with the PalCom architecture and implementation, through which alternative implementations can measure the interoperability with PalCom.

The last part of this deliverable – Part III – consists of a number of appendices relating to Part I and Part II:

First in Appendix A highlights of the of the PalCom Open Architecture has been distilled. Second in Appendix B a survey is given of service composition mechanisms – corresponding to our Assembly mechanism – in other architectures[1]. Next follows three appendices relating to communication and discovery: Appendix C details the PalCom communication wire protocols, and Appendix D gives a number of example messages. This is concluded by Appendix E explaining discovery messages.

[1]For other related work we refer to the survey included in the Conceptual Framework deliverable [70] and previous PalCom deliverables.

1.2 Dependencies

Although it has been a goal to keep this report as self-contained as possible, a some dependencies on other documents exist.

First, as mentioned in the introduction above, valuable background information on the process over the past four years, that has resulted in the current report, can be found in the previous deliverables of WP2-6 [48, 45, 50, 54, 55, 56, 57, 58, 59, 60, 61, 63, 64, 65, 66].

As also mentioned above, a strong relationship exists between the Conceptual Framework for Palpable Computing and the Open Architecture for Palpable Computing. It is therefore recommended that the reader study the PalCom Conceptual Framework [70] to establish an understanding of the concept of palpability.

Furthermore, for practical use of the PalCom Open Source dissemination toolkit, which provides a reference implementation of the architecture specified in this deliverable, we refer to the materials available on the website [41], and the PalCom Developer's Companion [71].

A number of architectural issues relating to assemblies are detailed in the WP6 End User Composition deliverable [68], available simultaneously with this report.

Finally, to keep the size of this report down, large parts of the specification of the `pal-vm` have been left out of section 7.5. As explained in that section, the reader is referred to the previous publically available WP3 Runtime Environment deliverable [63], which details the `pal-vm` specification.

Chapter 2

Qualities of Palpable Computing

Every properly constructed software architecture can be described by a set of meaningful, consistent and verifiable quality attributes, simply referred to as 'qualities' for the purposes of this document. These qualities predominantly describe the fundamental non-functional attributes of an architecture, although as is the case with the architecture defined in this document, they can also include aspects of functional behaviour. Regardless, they are the means by which an architecture can be generally characterized and, where relevant, differentiated from other architectures.

The derivation of the qualities described in this chapter was subject to several influences:

1. The set of challenges laid down at the commencement of the PalCom Project, within which the field of Palpable Computing has been defined. These challenges capture the limitations intrinsic to the core principles of pervasive computing when considering palpable, or humanistic social aspects, and form the grounding insight of what palpable computing should address. The challenges are outlined in Section 2.1.
2. The continuous iterative development of the general architecture for palpable computing (see Part I) and the specific reification of that, called the PalCom Architecture (see Part II). At all times during development, the various qualities were accounted for both in terms of their implications on design and potential refinement as a result of research activities.
3. Internationally standardized definitions of architectural quality attributes, such as those identified in the ISO/IEC standard 9126 dealing with product quality in software engineering. While guidance was taken from standards such as this, the unique nature of palpabe computing implied the necessity of introducing selected non-standard arechitectural qualities as described in section Section 2.2.

This Chapter therefore first presents the fundamental *challenges* that provide context to palpable computing, followed by the specific *qualities* that define the field itself, and thus its architectural realization as described in body of this document.

2.1 Challenges

The original formulation of these challenges can be found in *Palpable Computing: A new perspective on Ambient Computing* [40], the description of work outlining the PalCom project at its outset. The following descriptions are a refinement of these original formulations in accordance with knowledge gained during the course of defining the architecture for palpable computing and PalCom Architecture.

2.1.1 Challenge: Construction/Deconstruction

Actors must be able to construct and deconstruct collections of resources in accordance with their personal or business requirements. This incorporates both the aggregation of functional blocks to create a palpable system and breaking the same functional blocks apart to reconstitute them in different forms to, for example, resolve an operational problem. Moreover, it should be possible for such a constructed collection to be adaptively altered in terms of its membership and configuration according to events or other changes in the environment. All of these actions should be possible through both manual control by a human actor, and autonomous control by some computational element.

2.1.2 Challenge: Visibility/Invisibility

Actors interacting with aspects of a palpable system must be able to understand and control system configuration and behaviour with minimum effort, regardless of the system's size and complexity. An aspect of the approach to achieve this must be tunable levels of visibility over the internal structure and state of the constituent elements of a palpable system. For example, low visibility or opaqueness may be normal when an actor simply wishes a system to perform the operations it is expected to do regardless of what devices and services in the system are involved in the computation. Conversely, higher levels of visibility may be desirable in the case of erroneous behavior when actors may become personally interested in precisely what a system is doing and potentially may do, why it is behaving the way it does, and which specific system resources are involved in an operation. This notion of visibility and invisibility must apply for all resources which may participate in a palpable system.

2.1.3 Challenge: Change/Stability

Actors must be able to bring together collections of computational resources, such as devices and services, to form stable palpable systems that are nonetheless capable of change under prescribed circumstances. A primary factor in establishing and maintaining stability must be the structure and use of infrastructural and ad-hoc networking facilities to connect resources. Stability implies that an active palpable system should attempt to continuously provide expected behaviour by remaining as impervious as possible to disruptive conditions including changes to the resources constituting the system; a network connection can be considered to be a palpable resource.

2.1.4 Challenge: Understandability/Scalability

To effectively interact with and manipulate resources an actor must be able to understand how to manipulate and exploit them in meaningful ways both as individual entities and as participants in one or more palpable systems. In the latter case, as the number of resources and/or the complexity of their behavioural interactions scales up or down care must be taken that understandability in terms of the actor's experience remains consistent.

2.1.5 Challenge: Heterogeneity/Coherence

Typically a palpable system will be at least partially constituted from several physical and/or virtual devices. While nothing precludes these devices from being entirely homogeneous in form and/or behaviour, it is a realistic assumption that in the majority of cases devices will be heterogeneous. As a result sufficient interfaces, protocols and runtime support must be available when necessary to allow these devices to collaborate with one another as coherent groups.

2.1.6 Challenge: Autonomy/User-Control

A common requirement in most, if not all palpable systems is support for both human and machine decision-making concerning aspects including configuration and behaviour. Autonomous decisions are made by system resources themselves according to their comprehension of current state within the context of predefined rules and policies. User-controlled decisions are of course made by human actors. A balance and interrelationship must be effected between these two forms of decision-making when necessary, such as provisioning for oversight.

2.2 Qualities

The purpose of these qualities is to capture the essence of what defines the nature of palpability in simple, yet meaningful terms. Collectively they define both what palpablity is in terms of the discipline known as palpable computing, and also what features an architecture should exhibit in order to be considered compliant with that definition. This link between the abstract conceptual aspect of palpable computing and its architectural realization is of fundamental importance.

Significantly, these qualities should not be considered in isolation, but rather as interwoven contributory factors that exhibit dependencies and influences on one another. Moreover, for a system to be considered palpable, as defined by this document, it must exhibit all of these qualities to a degree appropriate to the application context. The manner in which these qualities are implemented is the choice of the designer.

Each quality is described in terms of its relationship to a resource, or set of resources, which are the fundamental elements of all palpable computing systems. A resource is typically some element of computation, but can essentially be any entity that can plays some role in a palpable system, including human actors.

2.2.1 Quality: Resource Awareness

Resource awareness describes the fact that resources within the scope of a palpable system can be, and often are, aware of one another's presence, availability and behaviour. Resources in this respect are any computational, physical or human elements that can be used or manipulated in some manner.

- Being aware of the *presence* of a resource implies the ability of one resource to discover other resources. In turn, this implies that discoverable resources are able to announce their presence, or to have their presence announced by others.
- Being aware of the *availability* of a resource implies the ability of one resource to know whether other resources can be used or manipulated in some manner.
- Being aware of the *behaviour* of a resource implies the ability of one resource to determine the operational capabilities of itself and other resources.

Note that awareness in the respect of this quality in no way relates to cogniscant awareness.

2.2.2 Quality: Assemblability

Assemblability describes the fact that resources can be assembled into multiple composite constructs that exhibit the collective sum behaviour of the constituent elements, or, in some cases, emergent behaviour that is only brought about through the collaboration of certain resources.

Assemblability is strongly related to adaptability which describes the fact that any assembled construct can be disassembled or reassembled into alternative formations in real-time, either manually or automatically.

Additionally, this quality is influenced by the quality of experimentability in that an experimental mode of interaction must be supported for the manual handling of multiple composite constructs.

2.2.3 Quality: Inspectability

Inspectability describes the fact that the structure, state and behaviour of resources may be inspected by users at an appropriate level of detail for a particular context of use. This includes the provision of purely programmatic means by which resources may access information about each other's structure and behaviour. Inspection is enabled through resource-specific monitoring and interrogation techniques. Monitoring passively observes behaviour, whereas interrogation actively probes structure, state and behaviour through available inspection interfaces, constructs and tools.

Inspection allows the attributes and behavioural capabilities of a resource to be exposed at various degrees of visibility, typically dependent on the particular resource type and function it is performing.

Inspection is instrumental for assemblability as it offers a means to navigate across the various levels of abstraction for a given composite construct to present an understanding of what can be assembled, when and under what constraints.

The combination of inspectability and experimentability is crucial as actor perception is typically a continuous participatory explorative actvity, rather than akin to taking of a snapshot followed by static interpretation. Similarly, experimentability is most valuable when supported with inspection to assess the result of an experiment.

2.2.4 Quality: Adaptability

Adaptability describes the fact that certain resources can dynamically change their behaviour in response to detected events or environmental conditions.

Dynamic resource reconfiguration and behaviour modification can be effected by either programmatic means or through human interaction.

Resource awareness and inspectability provide the necessary programmatic abstractions to enable human or algorithmic reasoning about adaptations, while assemblability provides the means to effect that reasoning.

2.2.5 Quality: Resilience

Resilience describes the fact that, when required, systems conforming to palpable computing principles should exhibit self-initiated behaviour that ensures a defined degree of reliability and survivability. Reliability implies that a system will attempt to do what is expected at all times, adapting its behaviour as necessary to ensure such. Survivability builds on reliability to ensure that a system can tolerate problem conditions and continue to perform as expected even in the presence of disruptive events.

Many resource types may not be intrinsically capable of ensuring their own resilience beyond behavioural adaptation. Typically, additional, specialized entities are required to monitor and perform actions leading to assured resilience. Regardless, one or more elements of a system must be responsible for automatic, self-initiated behaviour to bring about specific or general system resilience. However, such autonomic behaviour should be subject to inspection, that is, make the survival strategies navigable to the user.

Resilience is based on inspectability and resource awareness in the same manner as adaptability. It thus goes beyond standard models of reliability by employing a broader resilience-in-use perspective that includes empowering the user to help the system survive a problem condition.

2.2.6 Quality: Experimentability

Experimentability describes the capability of facilitating and encouraging exploratory experimentation by users participating in systems that conform to palpable computing principles. This is because palpability intrinsically implies the ability of a coherent collection of resources to be used, customized and altered within established degrees of freedom and constraints, such as performance and security.

Experimentability relies on resource awareness and inspection to provide an overview of what can be experimented with, and on assemblability and adaptability to enable an intervention to be made when required. Experimentability should also ensure that users should be able to trust that exploratory experimentation will not yield destructive results.

2.2.7 Quality: Multiplicity

Multiplicity describes the fact that palpability must be considered as systemic within and throughout the resources constituting multi-partite palpable systems. While an individual resource may exhibit behaviours necessary to palpability, it is only when that resource interacts with others, whether cooperatively or competitively, that a palpable system of computation is formed. Moreover any given resource may be participating in multiple simultaneous dependent or independent interactive relationships.

Another aspect of multiplicity concerns the requirement that computational entities cannot assume a well defined individual environment, traditionally denoted a system, in relation to which their role and scope is defined. Thus the open-endedness characterising ubiquitous computing poses a requirement on the concepts in terms of which the software is individualized.

Multiplicity also closely relates to a number of the other qualities as the lack of a ’system’ which can be stopped, recompiled and started implies a parallism amongst the individual computational entities. This in turn requires an increased dynamicity of those entities and the ways in which interactions among them are enabled, maintained and and carried out due to the decoupling of what determines their individual availability and context of use over time.

Part I

An Architecture for Palpable Computing

Chapter 3

Palpable Computing Architectural Ontology

The Palpable Computing architectural ontology is a correlated collection of the principle normative concepts identified as mandatory elements of a specific architecture, such as the PalCom Architecture (see Part II), which reifies this architectural ontology. In all cases these concepts map onto either a physical artifact or computational component deployed within a compliant implementation. In this respect, the term *ontology* is used to capture the notion of a set of fundamental concepts and, importantly, the relationships between them.

The concepts defined in this section have been derived through an interative process of architectural design and refinement informed by development work and experimental evaluation. they represent the minimum set of physical artifacts and computational components required to construct and deploy what is considered to be a palpable system or application. The derivation of these concepts is discussed in earlier versions of this document, e.g., *Deliverable 39* [61].

The level of abstraction selected for this architectural ontology is deliberately high in order to cleanly delineate between the required features of the architecture and the particular methods and mechanisms used to architecturally reify and implement those features. Part II of this document reifies these abstract concepts into concrete architectural and implementation-specific concepts within the context of a PalCom Architecture that may be considered as the reference architecture produced by the PalCom Project consortium.

Certain concepts in the architectural ontology are classified as mandatory in that they are *required* for a reification to be considered architecturally compliant. For example, a Device (see Section 3.2.9) is classified as mandatory in the sense that a palpable assembly can only exist if populated by at least one Device. Equally, certain other concepts are classified as optional in that they are *not required* for a reification to be considered architecturally compliant, but are nevertheless important contributory elements.

3.1 Ontology Overview

Table 3.1 provides a tabular overview of the Palpable Architectural Ontology in terms of its constituent concepts, their basic description, and mandatory or optional presence status in architectural reifications of the ontology.

Table 3.2 provides a tabular overview of the Palpable Architectural Ontology concepts in terms of the architectural qualities (see Section 2) relevant to that concept. The strength of relevance is informally classified as either strong or weak.

Table 3.1: Palpable architecture ontology concepts.

Concept	Description	Presence
1st Order Resource	Resources associated with a physical device	Mandatory
2nd Order Resource	Resources that either contain or consume 1st Order Resources	Mandatory
Actor	Human or some other system that uses a palpable system	Mandatory
Assembly	Organized collection of 2nd Order Resources	Mandatory
Assembly Descriptor	Description of the organization and purpose of an assembly	Mandatory
Assembly Manager	Constructor, deployer and lifecycle manager of assemblies	Mandatory
Communication Channel	Means of communication between services	Mandatory
Contingency Manager	Resolver of assembly fault and problem conditions	Optional
Device	Computational hardware that hosts palpable software	Mandatory
Resource	Logical abstraction of 1st and 2nd Order Resources	Mandatory
Resource Descriptor	Description of a 2nd Order Resource	Mandatory
Resource Manager	Discoverer and maintainer of 2nd Order Resource directory	Mandatory
Runtime Component	Executable instantiation of a software component	Mandatory
Runtime Environment	Host environment for assemblies and 2nd Order Resources	Mandatory
Service	Remotely communicative Runtime Component	Mandatory
Synthesized Service	Service interface defining an assembly service composition	Mandatory

Table 3.2: Architectural qualities attributable to each ontological concept (X=strong, x=weak, representation).

Concept	ResAwrn	Assmblty	Inspect	Adaptblty	Reslnce	Mltplcty	Exprmtablty
1st Order Resource	X	-	x	-	-	-	-
2nd Order Resource	X	X	X	X	X	X	X
Actor	X	-	x	x	x	X	X
Assembly	X	X	X	X	X	X	X
Assembly Descriptor	-	X	X	-	-	-	-
Assembly Manager	X	X	X	X	X	X	X
Communication Channel	X	X	X	X	X	X	X
Contingency Manager	X	x	X	X	X	x	X
Device	X	X	X	X	X	X	X
Resource	X	X	X	x	x	x	X
Resource Descriptor	-	X	X	-	-	-	-
Resource Manager	X	x	X	X	X	x	X
Runtime Component	X	X	X	x	X	X	X
Runtime Environment	X	-	X	-	X	x	X
Service	X	X	X	X	X	X	X
Synthesized Service	X	X	X	X	X	X	X

3.2 Normative Ontology Concepts

This section defines the **normative** concepts of the Palpable Architectural ontology. Each concept is described by a short-form definition, its specific relationships to other concepts, and a detailed explanation of its role.

3.2.1 1st Order Resource

Definition

A low-level resource associated with a physical device (i.e., Device), or human user (i.e., Actor). The former case includes resources such as processor load, memory, bandwidth and power. The latter case includes resources such as role and availability.

Relationship to other concepts

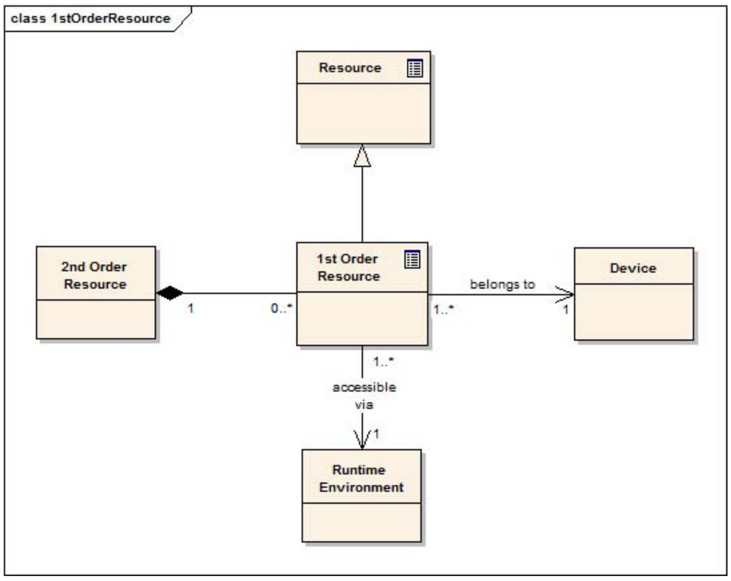


Figure 3.1: 1st Order Resource ontological relationships

- A **1st Order Resource** *specializes* **Resource**
- A **1st Order Resource** *is a constituent of a* **2nd Order Resource**
- A **1st Order Resource** *belongs to a* **Device**
- A **1st Order Resource** *is accessible via a* **Runtime Environment**

Explanation

All computational systems require basic resources with which to operate, including memory, processor cycles, power and bandwidth. As Palpable Computing considers applications as consisting of distributed populations of services running on various device types, it is reasonable and necessary to consider the consumable resources on those devices as raw materials that can be reasoned about and manipulated. For example, the deployment of a service that has pre-stated memory and bandwidth requirements in its service description should only be considered if those resources are available. In a further step, a resource manager service (see Section 6) may attempt to free up resources through priority overrides, scheduling, etc, or may even consider deploying the service on another networked device if feasible within the operational constraints of the service, devices and usage expectations.

First-order resources may be represented by functional abstractions to ease manipulation. For example, available processor load may be abstracted as the availability of threads in a thread pool.

3.2.2 2nd Order Resource

Definition

An abstraction used to describe those resources that either contain or consume 1st Order Resources. Examples include Assemblies, Services, Devices, Components and Actors.

Relationship to other concepts

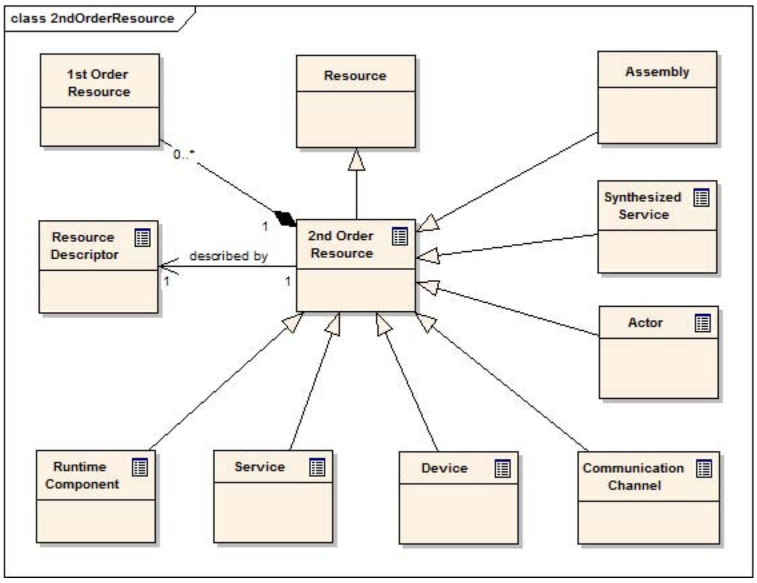


Figure 3.2: 2nd Order Resource ontological relationships

- A **2nd Order Resource** *specializes* **Resource**
- A **2nd Order Resource** *is constituted by* **1st Order Resources**
- A **2nd Order Resource** *is described by a* **Resource Descriptor**

- A **2nd Order Resource** *generalizes* **Runtime Component**
- A **2nd Order Resource** *generalizes* **Service**
- A **2nd Order Resource** *generalizes* **Device**
- A **2nd Order Resource** *generalizes* **Communication Channel**
- A **2nd Order Resource** *generalizes* **Actor**
- A **2nd Order Resource** *generalizes* **Service**
- A **2nd Order Resource** *generalizes* **Synthesized Service**
- A **2nd Order Resource** *generalizes* **Assembly**

Explanation

In order that the Palpable Architecture be flexible and robust enough to construct systems capable of effectively responding to change while maintaining the user's palpable experience, most entities are considered to be 2nd Order Resources which can be reasoned about and manipulated accordingly. These include components, services, assemblies, devices, communication channels, and actors. By considering each of these as resource abstractions, developers of palpable systems can more consistently apply well known resource management patterns [26] and treat contingency issues. By definition, many 2nd Order resources naturally contain, consume and consist of 1st Order Resources. Actors are considered as 2nd Order Resources because the computational aspect of palpable systems cannot and should not be considered in isolation from its participating (human) users.

Every 2nd Order Resource can be a member of an assembly and will have an associated Resource Descriptor.

3.2.3 Actor

Definition

An Actor is either a human or some other system external to an instantiation of the Palpable Architecture that can be perceived as a user and/or participant in of all or part of that instantiation.

Relationship to other concepts

- An **Actor** *specializes* **2nd Order Resource**
- An **Actor** *use zero or more* **Assemblies**
- An **Actor** *interacts with zero or more* **2nd Order Resource**
- An **Actor** *can control zero or more* **Assembly Manager**
- An **Actor** *can control zero or more* **Resource Manager**
- An **Actor** *can control zero or more* **Contingency Manager**

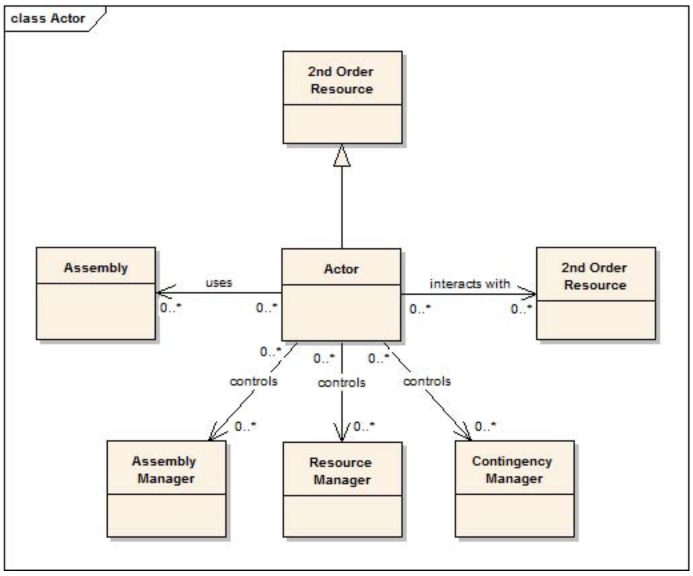


Figure 3.3: Actor ontological relationships

Explanation

The relationship between an Actor and the various parts of a computational system is of central importance to field of Palpable Computing. Referencing the PalCom Conceptual Framework [62], an Actor resides at or above the *surface* of an implemented palpable system using assemblies, interacting with 2nd Order Resources (e.g., services, devices, assemblies, etc.), and controlling middleware managers. However, Actors must also be considered as participants in assemblies, potentially recognized in terms of the constraints they place on assembly construction or operation.

Although an Actor is typically a human user, it may also potentially be some other system external to an instantiation of the Palpable Architecture that can be perceived as a user of all or part of that instantiation. This architecture makes no direct distinction between human and non-human Actors.

3.2.4 Assembly

Definition

An Assembly is an organized collection of 2nd Order Resources composed in such a way as to deliver all or part of some application functionality.

Relationship to other concepts

- An **Assembly** *specializes* **2nd Order Resource**
- An **Assembly** *consists of* **2nd Order Resources**
- An **Assembly** *is described by an* **Assembly Descriptor**
- An **Assembly** *is deployed on one or more* **Device**

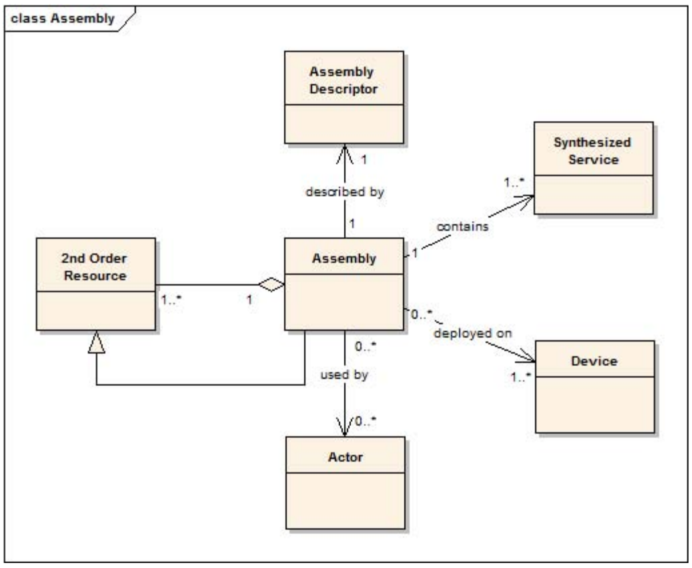


Figure 3.4: Assembly ontological relationships

- An **Assembly** *contains one or more* **Synthesized Service**
- An **Assembly** *used by zero or more* **Actor**

Explanation

The Assembly is the primary point of interaction between a user and a palpable system and consists of an organized collection of 2nd Order Resources, potentially including other Assemblies. The organization is typically an orchestrated composition of services exposed as Synthesized Services (see Section 3.2.16), that may, or may not, be set at deployment time and may, or may not, change dynamically over time. While some second order resources, e.g., services indigenous to a device, necessarily must be located locally to the assembly using them, e.g., a communication channel abstraction over some hardware specific communication for used in an assembly to enable remote communication, other resources can also be located remotely. Thus an assembly may be distributed over multiple physical devices and if so will include the definitions of Communication Channels required to connect the remote parts. The actual location of a second order resource exploited in an assembly can be defined in the descriptor of the assembly.

The composition of an assembly may be fixed for the duration of the assembly instantiation, or it may change dynamically, both in terms of the individual resources and the way in which they are composed. The baseline functionality of the assembly should remain stable throughout its lifetime in order to offer a guarantee that originally expected functionality remains available at all times.

In addition to the composition template, the specification of what an assembly is intended to achieve, how it will achieve it, what preconditions are necessary and what output and/or postconditions will be affected, is contained in an assembly's Assembly Descriptor (see Section 3.2.5). It is recommended, but not mandated, that an assembly define only one Synthesized Service in order to restrict complexity. An assembly is typically deployed according to a template that defines the resource composition, but without any specific resource instantiations initially bound.

Once an assembly has been created it takes on a new role in its life-cycle; *maintenance*. Maintenance of assemblies includes *Deconstruction*, when the assemblies are no longer needed, and *re-construction* when an assembly is subject to a partial failure that possibly can be resolved by dynamically replacing parts of the assembly with other (in the operating environment) available second order resources equivalent to those failing.

When an assembly is no longer needed, it can be decomposed. Doing so is initiated based on action by the actor. Decomposing an assembly simply involves unbinding the second order resources bound together according to the assembly descriptor. However, any synthesized services used in the assembly are left unaffected as they might still be used in other assemblies.

3.2.5 Assembly Descriptor

Definition

An Assembly Descriptor describes the organization of an Assembly in terms of its constituent elements, what the assembly is intended to achieve, how it will achieve it, what preconditions are necessary and what output and/or postconditions will be affected.

Relationship to other concepts

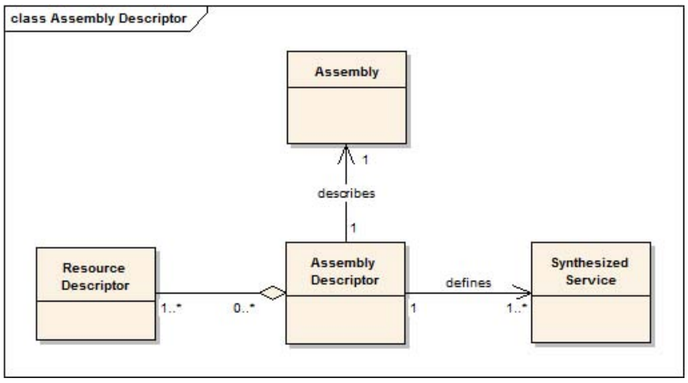


Figure 3.5: Assembly Descriptor ontological relationships

- An **Assembly Descriptor** *describes an* **Assembly**
- An **Assembly Descriptor** *contains one or more* **Resource Descriptor**
- An **Assembly Descriptor** *defines one or more* **Synthesized Service**

Explanation

An Assembly Descriptor describes everything relating to the structure, organization and properties of an Assembly that a user of the assembly needs to know concerning how to make use of it and what to expect in terms of functionality. An Assembly Descriptor describes this in terms of the static, dynamic, passive and active nature of individual elements and their functional relationships to one another. This implies that an Assembly Descriptor will contain one definitions of Synthesized Services that will be offered by the corresponding Assembly and one or more Resource Descriptors, or references to them.

An assembly is constructed according to an assembly descriptor associated with it. The assembly descriptor is a platform independent XML-based file, which defines what second order resources should take part in the assembly. The assembly descriptor defines which second order resources are needed for the assembly to provide its baseline functionality. The presence of the second order resources for maintaining the baseline functionality is mandatory; otherwise the assembly will be in a defunct state. The assembly descriptor might also define programmatic constraints that guide the extension of baseline functionality through the addition of new second order resources. Such extension is however assured not to impact the stability of the baseline functionality.

Associating second order resources to an assembly can be achieved by matching second order resources available in the operating environment of the assembly with the requirements defined in the descriptor. The requirements can be specified in terms of functionality properties, location, and identity. Using these tolerance specifications, the assembly is constructed from the second order resources present in the operating environment.

In addition to defining *which* second order resources are required for an assembly, the assembly descriptor also defines *how* (and in which order) these second order resources should interact in order to achieve the functionality represented by the assembly.

3.2.6 Assembly Manager

Definition

An Assembly Manager is responsible for constructing, deploying and managing the lifecycle of Assemblies.

Relationship to other concepts

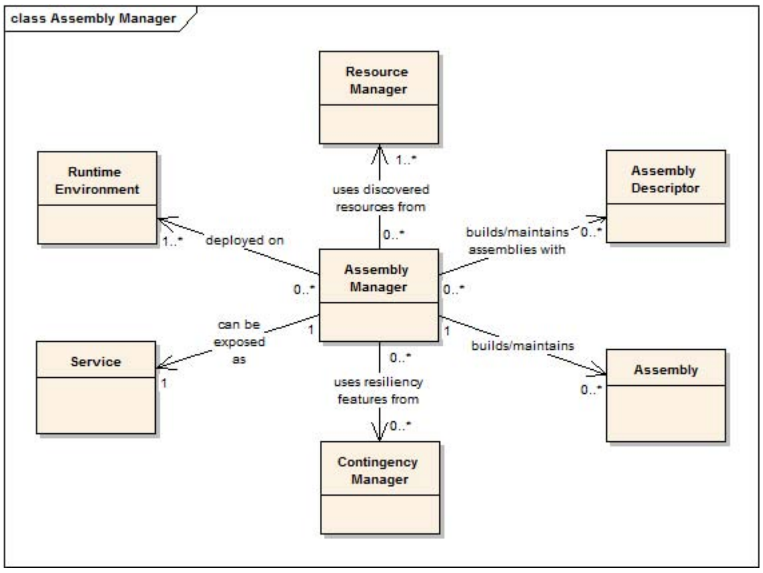


Figure 3.6: Assembly Manager ontological relationships

- An **Assembly Manager** *uses discovered resources from* **Resource Managers**
- An **Assembly Manager** *builds/maintains assemblies with* **Assembly Descriptors**

- An **Assembly Manager** *builds/maintains zero or more* **Assembly**
- An **Assembly Manager** *uses resiliency features from* **Contingency Managers**
- An **Assembly Manager** *can be exposed as a* **Service**
- An **Assembly Manager** *deployed on one or more* **Runtime Environment**

Explanation

Every Assembly will optionally have one active Assembly Manager residing on one Device participating in the assembly. Other inactive Assembly Managers may also be present within the assembly, especially to support distributed replication and failover redundancy. Such an inactive Assembly Manager would then be activated on failure of a previously active Assembly Manager.

Although it is conceivable that multiple Assembly Managers may be active within the same assembly, this requires special coordination and is thus not recommended. Interaction between Assembly Managers governing different Assemblies is possible however.

As mentioned, the first duty of an Assembly Manager is the construction of Assemblies according to manual, or task-based specification. In either case an underlying XML-encoded Assembly Description will contain details of the particular 2nd Order Resources to be used within the assembly and scripting of the contained Synthesized Service(s) which will make use of those resources to deliver composed functionality. Both templates and instances of 2nd Order Resources, and specifically Services, can be specified in the Assembly Description.

Templates abstract the choice of, for example, a Service, to the type of that Service rather than a specific instance. This allows loose assembly binding such that if a template is used, then the selection of a service instance can be delegated to the Resource Manager by sending a request for a service instance conforming to the template constraints. This can include, for example, the particular 1st Order Resources required on a Device that might be hosting a Service meeting the required template criteria.

Moreover, in forming Assemblies the resource requirements of the parts of an assembly are gathered and assessed in order to take informed decisions about the parts of an assembly considering resource constraints and preferences. The establishment of the assembly itself is also part of the resource management system, which does this based on a Assembly Descriptor, which contains the resource types required for the assembly as well as the work-flow descriptor.

Subsequent to construction, an Assembly Manager is then responsible for deploying the Assembly and thereafter managing its lifecycle in collaboration with one or more available Contingency Managers. The primary features of Assembly lifecycle management as performed by an Assembly Manager are as follows:

- Evolution of the assembly through the dynamic removal of existing resources, or inclusion of new resources.
- Monitoring of assembly behaviour.
- Recognition of problems, faults or failures in the assembly.
- Issuing of events to other middleware managers including, for example, those to a Contingency Manager relating to problems, faults or failures.
- Maintenance of the assembly in accordance with contingency actions and/or reccommendations.
- Describing its own behaviour when inspected.

3.2.7 Communication Channel

Definition

A Communication Channel provides a means of communication between Services. The concept incorporates the notions of communication medium (access type or bearer) and communication protocol.

Relationship to other concepts

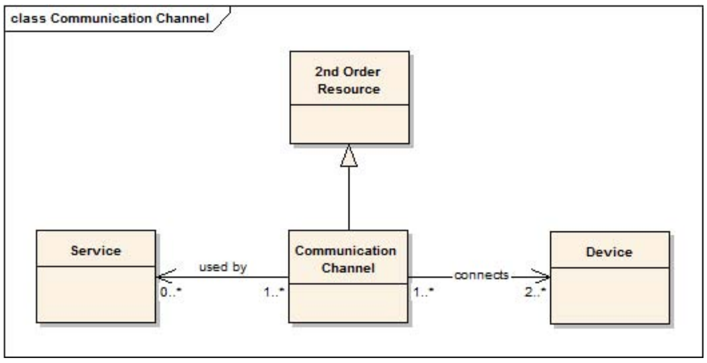


Figure 3.7: Communication Channel ontological relationships

- A **Communication Channel** *specializes a* **2nd Order Resource**
- A **Communication Channel** *connects two or more* **Device**
- A **Communication Channel** *used by* **Services**

Explanation

The Communication Channel offers an abstraction over the various types of network technology available to transfer information from one point to another. This abstraction allows an application to simply send/receive a message without being concerned which bearer technology, protocol or particular network configuration is used. The communication stack will then take care of delivering the message in accordance with separately definable policies, or simply choose the most readily available channel. Naturally, if preferred, an application can directly specify the communication channel and its configuration.

A Communication Channel is always associated with a physical device, i.e., Device.

Communication between localized (on the same Device) Runtime Components will typically not use a Communication Channel, but rather a form of direct invocation.

3.2.8 Contingency Manager

Definition

A Contingency Manager is responsible for resolving fault and problem conditions occurring in active Assemblies and 2nd Order Resources through the application of a variable set of contingency tools and mechanisms.

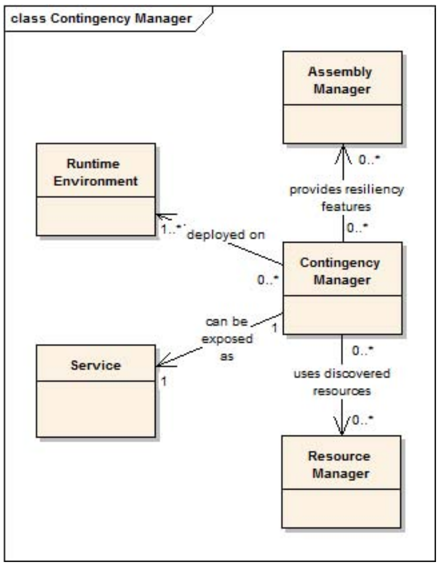


Figure 3.8: Contingency Manager ontological relationships

Relationship to other concepts

- A **Contingency Manager** *provides resiliency features to* **Assembly Managers**
- A **Contingency Manager** *uses discovered resources from* **Resource Managers**
- A **Contingency Manager** *can be exposed as a* **Service**
- A **Contingency Manager** *deployed on one or more* **Runtime Environment**

Explanation

Every Assembly will optionally have one active Contingency Manager residing on one Device participating in the assembly. Other inactive Contingency Managers may also be present within the assembly, especially to support distributed replication and failover redundancy. Such an inactive Contingency Manager would then be activated on failure of a previously active Contingency Manager.

Contingency implies the ability of a palpable system to automatically identify problem conditions, determine suitable means to resolve them and then apply appropriate mechanisms to prevent future error conditions. Therefore, for example, a temporarily lost network connection does not necessarily lead to an error condition, because it has to be considered as a legal operating state in system design. This not only ensures that a system becomes more resilient to failure, but also capable of adapting to ambient conditions such as resource starvation. The consideration of resource management is therefore also considered as a required step towards the establishment of a contingency paradigm for palpable systems.

For the purposes of contingency management a dinstinction is drawn between *errors*, *faults* and *failures*. An error is an exception condition resulting from some deviation from expected behaviour leading to a fault or failure. A fault is a non-catastrophic breakdown from which recovery is expected, and a failure is a serious condition from which recovery may not be readily possible.

The primary operations of a Contingency Manager are triggered by incoming events sourced from Assembly and Resource Managers. Also, in some cases Contingency Managers can offer reduced, but specialized functionality and operate in coordination with other specialized Contingency Managers to offer a complete, although distributed, service. The goal is for the assembly to remain stable over time and as such contingency management ensures that second order resources in the assembly are always available or, if not, that replacements are found and adopted *on the fly* to ensure that the baseline functionality of the assembly, i.e., the minimum set of functionality required to satisfy the assembly description, is always maintained within its defined tolerances (according to the assembly descriptor).

This ability to replace a resource can also be used to improve or extend an assembly. For example, if a new resource becomes available that matches some functionality already used within the assembly but with a better quality of service, that resource may be used to replace the original functionality. Naturally the converse is also true, whereby a resource may be replaced to reduce service quality if consistent with the functional requirements of an assembly.

Another aspect of this control over assembly constitution is allowing an assembly to optionally be extended with additional functionality through the addition of new resources, including the possibility of additional synthesized services. Extensions can be adopted or disposed of dynamically without affecting the *stability* of the baseline assembly. The extensions can be made by the user directly or by the Assembly Manager which can seek, discover and install/uninstall extensions according to programmatic constraints defined in the assembly descriptor. If it is desired that a temporary extension of an assembly become permanent, a new assembly is spawned consisting of the baseline functionality of the original assembly plus the added extensions. The original assembly remains intact.

While the goal of *stability* is to make the assembly appear consistent and functioning to the actors, from the point in time that a partial failure of the assembly is detected, there might be introduced non-deterministically long pauses in operation response times. Locating a replacement second order resource, might not be trivial nor possible for some time. Thus, while still appearing consistent, the assembly might actually be in a failure mode looking for replacement resources, which thus might cause disruption in its functionality provisioning.

Contingency Managers are expected to at least provide a set of *reactive contingency actions* (i.e., compensations) which respond to errors, faults and failures when an event is received indicating their occurrence. Events are typically sourced from Assembly and Resource managers.

Baed on the reception of events, the following primary operations characterize the reactive behaviour of a Contingency Manager:

- Monitor the performance thresholds of 1st Order Resources on specified (by events) devices. If a threshold is passed, a contingent action can attempt to trigger the re-balancing of resource load across additional devices.
- Compensate for an error/fault/failure with an Assembly-specific resource by attempting to locate an equivalent replacement resource.
- Compensate for the error/fault/failure with an Assembly-specific resource by attempting to reconfigure an assembly in coordination with the Assembly Manager.
- If replacement and reconfiguration fail then compensate for the error/fault/failure with an Assembly-specific resource by attempting to gracefully degrade the operation of an assembly in coordination with the Assembly Manager.
- Resolve dependencies in accordance with the mode and subject of a compensation. For example, a replaced service must be configured to match any residual dependencies remaining from its predecessor.
- Describing its own behaviour when inspected.

More advanced Contingency Managers may also provide *proactive contingency actions* which attempt to plan strategies for dealing with errors, faults and failures *before* they occur. Strategies can be proposed to actors allowing them the opportunity to override them if preferred. Some of the optional operations include:

- Establishing proactive precautionary error avoidance strategies which will attempt to mitigate potential error conditions: e.g., failure of a service, a device, network failures etc. by anticipating them and putting contingent actions in place as precautionary measures. These actions may either simply raise exception warnings, or may attempt to divert from an anticipated error condition into a safe state of operation.
- Immunising against known problem conditions by recording events and learning from past experiences. Using this knowledge structural (e.g., assembly formation) or behavioural (e.g., task formulation) adaptation can be applied to eliminate or at least reduce the chances of an error condition occuring. By these means a system optimizes its own operation.
- Preparation for the graceful degradation of the integrity of active Assemblies by containment of detected problem or failure conditions. This is achieved by attempting to isolate potential sources of errors, faults or failures and determine means of allowing systems to continue running in states of limited functionality while safely degrading if necessary.
- Restitution of system integrity through self-healing adaptation and external intervention (if necessary) to recover lost functionality and state.

The Contingency Manager is an optional middleware manager although it's use is highly recommended.

3.2.9 Device

Definition

A Device is an item of computational hardware that hosts one or more Runtime Environments and within each, one or more components, services or assemblies.

Relationship to other concepts

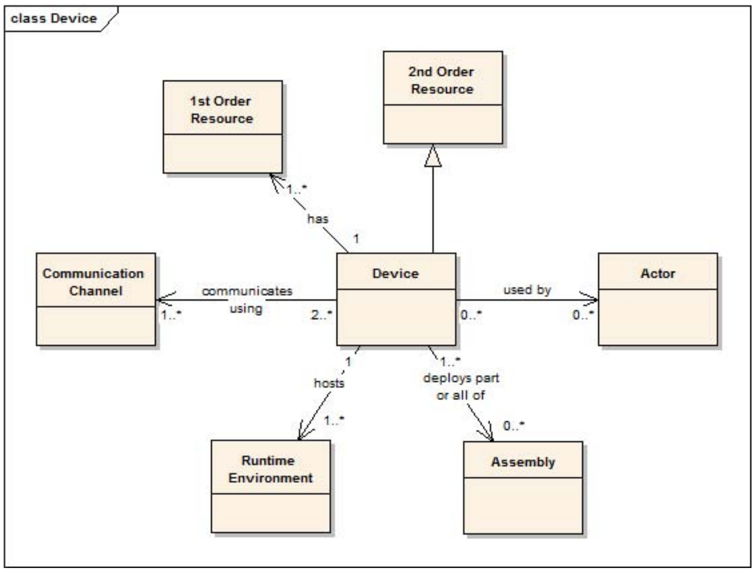


Figure 3.9: Device ontological relationships

- A **Device** *specializes a* **2nd Order Resource**
- A **Device** *has* **1st Order Resources**
- A **Device** *communicates using one or more* **Communication Channel**
- A **Device** *hosts one or more* **Runtime Environment**
- A **Device** *deploys all or part of zero or more* **Assembly**
- A **Device** *used by zero or more* **Actor**

Explanation

Explanation

Devices are nominally the physical, or virtual devices with which users interact with palpable systems. They are capable of hosting one or more Runtime Environments and all or part of one or several Assemblies (see Section 3.2.4).

3.2.10 Resource

Definition

A Resource is a logical abstraction of 1st Order Resource and Second Order Resource and represents some element of a palpable system that is limited and consumable by other elements of the system.

Relationship to other concepts

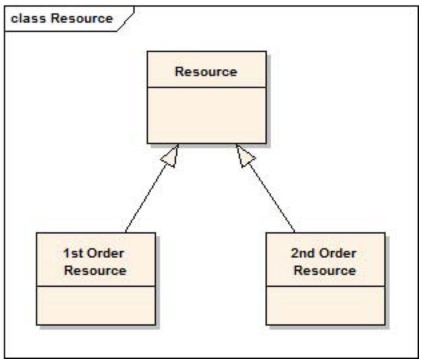


Figure 3.10: Resource ontological relationships

- A **Resource** *generalizes* **1st Order Resource**
- A **Resource** *generalizes* **2nd Order Resource**

Explanation

A Resource is a convenient logical abstraction over 1st Order Resources (e.g., memory and bandwidth), and 2nd Order Resources (e.g., Devices, Assemblies and Services).

Resources are considered as scarce commodities, may be shared and must be managed effectively to ensure proper behaviour of palpable systems.

3.2.11 Resource Descriptor

Definition

A Resource Descriptor is used to describe the characteristics of any entity that can be classified as a 2nd Order Resource (e.g., Device, Assembly, Service, etc.).

Relationship to other concepts

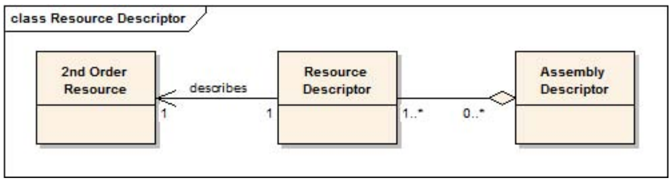


Figure 3.11: Resource Descriptor ontological relationships

- A **Resource Descriptor** *describes a* **2nd Order Resource**
- A **Resource Descriptor** *can be a constituent of zero or more* **Assembly Descriptor**

Explanation

A Resource Descriptor is essentially a two-level hierarchy of parameters. The top level contains those parameters that are common to all resources, such as identity and type (e.g., type=Service).

The lower level extends this common level with parameter specializations according to the specific resource type. For example, a Service will define at least a signature and accessor interface that describes what the service offers, requires and how it can be accessed.

3.2.12 Resource Manager

Definition

A Resource Manager is responsible for maintaining an up-to-date directory of discoverable 2nd Order Resources.

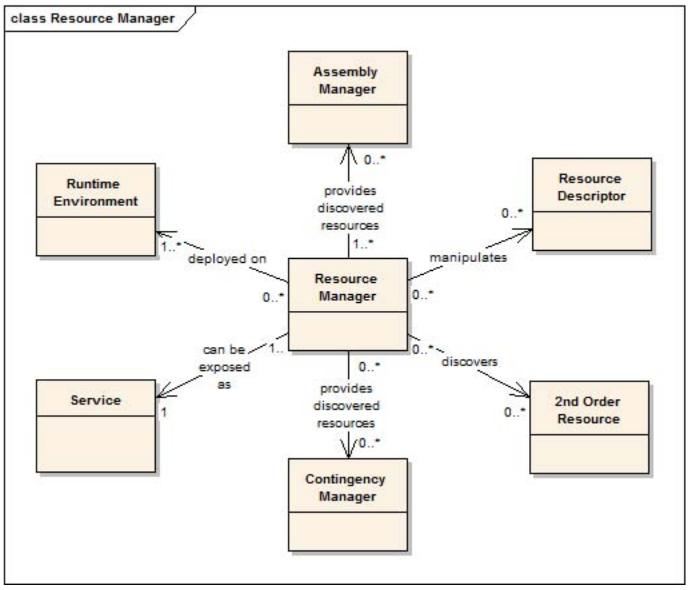


Figure 3.12: Resource Manager ontological relationships

Relationship to other concepts

- A **Resource Manager** *provides discovered resource to* **Assembly Managers**
- A **Resource Manager** *manipulates* **Resource Descriptors**
- A **Resource Manager** *discovers* **2nd Order Resources**
- A **Resource Manager** *provides discovered resources to* **Contingency Managers**
- A **Resource Manager** *can be exposed as a* **Service**
- A **Resource Manager** *deployed on one or more* **Runtime Environment**

Explanation

Every Assembly will nominally have one active Resource Manager residing on each Node participating in the assembly. Other inactive Resource Managers may also be present within the assembly, especially to support distributed replication and failover redundancy. Such an inactive Resource Manager would then be activated on failure of a previously active Resource Manager.

The primary operations of a Resource Manager are as follows:

- Discover 2nd Order Resources
- Persistently monitor previously discovered 2nd Order Resources.
- Match resources according to needs expressed by Assembly and Contingency Managers.
- Describe available 2nd Order Resources to Assembly and Contingency Managers.
- Provide available 2nd Order Resources to Assembly and Contingency Managers.

- Describing its own behaviour when inspected.

Some optional operations that may be performed by a Resource Manager include:

- Reserving resources for use at some point in the future for a specified duration. Quality of service guarantees can be assigned to provide a measure of certainty with respect to the availability of a resource.
- Negotiate with other Resource Managers for access to a particular 2nd Order Resource.
- Predicting resource requirements based on analysis of previous resource requests received from Assembly Managers. This is coupled with resource reservation to automatically reserve resources if a request for use is anticipated.
- Manage the lifecycle of selected resources including initiating, terminating, restarting and moving, etc. 2nd Order Resources.
- Arbitrate conflicts over resource access and usage between Assembly Managers.

3.2.13 Runtime Component

Definition

A Runtime Component is an executable instantiation of a software component.

Relationship to other concepts

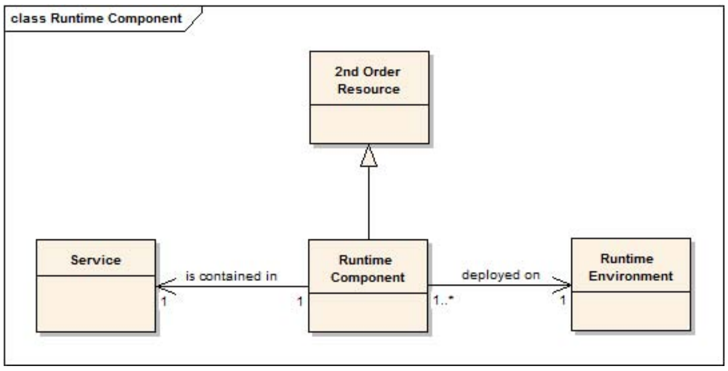


Figure 3.13: Runtime Component ontological relationships

- A **Runtime Component** *specializes a* **2nd Order Resource**
- A **Runtime Component** *is contained in a* **Service**
- A **Runtime Component** *deployed on a* **Runtime Environment**

Explanation

Runtime Components are instantiated software components that can be executed on a Runtime Environment. They also form the functional aspect of a Service, which in turn extends a Runtime Component with the ability to communicate remotely (i.e., non-local direct invocation).

3.2.14 Runtime Environment

Definition

The Runtime Environment provides the basic functionality required to host, execute and support the distributed communicative interaction of assemblies and the constituents thereof.

Relationship to other concepts

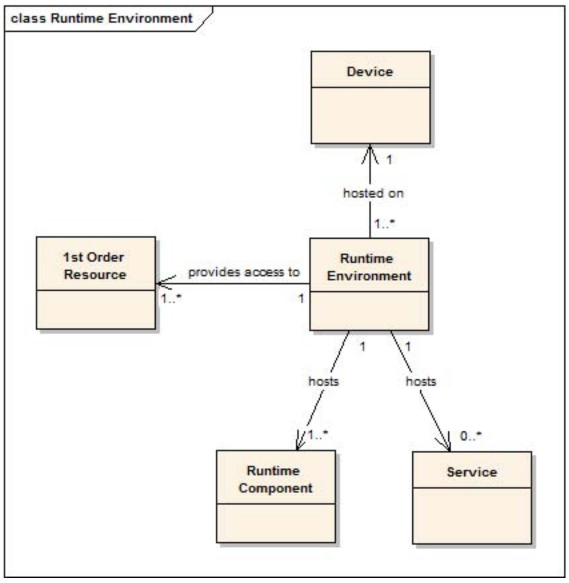


Figure 3.14: Runtime Environment ontological relationships

- A **Runtime Environment** *is hosted on a* **Device**
- A **Runtime Environment** *provides access to* **1st Order Resources**
- A **Runtime Environment** *hosts zero or more* **Service**
- A **Runtime Environment** *hosts one or more* **Runtime Component**

Explanation

The primary responsibility of the Runtime Environment is to facilitate the hosting and execution of Assemblies (and thereby all 2nd Order Resources) on devices. At least one instance of the runtime environment must be physically located on a Device for that Device to be considered a Device. Each runtime nominally contains a virtual machine.

Each Runtime Environment offers two main functionalities: a component infrastructure that hosts and executes Runtime Components, Services, and Assemblies, and a communication infrastructure for inter-device communication. Selected core components, including those that reify the Assembly, Resource and Contingency Managers, are also hosted on a runtime.

3.2.15 Service

Definition

A Service contains a Runtime Component coupled with the means to remotely communicate with other services, e.g., announcement, discovery, invocation. A Service is self-contained, can maintain state and is always expected to execute on a Runtime Environment.

Relationship to other concepts

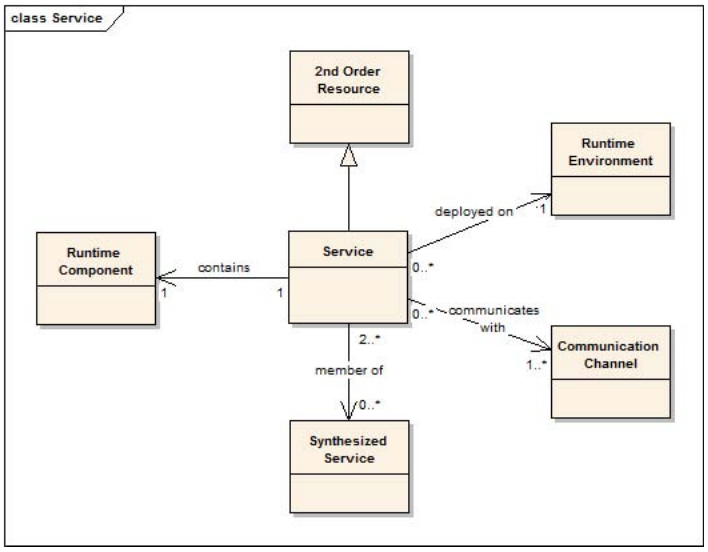


Figure 3.15: Service ontological relationships

- A **Service** *specializes a* **2nd Order Resource**
- A **Service** *is deployed on a* **Runtime Environment**
- A **Service** *contains a* **Runtime Component**
- A **Service** *communicates with one or more* **Communication Channel**
- A **Service** *may be a member of zero or more* **Synthesized Service**

Explanation

Services are the primary unit of distributed computation available to the creators and users of palpable applications. They contain Runtime Components, can be composed into Synthesized Services (see Section 3.2.16), and are the primary constituents of Assemblies. Services can only be executed on Runtime Environments.

When instantiated, a Service can be advertized, discovered and accessed from local and remote locations if at least one means of remote communication is available via a Communication Channel. Each Service is considered to be self-contained in that any functionality expressed by the interface signature is available within the service itself. Dependence on other services is only present at the level of relations that may be specified in the service's Resource Descriptor or an Assembly Descriptor. Services are reactive, or indeed proactive, to received events that conform to its published interfaces.

A Service can be either stateless or stateful depending on usage requirements. Stateless implies that session specific state is not maintained across multiple invocations of a service and stateful implies that it is. When possible it is recommended that services are implemented as stateless in order to minimize impact on the performance and scalability of the corresponding palpable System.

A Service can optionally belong to one of two sub-categories. A Service is *migratable*, if it has been programmed to be able to capture its state (if it has any), such that the instance of the service can be moved from one Device to another. Contrary, a Service is said to be *indigenous* if it is bound to the underlying hardware platform in such a way that it can only run on this particular device. A Service does not have to be neither *migratable* nor *indigenous*.

A Service can either be a classic reactive service, or may be capable of autonomous, proactive behaviour, or some combination thereof.

3.2.16 Synthesized Service

Definition

A Synthesized Service is a service interface created by the composition of zero or more Services as described by an Assembly Descriptor and contained in the corresponding Assembly.

Relationship to other concepts

- A **Synthesized Service** *specializes a* **2nd Order Resource**
- A **Synthesized Service** *defined in an* **Assembly Descriptor**
- A **Synthesized Service** *composed of zero or more* **Service**
- A **Synthesized Service** *contained within an* **Assembly**

Explanation

A Synthesized Service is the primary means by which a Assembly exposes functionality created by the composition of zero or more Services. It is essentially a service interface described by an Assembly Descriptor that defines the entry point into a composition of Services. A Synthesized Service therefore only offers new functionality by virtue of the composed services that it represents offering the equivalent to, or in some cases more than, the "sum of the constituent parts".

In addition, every Synthesized Service is a kind of 2nd Order Resource and as such has a Resource Descriptor that describes its characteristics in the same manner as a normal Service.

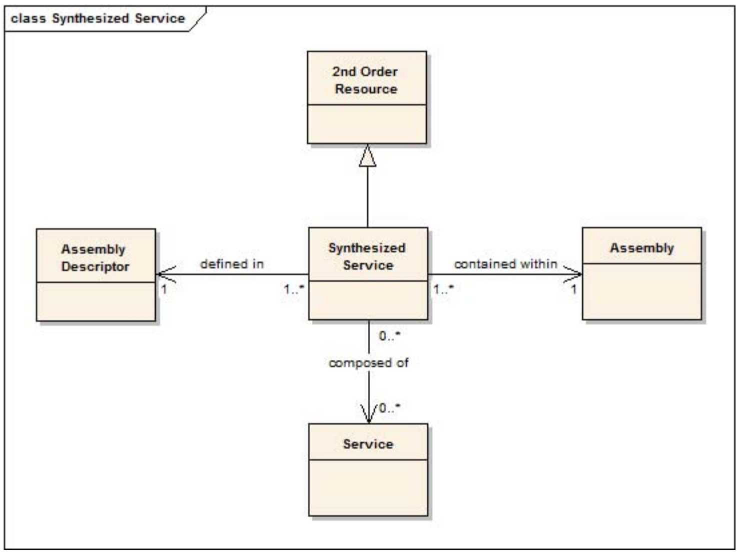


Figure 3.16: Synthesized Service ontological relationships

Part II

The PalCom Architecture

Chapter 4

A Computational Infrastructure to Realize Palpable Computing

This part of the document presents a series of chapters detailing a computational infrastructure that realizes the architectural concepts of Palpable Computing identified and discussed in Part I. Each chapter describes the reification,in our implementation, of one or more architectural concepts in accordance with the previously stated quality attributes of palpable computing.

The infrastructure is shown diagrammatically in Figure 4.1.

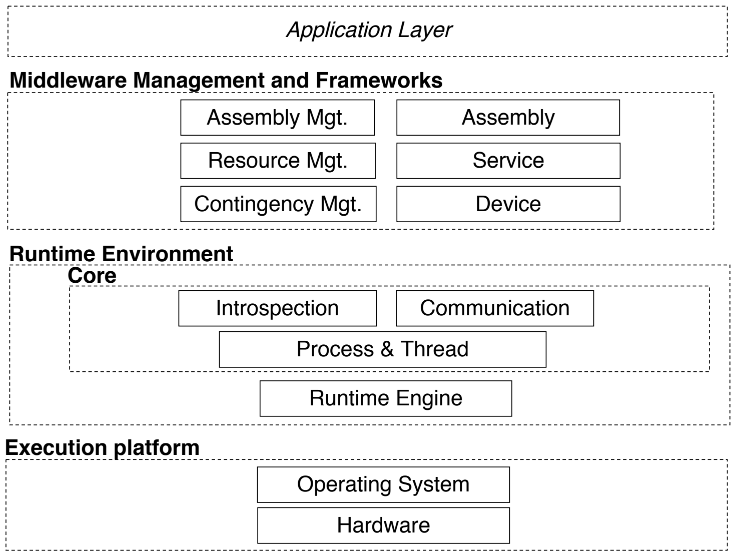


Figure 4.1: Layers and Functional Elements the PalCom Infrastructure. Elements in one layer is only dependent on lower layers. Note that the boxes represent functional blocks of code at runtime. That means the *Service* box does not represent the 2nd order resource Service, but instead the logical block of runtime functionality deriving from the source code in the service framework. The relationships between the elements in this figure and the ontology is explained separately in table 4.1. Notice that not all elements are mandatory e.g. the operating system may or may not be present, see Section 8.

Figure 4.1 consists of four areas that combine several computational aspects into logically associated groups. These groups are:

Application Layer There are no traditional applications in palcom. Instead assemblies and services are the units of delivery for functionality the user can apply. They build on the functionality and support provided by the palcom archtecture in the lower layers.

Middleware Management and Frameworks The special runtime components responsible for managing the functional computation, typically assemblies, that constitute the application logic of any palpable system. The concepts defined in the architecture for palpable computing and realized by this group are:

- Assembly Manager
- Resource Manager
- Contingency Manager

In addition, the architecture provides three frameworks to ease integration of new services, devices and assemblies with these managers (the frameworks are described in chapter 5):

- Assembly Framework
- Service Framework
- Device Framework

Runtime Environment The *Runtime Engine* is the virtual machine and a set of core libraries for low-level structuring of programs. This layer include the following logical units of functionality:

- Introspection, implemented using Hierarchical Graphs
- Communication, providing basic abstractions and functionality for network communication.
- Process & Thread, for managing concurrency and separation of address spaces.
- Runtime Engine, which includes a Virtual machine.

Execution Platform The underlying device hardware and systems on which palpable computation is hosted. The concepts defined in the architecture for palpable computing and realized in this group are:

- 1st Order Resource

Concept	Realized in...
1st Order Resource	The hardware part of the execution platform.
2nd Order Resource	For discovery, in the Resource Manager.
Actor	Not shown. Would be above the application layer.
Assembly	The Assembly framework and manager.
Assembly Descriptor	The assembly manager and tools at the application layer.
Assembly Manager	Middleware manager and framework layer.
Communication Channel	The communication core library block of the Runt. Env.
Contingency Manager	Middleware manager and framework layer.
Device	The device framework in the middleware manager and framework layer.
Resource	Not explicitly reified, see 1st and 2nd Order Resources.
Resource Descriptor	Resource manager, see also assembly descriptor.
Resource Manager	Middleware manager and framework layer.
Runtime Component	The runtime environment.
Runtime Environment	Core libraries and runtime engine.
Service	The service framework. See also 2nd order resource.
Synthesized Service	Assembly framework and manager.

Table 4.1: Reification of the PalCom ontology concepts

Table 4.1 summarizes how each element of the ontology is realized in terms of elements of the runtime layered view.

I the table, the right column describes which functional parts are influenced by that concept and therefore are part of its programmatic manifestation. Note that although e.g. a service is discoverable by the resource manager, that is not shown directly in the realization-entry for *Service*. Instead, it is implied because a service is a second order resource and that concept has its own entry in the table. Note in addition that since all elements may be made available for inspection in the hierarchical graphs (described in section 7.3), their representation contained herein are also part of the runtime manifestation for each concept.

The infrastructure has been implemented through a related collection of architectural prototypes developed throughout the duration of the PalCom project and reported on in the M36-37 Deliverables (i.e., Deliverables 39-44).

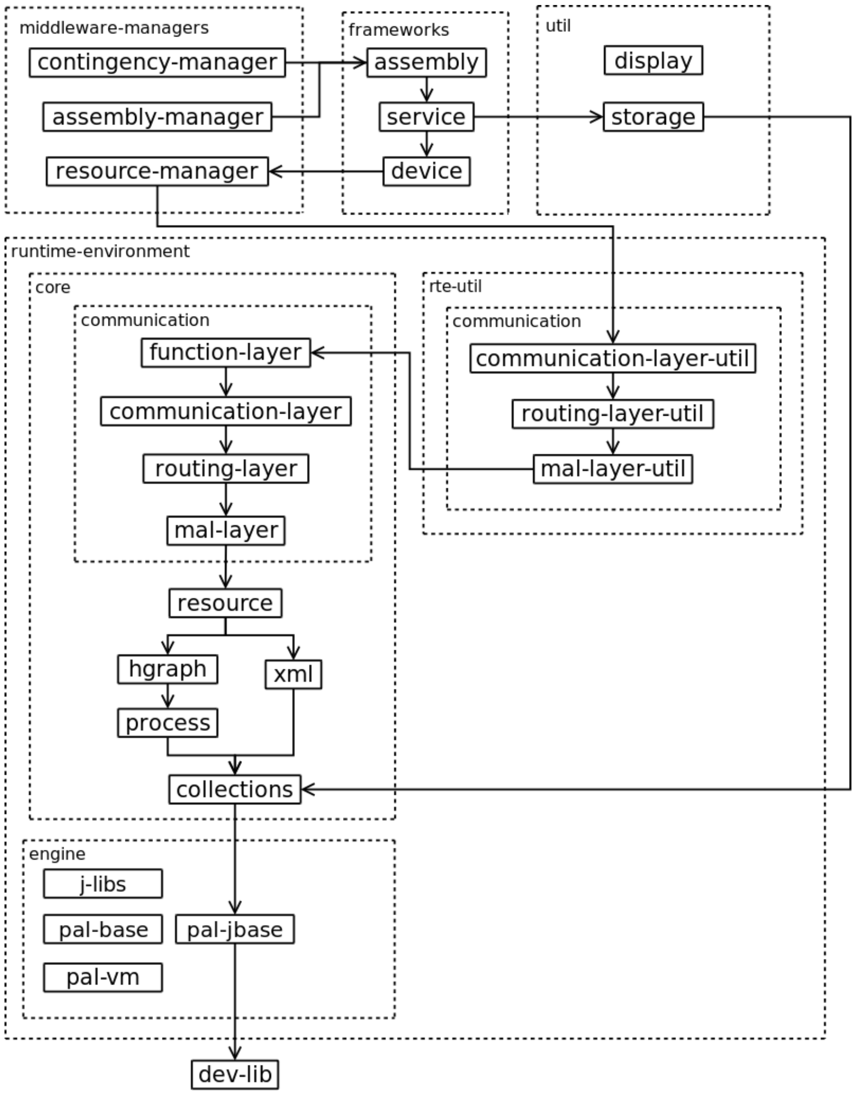


Figure 4.2: Module view of the PalCom infrastructure. The arrows between the modules denote compilation dependencies.

In terms of the software structure in the PalCom codebase, Figure 4.2 presents a slightly more detailed module view of the infrastructure. The majority of the modules map directly onto grouped areas in Figure 4.1 with the following extensions: middleware management is shown as being realized by a set of frameworks, the runtime environment contains an rte-util module that provides some communication-specific utilities, and the util module contains some additional utilities of occasional use within some application domains.

The individual modules are explained in subsequent sections and in the Developer's Companion[71].

Chapter 5

Resource Frameworks

To ease the programmer's task of creating new Services, Assemblies and Devices, a framework is provided for each. The Service and Device frameworks are used by subclassing *AbstractDevice* and *AbstractService* respectively. The classes in the Assembly framework are often used as they are for executing assemblies that have been parsed from XML, but it is also possible to subclass the *Assembly* class for implementing an Assembly directly in Java.

For other resources, including communication channel and the managers, we have not found it necessary to provide a framework. That is because it is rarely necessary to created new types of these resources, although it is possible to do so.

5.1 Service

The Service framework contains classes for implementing services.

A Service is accessible to other instantiated services only through its published interfaces which are available either through direct discovery or via a unique service description which reifies the Resource Descriptor defined in Section 3.2.11. Service descriptors can be used as the means to discover Services and are recorded by the Resource Manager, described in Section 6.1.

This service framework is not used to make synthesized services, as defined in Section 3.2.16 and in Section 5.2, because they are scripted compositions of Services which typically defines the core functionality of an Assembly.

In the Service framework there is also a Service Manager which handles dynamic instantiation of Services from software components. The binary format of the components depends on whether the `pal-vm` or the JVM is used in the Runtime Environment (see Section 7).

Further detailed elaboration of services and especially PalCom Services, their creation, management and use, may be found in *Deliverable 41* [64] and *Deliverable 42* [65], *Deliverable 43* [66] respectively, as well as in the Developer's Companion [71].

5.1.1 Building Services

AbstractService is the superclass of all services written in the framework. It has name and address information about the service.

Figure 5.1 shows as an example the basic structure of asimple service from the MIO prototype. It is programmed to be accessed through point to point messaging. In this case the messages are handled by a separate thread which applies the PalcomThread class' mail box functionality to receive messages from the communication manager (not shown in the figure). The *start()* method is executed when the service is started which is usually but not necessarily right after instantiation, and the *stop()* method is, akin to a destructor for classes in many languages, executed when the service is shut down.

```
package ist.palcom.ais;

import ist.palcom.ais.source.AISSource;
import ist.palcom.device.DeviceContext;
import ist.palcom.resource.descriptor.Command;
import ist.palcom.resource.descriptor.Param;
import ist.palcom.resource.descriptor.ServiceProxy;
import ist.palcom.services.AbstractService;

...

public class AISReaderService extends AbstractService {
  public static final String SERVICE_NAME = "AIS Reader Service";
  private AISSource reader;
  ...

  public AISReaderService(DeviceContext context, String urnLeaf,
      AISSource source) throws IOException {
    super(context, urnLeaf, SERVICE_NAME, createServiceProxy());
    reader = source;
    reloadShipFile();
  }

    @Override public void start() throws IOException {
      reader.start(new AISReceiver());
      maintenanceThread = new Thread(new MaintenanceThread());
      maintenanceThread.start();
      super.start();
    }

    @Override public void stop() throws IOException {
      reader.stop();
      super.stop();
    }
    ...
}
```

Figure 5.1: Code for a service from the MIO prototype.

5.2 Assembly

A PalCom Assembly is a composition, or orchestration, of PalCom 2nd Order Resources (see Section 3.2.2). It consists primarily of Services and provides some additional end-user functionality beyond that provided by the individual services. In addition to orchestrating services, an Assembly can provide one or more Synthesized Services that each defines a service interface, offering access to a specific composition of some or all of the Services contained by the assembly. An example of an Assembly can be one that combines a photo service on a camera device with a coordinate service on a GPS device. The functionality of the assembly is to tag images delivered by the photo service with GPS coordinates from the coordinate service. A Synthesized Service provides the tagged images to users of the assembly.

PalCom Assemblies are relevant both for the developer of PalCom systems as a means to build new functionality as part of planned development effort, as well as end-users of PalCom systems to dynamically tailor and augment the behaviour of the system within a concrete context. In cases where a PalCom Assembly interacts directly with a PalCom Actor, it may act as a user-level application that also maintains Actor-specific (session) state across multiple Actor interactions. In fact, in certain cases it may be helpful to include an Actor, as a 2nd Order Resource, within the definition of an assembly in order to, for example, express constraints on the usage of a Service or even the Assembly in its entirety. Developing support for having Actor definitions in Assembly Descriptors is future work in PalCom.

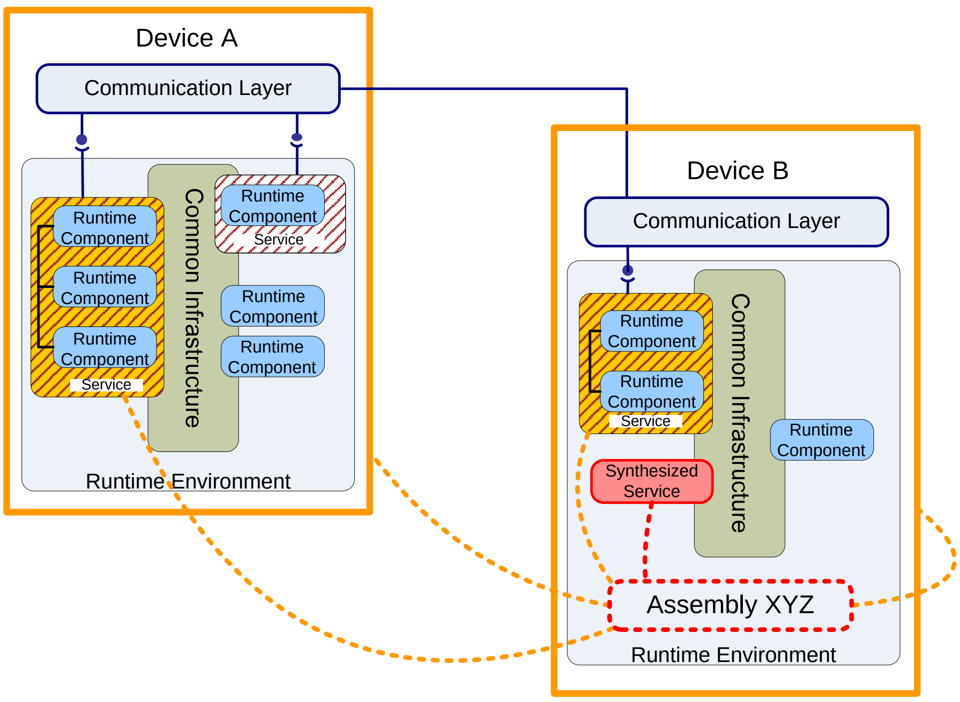


Figure 5.2: Illustrative example of a PalCom Assembly.

An Assembly is illustrated in Figure 5.2, which shows two Devices interacting via remote Communication Channels. Within Device A can be seen the expected elements: The PalCom Runtime Environment, the Middleware Management, the Communication Layer, several Runtime Components and two Services. The Middleware Management intersects all components and services, serving them with resource and contingency management facilities as described in Section 6. The PalCom Runtime Environment resides logically below all Runtime Components and Services, offering the means to manage and execute them. Device B has a slightly different Component/Service population. The differences are the presence of the XYZ Assembly, which is hosted on the Device, and the Synthesized Service which is defined within the Assembly and deployed as a nominal service combining the other member Services of the assembly to create some defined aggregated functionality.

One of the Services in Device A is shown as an executing combination of three Runtime Components. The other Service in this Device is shown as containing only a single component. It is possible for one software component to have multiple instantiations within a Runtime Environment, each of which resides in a different Service.

Shown bridging both Devices is the Assembly which, in the snapshot, consists of the two Devices, a selection of Services and a Synthesized Service. This particular composition may of course change with time, if so allowed by the assembly type and configuration.

An Assembly is deployed according to an Assembly Descriptor that defines the 2nd Order Resources present within the assembly (which will be primarily Services) and how these Services can be composed to offer aggregated behaviour and Synthesized Service interfaces. In the descriptor, different kinds of *bindings* can be specified, which govern how services can be replaced with others if they disappear, and whether a given service is optional for the execution of the Assembly, or not. The Assembly Descriptor can also contain a script defining how to execute the assembly. Appendix A of Deliverable 49 [68] contains an abstract grammar which describes the structure of PalCom Assembly Descriptors.

So, essentially, Assembly Descriptors contain a means of composition and a means of coordination. Composition is the ability to specify and scope a set of PalCom 2nd Order Resources that contribute toward the behaviour of a PalCom Assembly. Coordination describes the interaction of those resources within the context of a PalCom Assembly.

In the PalCom infrastructure, there is a framework for building Assemblies. The most common use of these framework classes is to execute Assembly Descriptors that have been read from an XML file or over the network. At the same time, the classes serve as an object-oriented framework. It is possible to extend the class *Assembly* at the Java level, for implementing an assembly directly in Java.

When the various elements of the assembly execute, a sequence of activites are performed which will include the deployment of the Synthesized Service, its execution and other actions relating to binding the assembly constituents together. As illustrated in Figure 5.2, the Synthesized Service combines the two Services. The result returned by the Synthesized Service will be the result of this two-service composition.

Assemblies can of course be interconnected to create a composite PalCom system. Assemblies can also be recursively embedded within one another, in a hierarchy, as illustrated in Figure 5.3. This hierarchy is established by including Synthesized Services of lower-level Assemblies as Services in higher-level Assemblies.

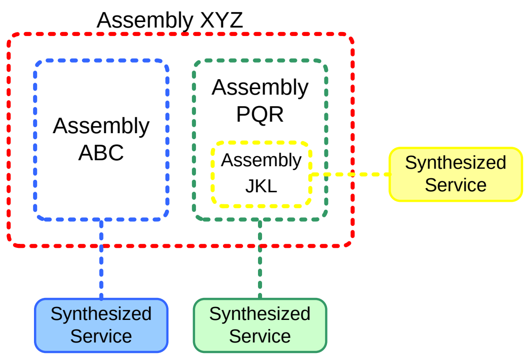


Figure 5.3: An embedded hierarchy of PalCom Assemblies.

An instantiated PalCom Assembly can either be static or dynamic in form. Static assemblies will not vary in structure throughout their instantiated lifetime. Dynamic assemblies, however, can be dynamically altered at run-time to add or remove 2nd Order resources if required by changes in operational requirements, constraints, context or contingency actions.

Assemblies are created and managed throughout their lifecycle by Assembly Managers, detailed in Section 6.2. The Assembly Manager is in particular responsible for coordinating the various elements of an assembly. In addition Contingency Managers, detailed in Section 6.3, are responsible for attempting to ensure that executing assemblies remain running by managing failures and other problems, and if necessary and possible ensuring graceful degradation of an assembly's operation.

At runtime, an Assembly runs in an Assembly Manager on one Device. There is nothing in the Assembly Descriptor that ties it to that particular Device, however, so it is possible to move the Assembly Descriptor to another Device and start the same assembly there. In Figure 5.2, e.g., the Assembly could equally well execute on Device A. 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. Migration of Assembly Descriptors is also used by the Assembly Managers' automatic update mechanism (see Section 6.2)

Assemblies may be transparent to the PalCom system as a whole in the sense that PalCom 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, for example, for *unbound services* (services that are not tied to a particular device, see [68]). It is future work in PalCom to put mechanisms in place which let the Assembly delimit access and discovery of the participating Services. In this way, Assemblies can provide scoping and encapsulation, supporting a scalable programming model.

Further detailed elaboration of Assemblies, their creation, management and use, may be found in *Deliverable 49* [68] and in the Developer's Companion [71]. These sources also provide more concrete examples.

5.3 Device

When executing, services and assemblies are deployed onto Devices (see Section 3.2.9) which can be programmed using the Device framework. It eases the setup necessary for giving a service access to other services on the device with which it may cooperate, and to device-common middleware managers. This is not a framework for creating the devices per se, but for writing the code that starts and schedules services and managers. The device framework provides points for initialization and termination of services and managers on a device.

There is a class *AbstractDevice*, which is subclassed for implementing a concrete PalCom device:

```
class MyDevice extends AbstractDevice {
  protected void initDevice() {
    MyService service = new MyService(context);
    context.addService(service);
    service.start();
  }
}
```

References to managers and services on the device are provided in a *device context* object. In the `initDevice` method above, a service is created, added to the device context, and started. The `initDevice` method is called by the framework for initializing a device.

For communication with the hardware on a device there is an interface *DeviceIO*, which supports event-based communication with lower-level processes such as interrupt routines. As a support for the developer, the PalCom toolbox also contains a small framework for *simulated devices*. Those are Java programs that run on a desktop machine, with graphical representations of the hardware of a device. Code written using the device framework can be run 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.

Chapter 6

Middleware Management

The *Middleware Management* of the PalCom Infrastructure resides on the PalCom Runtime Environment, as shown in Figure 4.1, and provides a set of middleware functions for managing the lifecycle of PalCom Runtime Components, Services and Assemblies.

The specific management tasks identified as the delineated functionality of the PalCom Middleware Management are *Assembly Management*, *Resource Management*, and *Contingency Management*. The concrete managers responsible for these three tasks are shown in Figure 6.1 in terms of selected relationships to key concepts in the PalCom Architectural Ontology.

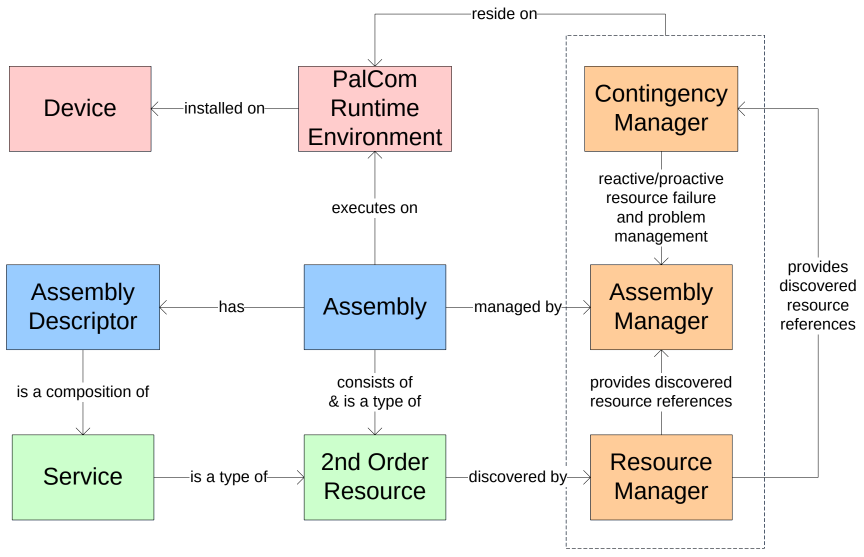


Figure 6.1: Middleware Management and its relationship to the concepts in the PalCom Architectural Ontology.

The Middleware Managers are shown on the right of the diagram, executing on the PalCom Runtime Environment and managing PalCom Assemblies and other 2nd Order Resources. Several instances of these managers may exist in accordance with system scalability and redundancy requirements.

- *Assembly Management* is handled by the Assembly Manager entity which is responsible for the creation and lifecycle management of PalCom Assemblies.

- *Resource Management* is handled by the Resource Manager entity which maintains an up-to-date directory of all active (and if required, inactive) 2nd Order Resources.
- *Contingency Management* is handled by the Contingency Manager entity which maintains resource and assembly resilience by monitoring failure and problem conditions and applying both reactive and proactive compensation policies and mechanisms.

A sketch of the way the managers interact is shown in Figure 6.2. The resource manager deals with low-level source events (LLSE) (e.g. generated by hardware entities) whereas the contingency manager deals with higher-level source events (HLSE), usually generated by software entities as e.g. services. The Resource Manager receives low-level source events containing information about the availability of resources used by an application (this is related to the availability of 1st and 2nd Order Resources). Based on this, it keeps an up-to-date view of the 2nd Order Resources currently available. The Assembly Manager interacts with the Resource Manager for performing the coordination tasks of its Assemblies. The Contingency Manager receives higher-level source events from services as a result of the services respective monitoring functions. This triggers the handling of contingency situations.

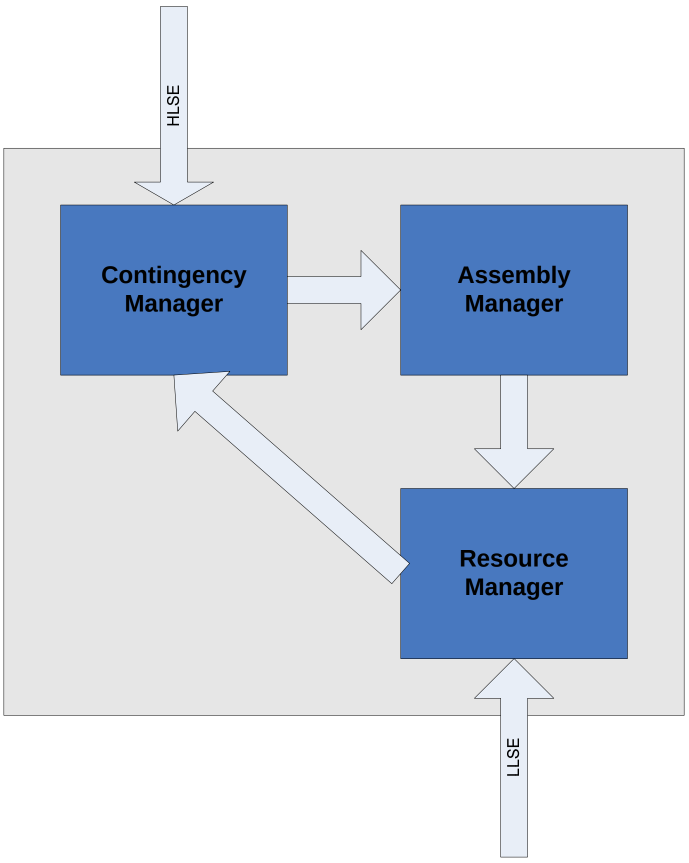


Figure 6.2: Schema of Middleware Manager Interactions triggered by Resource availability (LLSE) and Monitoring (HLSE) events.

Each of these managers is elaborated further in the following sub-sections, with more detailed models provided in previous PalCom Deliverables 42 and 43 [65, 66].

6.1 Resource Management

A Resource Manager is a constituent of the PalCom Middleware Management and is responsible for maintaining an up-to-date directory of all available 2nd Order Resources within its operational scope. An available 2nd Order Resource may be either active and thereby currently available for use (subject to any additional usage constraints), or inactive and thereby not currently available for use.

Every PalCom Assembly will nominally have one active Resource Manager residing on each PalCom Device participating in the assembly. Other inactive Resource Managers may also be present within the assembly, especially to support distributed replication and failover redundancy. Such an inactive Resource Manager would then be activated on failure of a previously active Resource Manager.

In accordance with the definitions available in the PalCom Architectural Ontology, Figure 6.3 shows the relationship between 1st and 2nd Order Resources. As shown, the Resource Manager is only responsible for discovering 2nd Order Resources (see Section 3.2.12).

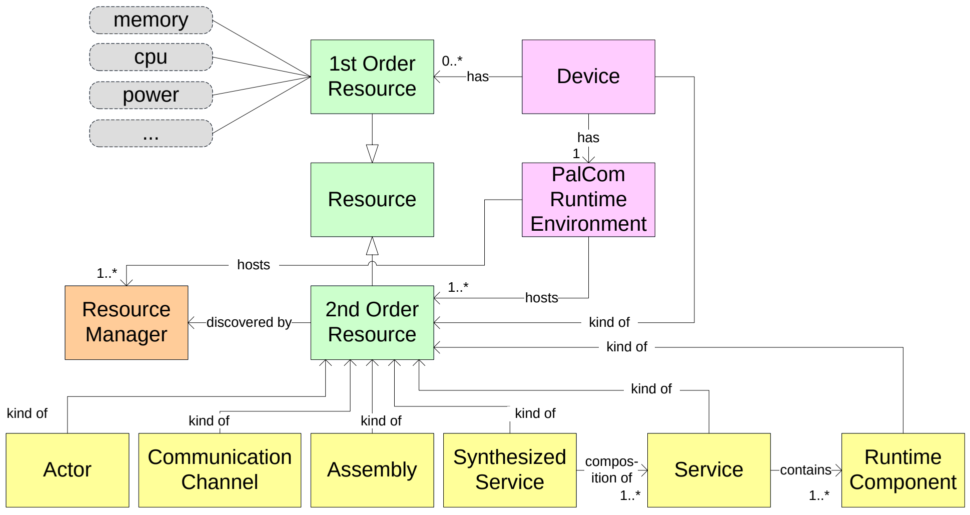


Figure 6.3: Focus on the PalCom Resources and their relationships to other concepts in the PalCom Architectural Ontology

The PalCom architecture bases resource awareness on the reflective properties of the PalCom entities. Each 2nd Order Resource in a PalCom system is able to offer a resource description providing essential data on how to deal with it, as discussed in Section 7.2

The primary operations of a PalCom Resource Manager are as follows:

- Discover 2nd Order Resources
- Persistently monitor previously discovered 2nd Order Resources.
- Match resources according to needs expressed by Assembly and Contingency Managers.
- Describe available 2nd Order Resources to Assembly and Contingency Managers.
- Provide available 2nd Order Resources to Assembly and Contingency Managers.
- Describing its own behaviour when inspected.

Some optional operations that may be performed by a PalCom Resource Manager include:

- Reserving resources for use at some point in the future for a specified duration. Quality of service guarantees can be assigned to provide a measure of certainty with respect to the availability of a resource.
- Negotiate with other Resource Managers for access to a particular 2nd Order Resource.
- Predicting resource requirements based on analysis of previous resource requests received from Assembly Managers. This is coupled with resource reservation to automatically reserve resources if a request for use is anticipated.
- Manage the lifecycle of selected resources including initiating, terminating, restarting and moving, etc. 2nd Order Resources.
- Arbitrate conflicts over resource access and usage between Assembly Managers.

Not all Resource Managers need be implemented in same way, or provide exactly the same functionality.

The functionality of PalCom Resource Managers is elaborated further in *Deliverable 42* [65].

6.2 Assembly Management

An Assembly Manager is a constituent of the PalCom Middleware Management and is responsible for constructing, deploying and managing the lifecycle of Assemblies.

Every PalCom Assembly will have one active Assembly Manager, typically residing on a PalCom Device participating in the assembly. Other inactive Assembly Managers may also be present within the assembly, especially to support distributed replication and failover redundancy. Such an inactive Assembly Manager would then be activated on failure of a previously active Assembly Manager. Although it is conceivable that multiple Assembly Managers may be active within the same assembly, this requires special coordination. Support for that has not yet been implemented in the PalCom toolbox. Interaction between Assembly Managers governing different PalCom Assemblies is possible however.

There is a protocol, defined at the PalCom Service interaction level (see Section 7.1.4), which Assembly Managers use for communicating information about assemblies, and updating Assembly Descriptors automatically. This automatic updating mechanism is based on versioning of assembly Descriptors, as elaborated in *Deliverable 49* [68].

As mentioned, the first duty of an Assembly Manager is the construction of PalCom Assemblies according to an XML-encoded Assembly Descriptor (see Appendix A of Deliverable 49 [68]). The Assembly Descriptor contains details of the particular 2nd Order Resources to be used within the assembly and scripting of the contained Synthesized Service(s) which will make use of those resources to deliver composed functionality. Both templates and instances of 2nd Order Resources, and specifically Services, can be specified in the Assembly Descriptor.

Templates abstract the choice of, for example, a Service, to the type of that Service rather than a specific instance. This allows loose assembly binding such that if a template is used, then the selection of a service instance can be delegated to the Resource Manager by sending a request for a service instance conforming to the template constraints. This can include, for example, the particular 1st Order Resources required on a Device that might be hosting a Service meeting the required template criteria.

Moreover, in forming Assemblies the resource requirements of the parts of an assembly are gathered and assessed in order to take informed decisions about the parts of an assembly considering resource constraints and preferences. The establishment of the assembly itself is also part of the resource management system, which does this based on an Assembly Descriptor, which contains the resource types required for the assembly as well as the work-flow descriptor.

Subsequent to construction an Assembly Manager is then responsible for deploying the PalCom Assembly and thereafter managing its lifecycle in collaboration with one or more available Contingency Managers (see Section 6.3). The primary features of PalCom Assembly lifecycle management as performed by an Assembly Manager are as follows:

- Evolution of the assembly through the dynamic removal of existing resources, or inclusion of new resources.
- Monitoring of assembly behaviour.
- Recognition of problems, faults or failures in the assembly.
- Issuing of events to other middleware managers including, for example, those to a Contingency Manager relating to problems, faults or failures.
- Maintenance of the assembly in accordance with contingency actions and/or reccommendations.
- Describing its own behaviour when inspected.

Not all Assembly Managers need to be implemented in same way, or provide exactly the same functionality.

In the PalCom toolbox, the functionality of the Assembly Manager is distributed over the class *AssemblyManager* and the class *Assembly*. The latter acts as a baseclass for assemblies (see Section 5.2).

6.3 Contingency Management

A Contingency Manager is an optional constituent of the middleware management responsible for attempting to ensure the survivability of Assemblies and other 2nd Order Resources by administering problem and fault conditions through the application of contingency actions.

Every Assembly will typically have an active Contingency Manager residing on one Device participating in the Assembly. Other inactive Contingency Managers may also be present within the Assembly, especially to support distributed replication and failover redundancy. Such an inactive Contingency Manager would then be activated on failure of a previously active Contingency Manager.

No behaviour is mandated for the Contingency Manager, which is considered to be an optional element of any PalCom System. Several recommendations for events and behaviours are detailed in *Deliverable 42* [65], with a short overview of each outlined in the following sebsections.

6.3.1 Events

Contingency Manager operations are triggered by incoming events sourced from Assembly and Resource Managers. A nominal set of events may include event handlers such as:

- *ContingencyEvent*
  A ContingencyEvent is a general purpose event type sourced from a Resource Manager or an Assembly Manager. It is created whenever some aspect of an assembly's operation fails.

- *ProblemEvent*
  A ProblemEvent is an early detection indicator sourced from either a Resource or an Assembly Manager. It describes a situation where Assembly behaviour is such that some intervention is required on the side of the Contingency Manager to prevent a possibly more serious fault or a failure.
- *ResourceExhaustionThreatEvent*
  Sourced from a Resource Manager, this event is sent when a threshold is passed indicating the availability of some 1st Order Resource is in danger of exhaustion potentially impairing or endangering correct operation of an assembly or other 2nd Order Resource.
- *ResourceReservationExpiryThreatEvent*)
  Sourced from a Resource Manager, this event indicates that a resource reservation on some 1st or 2nd Order Resource will expire., or has expired.
- *ResourceReservationExpiryEvent*
  Sourced from a Resource Manager, this event indicates that a resource reservation on some 1st or 2nd Order Resource has expired.
- *ServiceResponseTimeoutEvent*
  Sourced from a Resource Manager or an Assembly Manager, this event indicates that a service did not respond within am set timeout.
- *ServiceResultTypeFaultEvent*
  Sourced from a Resource Manager or an Assembly Manager, this event indicates that the called Service returned a value of the type that does not correspond with the signature.
- *ServiceResultIntervalFaultEvent*
  Sourced from a Resource Manager or an Assembly Manager, this event indicates that the Service returned a value of the correct type, but it does not fulfill other constraints, i.e. is out of bounds.
- *ResourceNotAvailableEvent*)
  Sourced from a Resource Manager or an Assembly Manager, this event indicates that a particular Resource required by an Assembly is no longer available.

6.3.2 Contingency Tolerance

Contingency tolerance is defined by the capability to detect and diagnose faults and failures. A fault arises from situations that result in some aberrant behaviour that while incorrect or unexpected, is typically transient and may not actually cause a breakdown in overall operation. A failure indicates that some event has occurred leading to continued abnormal behaviour that must be compensated for if the system is to regain normal operation. A Contingency Manager should be capable of diagnosing faults and failures with a form of root cause analysis by checking events against known, and historically recorded, fault and failure conditions. Special behaviours are capable of correlating detected events both internally, and with other Contingency Managers. The following are a selection of potential fault and failure detection and diagnosis mechanisms.

- *Monitoring*
  The Contingency Manager collects and statistically analyses system behaviour (and /or behaviour of the operating environment), this information is then stored and compared with expected data. Typical monitoring includes threshold monitoring to determine whether, for example, a certain amount of memory has been consumed which may trigger a contingent action to introduce a second device across which the memory load can be balanced. Other monitoring include resource exhaustion, service reliability, etc.
- *Heartbeat*
  A Contingency Manager periodically pings the constituent services of an Assembly to confirm their network availability.

- *Periodic Invocation*
  A Contingency Manager periodically invokes services and tests their interface specification as a quality measure.
- *Service Hot-Swapping*
  In case of service unavailability, another service with an identical signature is automatically made available. A Contingency Manager should offers the means to also preserve service state if required.

6.3.3 Contingency Containment

Contingency containmenet are operations designed to prevent a fault from causing widespread problems.

- *Deactivate a Service*
  Deactivate a service that is detected by a Contingency Manager to be malfunctioning in some way.

6.3.4 Contingency Repair

The primary contingency repair mechanisms include replacement, reconfiguration and graceful degradation. Replacement consists of replacing a faulty or failed resources with an alternative. This may either be effected by means of fail-over to a hot-swappable redundant resource, the initiation of new resources that can take over from the failed one(s), or to restart the faulty or failed resource. In all cases dependencies must be maintained, and if not possible then either the entire assembly must be restarted, the affected section be isolated if possible from the remainder of the assembly. In all cases the user can choose to be informed, or not, of all events and actions taken. If replacement is neither the possible nor preferred option, an alternative is dynamic reconfiguration of the assembly. If the assembly can support it, this implies that a faulty or failed service can be moved, or restarted on an alternative device. This is of course dependent on whether the service is required to be local to the original host device type. If a system cannot be repaired or reconfigured to operate as expected the secondary option is to attempt to ensure that it degrades in a controlled manner causing the minimum of disruption to expected behaviour. This can be a complex problem that requires that the Contingency Manager have some specific knowledge of not only what functions an assembly performs, but the criticality of each function and how whether its operation can be isolated from faults or failures to continue assembly operation in a degraded form. One useful technique is to use a form of load-balancing to dissipate operational load between alternative resources, if available.

- *Surrogate Service Replacement*
  In the event of a Service failure the Contingency Manager can seek surrogate Services providing the same (or sufficiently similar) functionality.
- *Service Migration*
  If a Service cannot be replaced, Contingency Manager can attempt to migrate it to another suitable device.
- *Service Restart*
  Restart a Service.
- *Assembly Restart*
  Initiate an Assembly restart.

6.3.5 Proactive Contingency Actions

More advanced Contingency Managers may also provide *proactive contingency actions* which attempt to plan strategies for dealing with errors, faults and failures *before* they occur. Strategies can be proposed to actors allowing them the opportunity to override them if preferred. Some of the optional operations include:

- Establishing proactive precautionary error avoidance strategies which will attempt to mitigate potential error conditions: e.g., failure of a service, a device, network failures etc. by anticipating them and putting contingent actions in place as precautionary measures. These actions may either simply raise exception warnings, or may attempt to divert from an anticipated error condition into a safe state of operation.
- Immunising against known problem conditions by recording events and learning from past experiences. Using this knowledge structural (e.g., assembly formation) or behavioural (e.g., task formulation) adaptation can be applied to eliminate or at least reduce the chances of an error condition occuring. By these means a system optimises its own operation.
- Preparation for the graceful degradation of the integrity of active PalCom Assemblies by containment of detected problem or failure conditions. This is achieved by attempting to isolate potential sources of errors, faults or failures and determine means of allowing systems to continue running in states of limited functionality while safely degrading if necessary.
- Restitution of system integrity through self-healing adaptation and external intervention (if necessary) to recover lost functionality and state.

Chapter 7

Runtime Environment

The PalCom Runtime Environment acts as both a platform for realising and deploying PalCom Components and provides an execution environment for PalCom Runtime Components, Services and Assemblies. It is therefore necessary that at least one instance of the PalCom Runtime Environment is physically located and available on each PalCom Device. In addition, the PalCom Runtime Environment is the only element of the PalCom architecture that explicitly relies on the existence of a network and the capability of PalCom Devices to join and leave this network when needed.

The description of the PalCom Runtime Environment is divided into a number of interrelated aspects as sketched in Figure 4.1.

The *Runtime Engine* handles the hosting of PalCom Components as well as the execution of processes, threads, PalCom Runtime Components, Services and Assemblies and provides primitives for the other functionality used by the other parts of the runtime environment.

The *Process & Thread* model specifies how to create parallel or alternating sequencing of actions within the runtime environment, and provides mechanisms for various levels of isolation between such units of functionality.

The *Introspection* mechanisms allows for inspection of running assemblies, services, runtime components, and communication channels at suitable levels.

The *Communication* model defines mechanisms that PalCom Services can use to access and interact with other PalCom Services residing (possibly within PalCom Assemblies) on other PalCom Devices.

The PalCom Runtime Engine may be realized in the form of a virtual machine. Whether or not this is chosen is optional according to the specification. In the reference implementation of the PalCom architecture, the PalCom Runtime Engine is implemented both as a separate new virtual machine `pal-vm`, and with the use of the standard Java Virtual Machine.

7.1 Communication

The PalCom communication architecture is designed as a layered architecture as outlined in Figure 7.1. The purpose of this part of the architecture is to provide PalCom with connectivity that matches the requirements as uncovered by the PalCom scenarios [69].

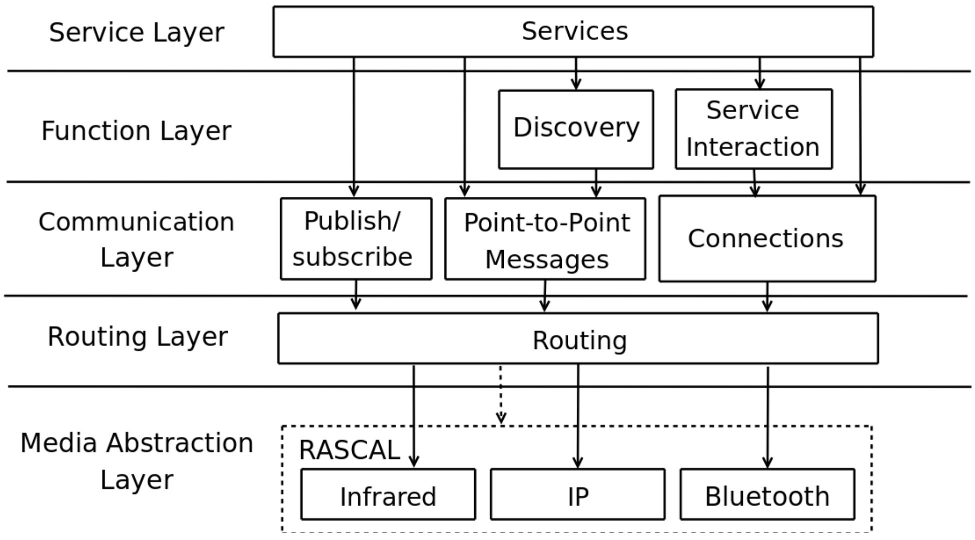


Figure 7.1: The PalCom Communication Model.

Media Abstraction Layer At the lowest level of the communication architecture, the Media Abstraction Layer (MAL) has the purpose to bridge between the PalCom communication model and the actual network used. This layer thus effectively hides all the implementation details of the used network from the upper layers. This is essential in order to make it easy to add support for new network technologies in the future, and also to make it transparent to dynamically switch between available network technologies. Currently there are three implementations of this layer, for IP, Bluetooth and IR communication technologies, respectively.

Routing Layer For PalCom devices that support more than one media interface (of different or of the same kind) there are some interesting questions which are handled in the Routing Layer. If the receiver can be reached more than one way, the choice of which media to use for communication is made here. Currently there are two implementations of this functionality. One very simple implementation that is suitable for devices that only have one media interface, (and thus routing is a non-issue). It also handles the situation with more than one media interface using a very simple, first-fit algorithm.

A second problem addressed in this layer is forwarding, where communication can be directed through a Device as indicated in Figure 7.2 on page 69. The layer also supports the implementation of Tunnels, that is VPN-like communication over hostile media between two or more local PalCom networks.

Communication Layer In the Communication Layer the common part of all PalCom communication is implemented, as a realization of the Communication Channel concept in the ontology (see Section 3.2.7).The layer supports both message based communication as well as streams. Distribution covers communication between two endpoints, i.e. 1-1, as well as 1-n (broadcast), and n-n (publish/subscribe). There are facilities for establishing stable connections and single-shot messages. It also supports reliable communication, and handling of large messages.

Function Layer The Function Layer, on top of the Communication Layer, supports two specific protocols. The Discovery Protocol is internal to the PalCom architecture and supports the discovery of Devices, Services, Assemblies, and established Connections in PalCom. For the higher levels of the PalCom architecture, these mechanisms are accessible through an API hiding these details. The second protocol in the Function layer is the Service Interaction Protocol used by services to exchange messages. The format of this protocol is defined, but the contents of the actual messages sent by a service naturally depends on the service.

Service Layer The Service Layer represents 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.

RASCAL In the MAL there can optionally be a RASCAL component, which is one of the outcomes of the PalCom work on *Transient Locations* (see [69]). The network-aware aspect of RASCAL allows the expression of policies to be enacted by the RASCAL control agent in accordance with network status when preparing to send or receive messages. Policies can be based on the bearer network technologies available or on some QoS parameters such as the load of the nodes/devices involved in the communication or with contingency situations such as failovers. A network-aware policy will consider using an alternative interface for sending messages to a particular node if a currently employed interface is unavailable. The usage-aware aspect of RASCAL allows the expression of policies to be enacted by the RASCAL control agent in accordance with deployed user-level services. Examples of these are: contingency management decisions, content adaptation decisions, deferred service provisioning decisions, role management decisions, etc. An example of contingency policy particularly useful in disruptive environments consists of sending the message using two or more different routes simultaneously to maximize the potential for a message to reach a target node. All policies allow RASCAL to dynamically adapt communication behaviour according to particular scenarios in order to exhibit stability to the PalCom user.

7.1.1 Protocols

Although there is a reference implementation of the PalCom Architecture, there is also a set of defined protocols for the communication. This is important in order to enable independent implementations of the PalCom Architecture in for example other languages. The three PalCom communication protocols are outlined below and described in more detail in the next section.

Wire Protocol The Wire Protocol defines the format of messages in the communication. It has been designed as a compromise between compactness, flexibility, and readability to be based on a terse representation in readable characters. The heading part of communication is structured as a set of nodes where new ones can be introduced with little or no changes to existing such nodes. This supports a high level of flexibility for new demands to the price of some parsing of the heading information. The Wire Protocol is handled by the lower levels of the Communication architecture: Media, Routing and Communication layers.

Discovery Protocol The Discovery Protocol transports information about PalCom Devices among themselves. At the heart of the discovery mechanism broadcasting is used, but as soon as the devices are in contact, information is transported unicast, on demand. The Discovery Protocol handles information about a PalCom Device, what Assemblies exist on it, what Services it offers, description of the Services and what messages it can send/understand, and established connections to other Services. The protocol is designed with embedded status information. This makes it safe to cache information, trusting that originator will flag when the information is no longer valid.

Service Interaction Protocol The Service Interaction Protocol is used after services have been discovered using the Discovery Protocol. It is used for establishing connections between PalCom services, using the Remote-Connect sub-protocol, and for communication between services, by means of commands that can be sent and received.

7.1.2 Wire Protocol

The Wire protocol is used to transfer data over the underlying network(s). The protocol is used primarily by the Communication Layer and Routing Layer, but also affects the Media Abstraction Layer (which implements aspects that are specific for each ”medium”). Compared to the initial prototypical definition and implementation of the communication format in PalCom, a number of features have been added in this version:

- Support for transfer of *long messages* over technologies that offer limited message lengths.
- Support for *reliable communication*.
- Consistent use of *unique names* for devices and services.
- Enabled implementations of *routing*. In particular, addresses in terms of DeviceIDs and URLs need to be placed in proper places (see below).
- Different *versions* of the formats is supported, so that possible future updates can be handled.
- Support for protection by means of a *user/password* facility.
- *Efficiency* with respect to compactness of messages and reducing redundant information. This is mainly addressed by the special support for heartbeats at this wire-level and the introduction of the cache-info.

The changes not only require changes to the wire protocol as specified in this section, but also a revision of the XML messages used for Discovery (including removal of URLs from there), see Section 7.1.3.

The line format of messages is structured as a sequence of nodes. These nodes are handled by the lower layers, which bridge from the underlying network and up to the Palcom Function and Service Layer. See Figure 7.1. The data (payload) of the messages is handled by these upper layers. The following sections describe the parts of the Wire format relevant for the different layers.

7.1.2.1 Media Abstraction Layer

The Media Abstraction Layer bridges between the underlying network at the lowest level, based on IP, BT, IR or other technologies, and the Palcom Communication model. At this level we provide the information needed by these APIs for sending information, such as the actual URL, e.g. IP-address and port for IP and similar for other technologies (BT, IR etc.). We assume that when we receive information from the network this layer can figure out what URL is needed to contact the sender again. Technically this layer translates between URL addresses used by the Medium and the medium independent addresses used by PalCom.

DeviceID - a unique identifier of a Device. The DeviceID identifies the device, not an individual network interface.

For incoming messages, the MAL uses the URL of the sending device and translates it to the DeviceID. The DeviceID is retrieved through a special protocol after the first message from that device has been received. For outgoing messages, the MAL translates the DeviceID to the matching URL. URLs are not visible above the MAL layer.

The model is that a device sits on one or more networks, with one MAL layer manager for each network, as illustrated in Figure 7.2. URLs are assumed to be valid only within one such network, while PalCom DeviceIDs and Selectors span across networks.

7.1.2.2 Routing Layer

The Routing Layer provides the mechanisms needed to choose among different available media and interfaces. In case there is only one communication interface available, this layer is trivial.

However, for devices that have more than one communication interfaces, there is a need of routing outgoing messages through one of them. There might also be a need for forwarding incoming messages on one interface/network medium to another interface. Such facilities are implemented in this layer.

This layer also handles messages that are too large to send in one go on a particular interface/network. These messages are chopped up in smaller parts on the sending side and assembled on the receiving side. When the message is complete it is forwarded to the listener, or it is all dropped.

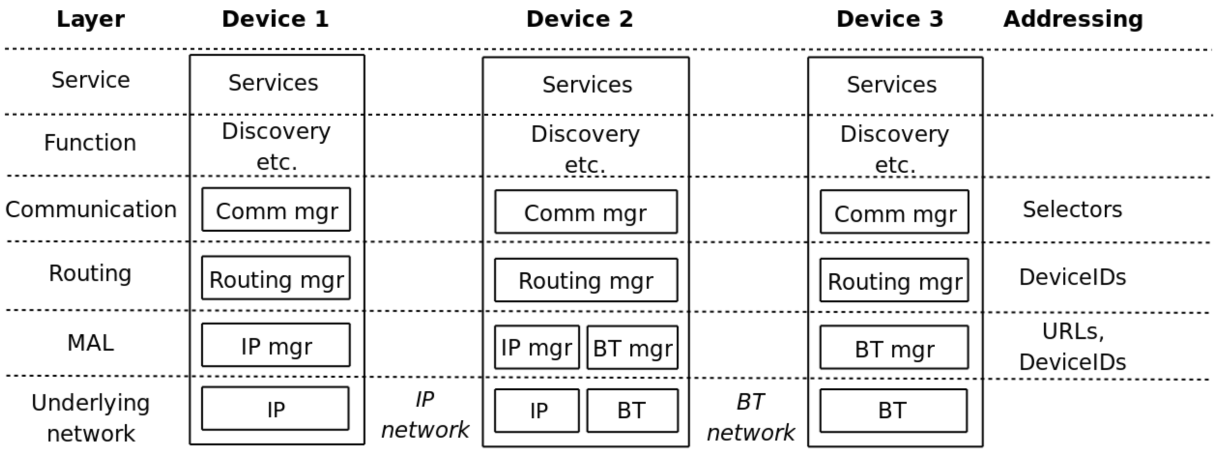


Figure 7.2: A schematic view of three PalCom devices, communicating in IP and Bluetooth networks. The right-most column shows what addressing mechanisms are used in the respective layers: above the MAL layer, no medium-specific addresses are used. Device 2 is in both networks. The other devices are only in one network each, but can communicate by addressing each other using DeviceIDs and Selectors. The communication is routed through Device 2.

7.1.2.3 Communication Layer

The Communication Layer handles the mapping of the Palcom communication model onto the Wire protocol. The Palcom Addressing mechanism is built on the use of DeviceIDs and Selectors. Selector - a local identifier, matching a connection to a communication end-point (typically a service) on a particular device.

This layer includes the following important functions:

- *Internal delivery.* This is the problem of delivering an incoming message to the appropriate ”Listener” on a device. In most cases, the listener is a service, but it can also be, e.g., the discovery manager on the device. This layer handles the communication over Connections, that is addressing local to this device, based on Selectors. It will thus use the receiving-selector (selR) of an incoming message to find the corresponding listener. For outgoing messages on a Connection this layer will provide the selector of ”the other end”. This layer also handles establishment of Connections. For so called ”single-shot” messages the selector of the sender (selS - ”other end”) is saved/forwarded to the listener on this device to guide a possible reply from the listener back to the sender.
- *Reliable connection.* Reliable communication over an unreliable connection is handled in this layer. Here, messages over a reliable connection are acknowledged by sending a reply to the sender, or are re-transmitted as needed.

7.1.2.4 Addressing

The general addressing mechanisms of PalCom are made up of two parts:

- *DeviceID:* a unique identification of a Palcom device that remains stable. It represents the device itself, rather than a particular network interface it might have. Methods for assigning DeviceIDs are described in the Discovery Protocol (see Section 7.1.3).

- *ServiceID:* an aggregated, unique identifier for a Service. Part of the ServiceID stays stable when a Service is moved or cloned to another Palcom device. It also includes a part that makes it possible to have more than one service of the same sort on the same device as well as versioning information.

DeviceID and ServiceID are used by PalCom applications to represent connections between devices making it possible to re-connect to exactly the same device at a later time, or to make sure a connection is to a similar service on another device. These identifiers are, e.g., stored as part of Assemblies and used for re-connecting when the Assembly is started.

The DeviceIDs and ServiceIDs are fairly bulky and sent over the network as seldom as possible. Instead, the DeviceIDs are translated to the primitive addressing mechanism of the underlying network as often as possible, while still maintaining the guarantee of binding to the particular device. Similarly the ServiceIDs are represented as a much shorter identity number when devices are actually communicating.

Taking an IP network as example we have the following addressing elements that come up in Palcom and this document:

- *IP address:* a number that at every instance in time and network is identifying a computer, or a Palcom Device. It can, however vary so the same Device will have different IP-numbers from time to time, for example using DHCP. And correspondingly, a given IP-address can lead to another Device at another point in time. During a period when one has contact with a Device, the IP-number is, however, stable. The MAL is keeping a translation table between the IP-address and the DeviceID of a PalCom Device. This association is built up when the Device is discovered which means that the actual DeviceID needs to be sent over the network initially, but only once.
- *Port:* a number used by IP protocols to establish connections to sockets and particular programs executing on a computer. Often ports are associated with particular protocols and many firewalls are blocking communication over specific ports for security reasons. Palcom is trying to use as few Port numbers as possible in order not to demand particular ports available, ideally only one.
- *Selector:* an identifier used by the Palcom communication layer to address a certain service on a Device (or another communication end-point, such as a manager). Selectors are short, only used for selecting a service on a particular Device. The mapping is sent as part of the Discovery protocol (see Section 7.1.3).

The PalCom communication protocols and addressing schemes abstract over 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 DeviceIDs and selectors, PalCom devices can communicate across network technologies in several hops, relying on routing functionality in the Routing Layer. For limited environments, the mapping to a concrete technology can be tailored specifically for resource constraints.

In the PalCom world there is stable addressing in terms of DeviceID/ServiceID. Such an address is unique and is used to establish connection with the very same Service, in a particular version on a given physical device (even if it has changed IP number meanwhile). DeviceIDs are translated to IP addresses (and Port number) when a Device is discovered, and when a connection is established the ServiceID is mapped to a Selector. This way, uniqueness is combined with very efficient representation when communication is taking place.

The above overview is focusing on IP as an example. For other media we assume that there are low-level mechanisms similar to IP-numbers available. Otherwise, this can be implemented in the Media Abstraction Layer.

7.1.2.5 Messages formats

This specification of the Palcom Wire protocol can be read in two ways. It can be read to specify the abstract syntax of the protocol, i.e. the information content of the messages, and it can be read to also specify the concrete representation of the messages. For the concrete representation we are specifying a human readable format. This is the default representation that PalCom devices are supposed to support. There might, however, be situations where it is beneficial to use a more compact (e.g. binary) representation of the same information. Such a representation remains to be specified.

The default representation of the header-nodes of this wire-protocol is ASCII. The representation of the Payload is not defined here, but up to each service to define.

In Appendix C, there is a cross reference for the message nodes in the wire protocol. Appendix D contains examples of messages.

Basic message format structure The fundamental layout of all information in messages follows the pattern

```
Message-node == <F>;<L>;<DATA>
```

where

- `<F>` is a one-byte format identifier. Often a single letter.
- `<L>` is the length of `DATA`.
- `<DATA>` is the structure of the rest of a message, whcih depends on the `F`.
- `';'` is a semicolon that separates the three parts of a message.

Note: Calculating the `L` involves taking the length of `DATA`, 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).

A message is composed of a heading with a number of message nodes followed by the actual data in the last message node. All nodes include a format identifier. The order of the message nodes is designed to support the layered structure by placing the information relevant for the lower layers in the beginning of the compound message.

```
Message == Heading DataNode?
```

Message Data nodes The Data node is the ”payload”, thus the information the sender wants sent to the receiver. There are possibilities to combine several messages sent by the same sender to the same receiver (using the same selectors in both ends).

```
DataNode ::= Multipart | SingleMsg
Multipart == '+';<l>;<data>
<data> ::= M*
```

- `<l>` is the length of `<data>`, including the length of the header.
- `<data>` is a sequence of bytes, containing zero or more messages, concatenated directly after each other.

```
SingleMsg  == 'd';<l>;<data>
```

- `<l>` is the length of the `<data>`.
- `<data>` is `l` bytes of application data.

Examples:

- `d;5;Hello` = Hello
- `+;18;d;5;Hellod;5;Again` = Hello&Again (18 is the length of the underlined part)
- `+;40;+;27;d;5;Hellod;5,Hellod;5;Againd;4;Last` = ((Hello&Hello&Again)&Last). 27 is the length of the underlined part. Note that there are three sub-parts of the first part of this example.

Message Header nodes `Heading == [HeartBeat] [HeartBeatAck] [HBInfoRequest] [HBInfoReply] [HeartAttack] [FormatVersion] [RoutingR] [ RoutingS] [Mark*] [Connection] [Reliable] [AckMessage] [ResendMessage] [Chopped] [SingleShot] [PubSub]`

All header nodes are declared as optional (in brackets), because none of them occur in all messages. When there is more than one header node in a message, the order is given by the order in the `Heading` definition. `Mark` can be present in zero or more copies.

HeartBeat The HeartBeat node is used as a single node, very short, message for the heartbeat part of the Discovery protocol. The direct representation here in the wire protocol is motivated by the need for very compact communication in some situations. This section presents the format of the different heartbeat nodes.

```
HeartBeat == 'h';l;<CachedInfo>;<StatusInfo>
HeartBeatAck == 'H';l;<CachedInfo>;<StatusInfo>
```

HeartBeat nodes are broadcasted periodically by Palcom Devices. When a Device receives such a message it responds by broadcasting a corresponding HeartBeatAck message reflecting its own presence. In this way the most eager Device on the local net is determining the rate of the heartbeats.

If a device is not responding to HeartBeat messages it is understood to be unavailable, at least for the time being.

CachedInfo is a string that reflects the current state of the sending Palcom Node. This string stays the same if the state of the Device has not changed, and is changed if the state of the Device has actually changed. Any cached information about the Device and its service is in that case invalid and must be discarded. The exact meaning of this information and its interplay with the Discovery protocol is described in the Discovery Protocol (Section 7.1.3).

StatusInfo is a string that reflects the operational status of the Device. The format and meaning of this string is defined in the Discovery Protocol.

HBInfo `HBInfoRequest == 'i';l;`
```
HBInfoReply  == 'I';l;<DeviceID>;<DiscoverySelector>
```

When a device receives a HeartBeat from a Device that it has no previous information about, it sends a HBInfoRequest to that Device which in its turn is assumed to reply with a HBInfoReply, providing its unique DeviceID. This information is used in the MAL to build a table with DeviceIDs and associated primitive addresses on the media level (such as IP-numbers). Note that of these it is only the HeartBeat message that is broadcasted. Also note that HBInfoReply message nodes are sent together with VersionNodes (as of 4.4.2 below).

DiscoverySelector is the Selector used by the remote device for sending and receiving Discovery messages. For the content and format see the Discovery Protocol (see Section 7.1.3).

HeartAttack `HeartAttack == 'X'`

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 crashes.

FormatVersion Format and Version info is only sent together with HBInfoReply messages during discovery.

```
FormatVersion ==
'v';l;<wire-version-identifier>;<discovery-version-identifier>
```

A FormatVersion node is sent together with the HBInfoReply node specifying the version of the communication software supported by this Device (i.e. the device that sent out the initial HeartBeat). On receiving the HBInfoReply, the device can decide if it supports the same version or not. If it can't support the selected version, it will not further communicate with the device, and make sure it will not repeat this process at the next heartbeat.

The current wire-version-identifier is "W2". The value of the discovery-version-identifier is defined in the Discovery Protocol (See section 7.1.3).

Routing support The RoutingR and RoutingS nodes are used by Routers and Gateways to tag messages that they forward on behalf of another device (on another network). They expect the corresponding tagging of messages they should forward to a device on some other network.

```
RoutingR == 'r';l;ShortID
RoutingS == 's';l;ShortID
```

For communication between end-point devices (that are not PalCom Routers or Gateways) on a local PalCom network, there is a one-to-one correspondence between DeviceID and the primitive addressing of the underlying network (such as IP-numbers), because the communication is only one hop. For direct communication between these devices the routing-nodes are not needed.

ShortID - a number assigned by the Router/Gateway to address a remote Device. The ShortID is used as an optimization instead of sending DeviceIDs, because DeviceIDs are globally unique and thus cannot be as compact as ShortIDs.

PalCom Routers and Gateways will act as a proxy for PalCom devices on remote networks. When the Router/Gateway announces remote devices on the local Palcom network they will attach a RoutingS node on these messages. This will be used by the receiving devices, informing them that they must attach a corresponding RoutingR node (with the same ShortID) to messages to such devices when send to the Router/Gateway. It is then up to the Router/Gateway to use the ShortID to determine the DeviceID of the receiving device and then figure out where to send the message. If this in its turn leads to another router, the first router will have to replace the RoutingR node with a new one, with the ShortID it once received from that router, to the forwarded message. The router must also attach a RoutingS node on the message providing a ShortID to the original sender of the message.

Mark A sender can add arbitrary information to a message using this header-node.

```
Mark == 'M';l;<data>
```

Markers can, for example, be used by a routing device, in order to tag a message to detect a message that is traveling in a loop in the network. It is up to the user of a mark to define the structure and contents of the data-part of the mark.

Note: There can be more than one mark in a message as they could be put there by different users or layers for different reasons. For example sending a message with some extra information contained in a mark through a routing device to another network will get an extra mark from the routing device to avoid looping the message.

Connection The Communication layer implements the Connection mechanism, which takes care of the internal routing of incoming messages to the corresponding listener based on Selectors. It also facilitates reliable communication which ensures that messages are delivered to their destination and arrives in order. The Connection is a realization of the Communication Channel concept in the ontology (see Section 3.2.7).

- *Internal delivery of messages based on Selectors.*

  The connection mechanism is implemented through the Connection tags.

  `Connection == 'c';l;<f>;...` – the rest of the format depends on ¡f¿-tag.

  There are several kinds of connection-tags:

  – *Radiocasted:* `<f>='b';selS`
    These are messages sent from one single PalCom device to all other reachable PalCom Devices. It is assumed this can be implemented by some efficient mechanism that avoids transmitting the message more than once on the network. It is up to each Device to determine if it is interested in this message, otherwise the message is ignored. The choice is based on the sending DeviceID and the Selector selS.
    For Radiocasted messages there are no replies.

  – *Open:* `<f>='o';selR;selS`
    Establish a connection, create a new connection. The Device sending this message is called the "Initiator" of the Connection. `selR` is the selector on the Receiving device, used for receiving connection messages to a Service. `selS` is the selector used by the Sender, the Initiator, for receiving messages from the Service.
    The connection layer on the Receiving Device locally allocates a unique new selector (`selN`) used for the new connection and responds to the request by replying to the Initiator a Connection-tag with and *OpenReply.*

  – *OpenReply:* `<f>='p';selS;selN`
    On receiving a Connection-tag with `<f>='p'`, the Initiator makes sure that its connection associated with selS is using `selN` for further communication.

  – *UserID:* `<f>='u';Authentication`
    Provide identification information required by the service attempting to connect to. `Authentication` = string, format of which is defined by the receiving Service.

  – *Close:* `<f>='c';selR`
    The Connection associated with `selR` is removed and the associated Listener is informed that the Connection is "closed".

  – *Re-open:* `<f>='r';selR`
    This is a response to a message (`<f>='m'`). The connection that used to be associated with `SelR` is no longer available (the device probably re-booted). Please connect again to get a valid new `selR`.

  – *Message over connection:* `<f>='m';selN`
    For an incoming message the selector `selN` is used to identify the corresponding Connection (resulting from an earlier `<f>='p'` message) and associated service-level Listener. Correspondingly, an outgoing message, over a Connection, is tagged with a connection `<f>='m'` with the `selR` of the receiving node (as the sender sees it).

- *Reliable Connection.*

  A message sent over a reliable connection is instrumented with this optional header node, which is recognized by the Communication layer on the receiver side.

  `Reliable == 'R';l;<Seq>`

  – `seq` is a sequence number over the reliable channel, taking the values: $1, 2, \ldots 2^{16} - 1, 0, 1, \ldots$

  `AckMessage =='A';l;<Seq>`

  If a message with a matching `<seq>` is received (i.e.. `seq= previous-seq+1`) an acknowledge message is sent back to the sender, over the same connection, indicating that the message has safely arrived.

  `ResendMessage =='B';l;<Seq>`

  In the case seq-numbers do not match, one or more requests for re-sending is sent (with `<seq>` set to `previous-Seq+1, +2 ... , seq`).

This layer make sure messages arrives to the Listener in order, possibly by buffering later messages if messages are dropped.

On the other hand, if the original sender does not receive an Ack or ResendMessage (within some timeframe) it must resend the oldest not-Acked message.

Reliable communication in a connection is initiated by the sender by including a Reliable-node. The first time a receiver detects a Reliable-node (with `seq=1`) it takes this to mean that the sender wants to switch to reliable communication.

Chopping large messages Large messages are sent in pieces in messages that have an (optional) Chopped-tag.

```
Chopped == <'-'>;<l>;<M-id>;<part>;<parts>
```

- Format=`'-'` signifies a chopped message.
- `<M-id>` is an identifier of the chopped message. It is the same for all parts of the same message, but changes value between messages.
- `<part>` is the part number. `part=1` means the first part of a chopped long message
- `<parts>` is the total number of parts. `part=parts` means the last part of chopped message.

When the Routing layer receives such a message the parts are assembled in order. If parts are missing or arrives out-of-order the whole partial message is dropped. `<M-id>` is chosen by the sender so that there is no risk that parts from different messages are mixed up.

Note that a reliable chopped connection is possible, in which case the reliable delivery is handled by the mechanisms in Reliable Connection.

Single-shot messages `Single-shot message  = <f>='S';selR;selS!`

An incoming single-shot message is using the `selR` of a Service and the message is routed to the associated listener. The `selS` is passed along to the listener for use in event of a reply to the sender. When sending a single-shot message the sender is providing the `selR` of the receiver, getting them either from an announced service, or in case of a reply, from the message, which it replies to.

Publish/subscribe Publish/subscribe is a very general form of communication where a number of nodes can both send and receive. All communication will reach all the nodes. There are two uses of this communication form in PalCom, a restricted form for Groups of services and a general form.

General Pub/Sub `PubSubMessage == 'P';l;<Topic>`

- `<Topic>` is a string.

When a node sends information using publish/subscribe, it will reach all nodes in the local network. In the receiving nodes there might be listeners, which have registered interest for a particular `Topic`. The Communication layer filter out the messages with matching topic and forward them to the appropriate listeners, if any. This kind of communication will not reach outside the local network, and is intended for information that should be of general interest for most of the available nodes.

Group communication `GroupJoin == 'g';l;<GroupID>`

```
GroupMessage == 'G';l;<GroupID>
GroupLeave == 'q';l:<GroupID>
```

- `<GroupID>` is a string.

When a node sends information using this group-mechanism it will reach all nodes that have registered interest for taking part in this particular group-discussion. Such nodes might also be on remote networks.

Note: Using publish/subscribe communication can come from many different senders to many receivers, while the connection-based Radiocast supports communication from one single sender to many receivers.

7.1.3 Discovery Protocol

This section describes the PalCom Discovery Protocol. It is used to communicate information about Devices, Services, Connections and Assemblies between PalCom devices. The Discovery protocol is used by PalCom devices to announce themselves and to present capabilities they offer to other Devices, and to prospective users in the end.

The Discovery Protocol is understood by the DiscoveryManager and the ResourceManager in the PalCom Toolbox. The DiscoveryManager and the ResourceManager send, receive, and interpret these messages. Other parts of a PalCom application, such as Services and Browsers, interact with these managers, and need not deal with the Discovery Protocol directly.

The section is of interest if you need a deeper understanding of how the Discovery Protocol is organized and works. This is the case if you cannot use the DiscoveryManager or the ResourceManager for some reason and need to re-implement some of their functionality for example in another programming language. This could be the case if you want to make the Services of a small device, for example already programmed in a low-level language, available in the PalCom world. In this particular situation only a part of the protocol is necessary to implement, namely the part that responds to heartbeats and requests. For a small device, it is not necessary to implement the functionality needed for triggering discovery of other devices and their services.

The Discovery Protocol, described in this document, is implemented on top of the Wire Protocol described (see Section 7.1.2). The Wire Protocol is used both for communicating the Discovery Protocol, as described in this section, and for the communication between services, as described in Section 7.1.4.

The first part of this section is written with focus on which messages are included in the protocol and which information they contain. The exact representation and formatting of the messages and the information is defined in Appendix E.

The Discovery protocol is much about names on devices, services etc. The PalCom strategy here is to use unique names for these things as a bottom line. Every Device has a unique name, every Service has a unique name, and so on. Unique names are used to make sure that connections are made to the same device and the same service, in the same versions from one time to another. As a consequence of being unique, these names get long and non-intuitive. Therefore there are two kinds of nick-names for these, readable-names that are used to present to the user and short-names that are used for efficient communication in restricted situations. Notice that the same service can be provided by different devices and it is sometimes important to identify that they are exactly the same. This problem is solved by using (a part of) the unique service name. On the other hand, on a given device, the available services can be identified much more compact and here the short-names are used. The short-names are thus only valid in a local environment, and when this changes, the provider of the name must signal the change in order to trigger an update of short-names. This means that information communicated in the Discovery Protocol is valid until further notice and the same information does not need to be communicated more than once unless there is a change.

The major part of the Discovery Protocol is organized as a unicast request-reply communication between PalCom devices. If a device needs to know about which services a devices is providing, or the details of a service etc. the Discovery Manager on the device typically sends a request to that device and (hopefully) will get an answer.

The fundamental, broadcasted, discovery part of the Discovery protocol is, however, supported directly in the underlying Wire Protocol, making it possible to explore efficient mechanisms on the actual media used. These so called HeartBeat messages are communicated directly from the Media Abstraction Layer in the implementation of the Wire protocol to the Discovery.

7.1.3.1 Device naming

DeviceID PalCom devices are given unique identities that are used both for addressing of the devices and for unique naming of the device and its services. Device names are used both in the discovery protocol and in the wire protocol. A DeviceID can be formed in several ways:

Assigned names these are names 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.

MAC names These names start with a 'B' followed by the 48-bit binary address of a network interface card of the device.

UUID names These names start with a 'C' followed by a 128-bit number, using the UUID standard [29].

In some situations it can be important to be able to give devices rather short unique identities which are possible using Assigned names ('A' names).

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.

The DeviceID is thus independent of what communication medium the device is using. In the Wire protocol the DeviceID is mapped to an actual address depending on the available communication interfaces.

The DeviceID is represented as printable characters in ASCII. The leading character is a letter, which is followed by the bitpattern represented as hexa-decimal digits (0-F). A 128 bits bit pattern will thus be represented as 1+32= 33 characters

DeviceVer A DeviceVer uniquely defines a version of the "platform" on a device. When the DeviceVer 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 DeviceVer changes if manager implementations change, or when the hardware or underlying software changes.

DeviceVer is represented as a string defined by the implementer of the device. The only requirement is that it changes to a new value when the software changes.

DeviceCache DeviceCache is an indicator that signals if the information about this device, received through the discovery protocol, is still valid. A device change the value of the DeviceCache whenever the information it provides via the Discovery protocol is changed. When such a change has been signaled the receiver can either cancel all information about the device and fetch fresh information as needed, or it can use the information in the DeviceInfo message (see below) to see what actually changed in more detail.

An example case when the DeviceCache is changed might be when an AssemblyManager starts a new Assembly providing a new Service on that device. An example of a device that keeps track of some discovery information on some remote device is for example a PalCom Browser. It needs only watch the DeviceCache of the device and as long as that stays the same, it can trust that the collected information is still valid.

DeviceCache is represented with a string as defined by the implementer of the device. The only requirement is that it changes to a new value when the information provided by the devices over the Discovery protocol changes.

7.1.3.2 Device-oriented messages

HeartBeat message HeartBeat is a discovery message that is received repeatedly from other visible devices. Its primary purpose is to report that a PalCom device is, or still is, visible and executing. When there have not been any HeartBeat messages from a device for some period it means that it is unavailable, at least for the time being. This message is sent using the Wire Protocol level directly and encoded/decoded at the MAL level. It is optimized not to include redundant information as described in Section 7.1.2. When the HeartBeat message is forwarded to a ResourceManager it includes the following information of the sending device:

```
HeartBeat ::= DeviceID DeviceCache DiscoveryVer
DeviceStatus DiscoverySelector
```

- *DeviceID* is the unique identity of the remote device.
- *DeviceCache* is the current discovery info version identifier of the remote device.
- *DiscoveryVer* is the version of the DiscoveryProtocol the remote device supports. The current version is named ”D2”.
- *DeviceStatus.* The current status of the remote device. One of three values:
  - *”R”.* Red - not operational
  - *”Y”.* Yellow - partially operational
  - *”G”.* Green - fully operational
- *DiscoverySelector* is the Selector the remote device is using for listening to and sending discovery messages. Selector is defined below.

Only the DeviceCache and StatusFlag fields need to be sent over the wire protocol at each heartbeat, both of which can be kept very short (a byte each), which is important when using a communication media with low bandwidth.

The DeviceID and DiscoverySelector are communicated on the MAL level, typically only when a device becomes visible for the first time.

PRDDevice messages PRD is an abbreviation for Palcom Resource Descriptor.

PRDDeviceRequest PRDDeviceRequest is a unicast message, sent to the DiscoverySelector of a device for requesting device information in the form of a PRDDeviceReply. The DeviceID and DiscoverySelector needed for addressing the device are taken from a HeartBeat message. The PRDDeviceRequest has no parameters.

PRDDeviceReply PRDDeviceReplies are unicast messages with information about the current state of a device, more fine-grained than what is provided directly in the HeartBeat.

```
PRDDeviceReply ::= ServiceCache ConnectionCache AssemblyCache
SelectorCache ServiceStatusCache PRDDevice

PRDDevice ::= Name [DeviceID] DeviceVer [DeviceStatus]
[DiscoverySelector] RemoteConnectSelector
```

- *ServiceCache* is an identifier that can stay unique when the provided services are the same. It must change when a service, that this device provides, is added, deleted, or changed. If the service information is the same, the device can use the same value for ServiceCache even after re-booting or other changes. Table 7.1 shows what information needs to be updated when the respective cache values have changed.
- *ConnectionCache* is an identifier that can stay unique when the connections the device has established are the same. It must change when a connection is established, or closed, by the device.
- *AssemblyCache* is an identifier that can stay unique when the assemblies handled by the device are unchanged. It must change when an Assembly, hosted by this device, is started or stopped.
- *SelectorCache* is an identifier that can stay unique when LocalServiceIDs mapped to Selectors it has provided are still valid. It is updated if the mapping from LocalServiceID to Selector is changed for this device. This is typically the case when the device re-boots if it cannot be guaranteed that exactly the same mapping is maintained. It does not need to change if new mappings are added or deleted.
- *ServiceStatusCache* is an identifier that can stay unique when the ServiceStatus of the Services available at the device are unchanged. It must change when a ServiceStatus, offered by this device, is changed.

- *PRDDevice* follows the structure of a PalCom Resource Descriptor for a Device. Some of the fields are, however, left out when transmitted over the wire, as explained below. The fields are:
  - *Name* is human-readable name of the device, used for GUI purposes.
  - *[DeviceID]* is the unique name of the sending device.
  - *[DeviceVer]* is an identifier and version of the PalCom software on this device. It changes when the software is updated, as discussed above.
  - *[DeviceStatus]* is the same as DeviceStatus in the HeartBeat.
  - *[DiscoverySelector]* is the same as the DiscoverySelector in the HeartBeat.
  - *RemoteConnectSelector* is a Selector used for sending RemoteConnect messages to the device (see Section 7.1.4).

  The current values of DeviceID, DeviceStatus and DiscoverySelector have already been sent in the preceding Heartbeat. Therefore, they are left out when sent as part of a PRDDeviceReply, as indicated by the brackets in the grammar rule above. When a PRDDeviceReply has been received, values from the last HeartBeat from that Device are filled in before delivering the descriptor to upper layers.

  A change in the value of any of ServiceCache, ConnectionCache, AssemblyCache, SelectorCache, or ServiceStatusCache implies that also the DeviceCache changes value.

Comment The information about which services a device offers, and the identity of these services is communicated through a set of messages in the chapters below.

7.1.3.3 Service Naming

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 on a service that is independent on 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 easily be updated after being deployed in this distributed setting. We need to represent the version of a service so the relations between such modifications can be understood and managed.

ServiceID The role of the ServiceID is to cope with the general situation, to identify a particular version of a particular service, independent of from where it is available.

A ServiceID is made up of the DeviceID of the device where it was first created, and the DeviceID of the device where it was last updated. This identity will stick to the service when it is moved (or rather cloned) and instantiated on other devices. The ServiceID also contains information about the previous version of the service which is needed to facilitate automatic update. A service identity is put together as:

```
ServiceID ::= CreatingDeviceID CreationNumber
              UpdatingDeviceID UpdateNumber
              PreviousDeviceID PreviousNumber
```

- *CreatingDeviceID* is the DeviceID of the device where this service was initially created.
- *CreationNumber* is a string that together with the CreatingDeviceID uniquely identifies this 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.
- *UpdateNumber* is a string that together with UpdatingDeviceID uniquely identifies this version of the service.
- *PreviousDeviceID* is the UpdatingDeviceID of the previous version of the service.
- *PreviousNumber* is a string that together with PreviousDeviceID uniquely identifies the previous version of the service.

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

The ServiceID can be seen as the static identity of a Service (comparable to a class name). The ServiceInstanceID (below) is the unique name of an instance of the Service (comparable to an object).

The parts of the ServiceID are defined as bit patterns, and do not need to be readable for a human.

ServiceInstanceID A ServiceInstanceID is used for identifying an instance of one version of a service on a device.

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

- A *DeviceID* is part of the ServiceInstanceID. It identifies the device on which the service runs. In a similar way as for the DeviceID in PRDDevice above, the DeviceID can be left out of the ServiceInstanceID when transmitting over the network, unless the ServiceInstanceID is about a different device than the one sending the message.
- *InstanceNumber* is a string that distinguishes otherwise similar instances of services on a device. This enables the same service to be available in several incarnations 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. InstanceNumbers must uniquely separate instances of a version of a service (having the same ServiceID) on a specific device.

ServiceInstanceIDs are typically stored as part of an Assembly description. They make it possible to safely change to a service on another device knowing that it is providing exactly the same interface. They are also used when safely updating to new versions of Assemblies and the services it relies on.

LocalServiceID `LocalServiceID ::= [DeviceID] ID`

A LocalServiceID is used to address a service on a particular device. It is a short identifier that is used for addressing when connecting to a service.

A DeviceID is part of the LocalServiceID. It identifies the device on which the service runs. In a similar way as for the DeviceID in ServiceInstanceID above, the DeviceID can be left out of the LocalServiceID when transmitting over the network, unless the ServiceInstanceID is about a different device than the one sending the message. This way, the LocalServiceIDs can be kept compact.

A LocalServiceID maps one-to-one with a ServiceInstanceID on a given device. It is used instead of the ServiceInstanceID, because it can be more compact. In contrast to a ServiceInstanceID it only needs to be unique within the device.

When a service is updated, its ServiceID is changed, and the service is given a new LocalServiceID. It is possible that the device will continue to offer both versions of the services in parallel.

Selectors A Selector is a local identifier on a device that is used to connect to one of its services. Selectors are used both to connect to a service (somewhat as a port in Unix) and for the communication when connected (somewhat as a socket in Unix). When establishing a connection a LocalServiceID is translated to a Selector. This Selector is used to contact the service, which in its turn provides a new, dynamically allocated, Selector for that particular new connection (very much like how TCP handles ports). Selectors for established connections are thus short-lived and are invalid after a device reboots. Selectors for services are communicated as part of the Discovery protocol.

7.1.3.4 Service-oriented messages

All these messages are unicast.

When changed value	Implies one or more has changed	What to do	Messages involved to get update
DeviceCache	ServiceCache, ConnectionCache, AssemblyCache, SelectorCache, ServiceStatusCache	Fetch new information	PRDDeviceRequest/Reply
ServiceCache	A service has been added/changed/deleted	Cancel info about Services, get updated info	PRDServiceListRequest/Reply etc.
ConnectionCache	The information about established connections is changed	Cancel info about Connections, get updated info.	PRDConnectionListRequest/Reply
SelectorCache	The mapping from LocalServiceID to Selector has changed	Cancel information about Selectors. Get updated info.	SelectorRequest/Reply
ServiceStatusCache	ServiceStatus of some Services has changed	Update status indicators of presented Services	ServiceStatusRequest/Reply
DeviceVer	Device software and Service organization on the device	Update mapping between ServiceInstanceID and LocalServiceID	LocalServiceIDRequest/Reply

Table 7.1: When does cached information go out of date?

Hierarchical organization of Services Services on a device are structured in a hierarchical fashion for presentation purposes. On each branch in the tree there is a list of available services, or new lists. The leaves in this tree are the Service for which more detailed information can be requested.

This tree-structure of Services is addressed in terms of LocalServiceID.

There are messages to provide more details about the Service, as a Service Description which describes the commands supported by the service. There are also messages to fetch its ServiceInstanceID (to save as part of an Assembly description) as well as its Selector (used to connect to the Service).

PRDServiceList The two messages PRDServiceListRequest/PRDServiceListReply form a request/reply pair used for exploring what Services a device offers.

PRDServiceListRequest A PRDServiceListRequest is used for requesting a PRDServiceList.

```
PRDServiceListRequest ::= [Parent]
```

Parent is a LocalServiceID. If it is empty the request relates to the Device and the response should list the Services organized visible directly at the device level. Otherwise the reply should relate to the Services visible at Parent.

PRDServiceListReply A PRDServiceListReply is the response to a PRDServiceListRequest.

```
PRDServiceListReply ::= CacheNo PRDServiceList

PRDServiceList ::= [DeviceID] [Parent] (PRDService | PRDSubList)*
```

- *CacheNo* is a string, it is unique within the session. It changes if any subordinate PRDServiceList or Service changes.
- *PRDServiceList* is the descriptor of the service list. It contains the following fields:

  - *DeviceID* is the unique name of the sending device. It is, similarly to PRDDeviceReply above, left out when transmitting over the wire.
  - *Parent* is a LocalServiceID (which may be empty, indicating the device itself).
  - For each child item *X* of the service identified by Parent, there is either:
    - *PRDService-X*, or
    - *PRDSubList-X*

PRDService The information provided for each Service in a PRDServiceListReply consists of:

```
PRDService ::= LocalSID Name Distribution HasDesc
RemoteConnect Protocol Reliable ReadableVersionName
ServiceHelpText
```

- *LocalSID* is the LocalServiceID of the Service.
- *Name* is a name that is used to present the Service in a GUI.
- *Distribution* is either '1' (Unicast 1-1), 'N' (Radiocast 1-N), 'G' (Groupcast g-g), or 'B' (Broadcast N-N). See below for an explanation.
- *HasDesc* is either '1' or '0'. '1' indicates that the Service has a PRDServiceFMDescription and the message PRDServiceFMDescriptionRequest will be answered. If a Service does not have a PRDServiceFMDescription that means that other Services cannot connect to it (but it can itself connect to other Services).
- *RemoteConnect* is either '1' or '0'. '0' indicates that the Service will not act if sent a RemoteConnect message, '1' that it will. See WN 119 for information about RemoteConnect.
- *Protocol* indicates the type and version of the service interaction protocol supported as described in Section 7.1.4. The current value is 'P1'.
- *Reliable* is either '1' or '0'. '1' means that a Service connecting to this Service must use a Reliable Connection (see WN111). '0' means that a connecting service does not have to use a Reliable Connection, but is free to choose.
- *ReadableVersionName* is a name that is used to present the version of the Service in a GUI.
- *ServiceHelpText* is a string that describes the Service, for GUI purposes.

A service is characterized by the values of Distibution, HasDesc, RemoteConnect. When Distribution='1', Unicast 1-1, the Service is designed to be connected to another Unicast Service. Information sent by one (or the other) of these Services will only reach the other Service. It is up to the Service to accept 1 or more such connections.

There are four possible combinations of values for HasDesc and RemoteConnect:

- *HasDesc=1, RemoteConnect=0.* This is a common combination for a plain Service that simply offers its capabilities to others, to connect and use. It will not by itself connect to other Services.
- *HasDesc=1, RemoteConnect=1.* This is a combination used for a Service that is designed to connect to another Service. Notice that this is only meaningful if their PRDServiceFMDescription match, at least partially.
- *HasDesc=0, RemoteConnect=0.* This is used in Assemblies, for the services created there, tailored to connect to existing Services. These will connect to the existing Services as managed by the Assembly, but will not be possible to manipulate externally by RemoteConnect messages.
- *HasDesc=0, RemoteConnect=1.* This combination is used for "Meta" services provided by the Palcom framework, offering services that can combine with any existing service, such as debugging and remote control.

When Distribution='N', Radiocast 1-N, the Service is designed to work in a situation with unidirectional communication, with one service sending messages to many receiving services.

- *HasDesc=1, RemoteConnect=0.* This combination is used by a service acting as the radio-station - actually sending out information to many.

- *HasDesc=1, RemoteConnect=1.* This combination is used by a service acting as a radio-receiver -actually receiving information from a single radio-station.
- *HasDesc=0* - these two combinations have the same meaning as for Unicast, above.

When Distribution='G' Groupcast g-g, there is a group of Services that are connected together, they can all send information, and the information will reach all the members of the group, that are available. Members of the group can be distributed in a wide area network. One use of this mechanism is to provide for a group of Assemblies to coordinate.

- *HasDesc=1, RemoteConnect=1.* This is the combination commonly used for Services taking part in groupcasting. This is also used for a Service provided by an Assembly (Synthezised Service), which is designed for groupcast.
- *HasDesc=1, RemoteConnect=0.* As above, but the service will not accept remote connect commands.
- *HasDesc=0.* These two combinations have the same meaning as for Unicast, above.

When Distribution='B' Broadcast N-N, this is similar to groupcast, but the difference is that these messages are not distributed outside the local network.

PRDSubList For items in a PRDServiceList that do not represent an actual service, but act as parents for PRDServiceLists below, there are PRDSubLists:

`PRDSubList ::= Name Kind ListLocalSID`

- *Name* is a name that is used to present the list in a GUI .
- *Kind* is an attribute characterizing the origin of the list, mainly for GUI presentation purposes. Values can be: 'H' for a list used for hierarchical ordering of services and/or lists, 'A' for Assembly and the list will contain the synthezised services offered by the Assembly, 'S' for a list of services offered by unbound components.
- *ListLocalSID* is a LocalServiceID that identifies the list below.

ServiceInstanceID The two messages ServiceInstanceIDRequest/ServiceInstanceIDReply form a request/reply pair used for fetching the unique ID of a Service. This unique ID is used to ensure that future connections will go to exactly the same device when that is demanded.

ServiceInstanceIDRequest A ServiceInstanceIDRequest is used for looking up the ServiceInstanceID corresponding to a current LocalServiceID.

`ServiceInstanceIDRequest ::= LocalSID`

- *LocalSID* is the LocalServiceID for which the ServiceInstanceID should be sent.

ServiceInstanceIDReply ServiceInstanceIDReply is the unicast reply to ServiceInstanceIDRequest.

`ServiceInstanceIDReply ::= LocalSID ServiceInstanceID`

LocalServiceID The two messages LocalServiceIDRequest/LocalServiceIDReply form a request/reply pair used for fetching the LocalServiceID of a Service, given its ServiceInstanceID. This unique ID can be fetched and stored to ensure that future connections will go to exactly the same device when that is demanded.

Assemblies store both LocalServiceIDs and ServiceInstanceIDs for Services it depends on. When the DeviceVer of a device changes, LocalServiceIDRequests can be used for updating to the possibly new LocalServiceIDs.

LocalServiceIDRequest A LocalServiceIDRequest is used for looking up the current LocalServiceID for a Service.

`LocalServiceIDRequest ::= ServiceInstanceID`

- *ServiceInstanceID* is the ServiceInstanceID of the Service for which the LocalServiceID is requested.

LocalServiceIDReply `LocalServiceIDReply ::= ServiceInstanceID LocalSID`
This is the response to a LocalServiceIDRequest.

Selector The two messages SelectorRequest/SelectorReply form a request/reply pair used for fetching the Selector of a Service. A Selector is used for connecting to the Service.

SelectorRequest A SelectorRequest is used for looking up the current selector for a Service.

```
SelectorRequest ::= LocalSID
```

- *LocalSID* is the LocalServiceID for which the Selector should be sent.

SelectorReply SelectorReply is the reply to SelectorRequest.

```
SelectorReply ::= LocalSID Selector
```

- *Selector* is used for connecting to the Service LocalSID.

PRDServiceFMDescription

PRDServiceFMDescriptionRequest PRDServiceFMDescriptionRequest is used for requesting a PRDServiceFMDescription, which describes the command structure of a Service.

```
PRDServiceFMDescriptionRequest ::= LocalSID
```

- *LocalSID* is LocalServiceID of the Service for which the PRDServiceFMDescription is requested.

PRDServiceFMDescriptionReply PRDServiceFMDescriptionReply is a response to the PRDServiceFMDescriptionRequest. It contains a service description of the addressed Service.

```
PRDServiceFMDescriptionReply ::= PRDServiceFMDescription

PRDServiceFMDescription ::= LocalSID
(GroupInfo | CommandInfo | StreamInfo)*
```

- *LocalSID* is the LocalServiceID for which the ServiceDescription is provided.
- For each command the Service offers, there is a GroupInfo, a CommandInfo, or a StreamInfo.

Notice: The Service Name and ServiceHelpText are part of the ServiceListReply.

GroupInfo `GroupInfo ::= Help (CommandInfo | GroupInfo)*`
GroupInfo is used for arranging commands hierarchically in a PRDServiceFMDescription. This is used for presentation purposes when presenting a user interface for interacting with the Service, but otherwise has no semantic meaning. The GroupInfos are normally used for grouping related commands.

- *Help* is a string that describes the command group.

CommandInfo `CommandInfo ::= ID Direction Name Help (ParamInfo)*`
Each command (or ”message”) a Service can accept or send is specified with a CommandInfo. The ID must be unique within the set of CommandInfos listed for the Service.

- *ID* is a (short) identifier for the command, unique within the Service.
- *Direction* is either ’I’ for In (to the Service) or ’O’ for Out (sent by the Service).
- *Name* is a readable string used for GUI purposes.
- *Help* is a string that describes the command.

ParamInfo There is one ParamInfo for each parameter in a command.

```
ParamInfo ::= ID Type Name Help
```

- *ID* is a (short) identifier for the parameter, unique within the Command.
- *Type* is a MIME-type indicator.
- *Name* is a readable string used for GUI purposes.
- *Help* is a string that describes the parameter.

StreamInfo Streams are a kind of Service where information has the form of an open-ended flow.

```
StreamInfo ::= Direction StreamType
```

- *Direction* is 'O' for out (broadcasting end) and 'I' for in (receiver).
- *StreamType* is a MIME-type indicator for the content of the stream.
- *Help* is a is a string that describes the Stream

ServiceStatus The two messages ServiceStatusRequest/ServiceStatusReply form a request/reply pair used for fetching the Status of a Service.

ServiceStatusRequest A ServiceStatusRequest is used for fetching the current status of a Service.

```
ServiceStatusRequest::= LocalSID
```

- *LocalSID* is the LocalServiceID for which the Status should be sent.

ServiceStatusReply ServiceStatusReply is the reply to ServiceStatusRequest.

```
ServiceStatusReply ::= LocalSID ServiceStatus StatusHelpText
```

- *LocalSID* is the LocalServiceID for which the Status is sent.
- *ServiceStatus* is either 'R' - Red, 'Y' - Yellow, or 'G' - Green, indicating a status of 'Not Operational'. 'Partially Operational' or 'Fully Operational', respectively.
- *StatusHelpText* is a string, possibly empty, with a content defined by the Service.

Exactly how the status should be interpreted depends on the Service itself and 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 flashcard available"
- *'Y'* with the text "Flashcard full"
- *'G'* with no text when everything is fine.

7.1.3.5 Connection related messages

PRDConnectionListRequest A PRDConnectionListRequest is a unicast request for the list of current connections established by a device.

PRDConnectionListReply `PRDConnectionListReply ::= PRDConnectionList`

```
PRDConnectionList ::= [DeviceID] PRDConnection*
```

- *DeviceID* is a unique name of the sending device. It is, similarly to PRDDeviceReply above, left out when transmitting over the wire.

There is one set of IDs, one PRDConnection, for each connection the replying device has initiated (where the device plays the role of customer).

PRDConnection A PRDConnection follows the structure of a PalCom Resource Descriptor for a Connection. Similarly as for PRDDevice above, some of the fields are optimized away when transmitting the PRDConnection over the network, as part of a PRDConnectionListReply.

```
PRDConnection ::= [ProviderDeviceID] ProviderServiceID
[CustomerDeviceID] CustomerServiceID
```

- *ProviderDeviceID* is the DeviceID of the provider device. In the case that the provider and the customer are on the same device, ProviderDeviceID is left out.
- *ProviderServiceID* is the ServiceID of the provider.

- *CustomerDeviceID* is the DeviceID of the customer device. It is left out when transmitting over the wire, because that device is the sender of the message, which is known by looking at message headers outside the PRDConnectionListReply.
- *CustomerServiceID* is the ServiceID of the customer.

7.1.4 Service interaction protocol

This document describes the PalCom Service Interaction Protocol. This protocol used after services have been discovered using the PalCom Discovery Protocol. It is used for establishing connections between PalCom services, using the RemoteConnect sub-protocol, and for communication between services while they are operating. The document provides the message formats needed for implementing a PalCom service on top of the PalCom Wire Protocol.

On the service level there can be very different requirements for different types of services, regarding bandwidth, encryption, etc. Therefore, PalCom is open for special service protocols and message formats, optimized for specific requirements. Services that use such specialized protocols declare so in their service descriptions (see Section 7.1.3).

There is one standard PalCom service interaction protocol, described in this section. Services and applications that use the standard service interaction protocol can communicate with services developed within the PalCom project.

In the standard 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 commands with their parameters, when sending them in messages over the network. This format builds on the PalCom Wire Protocol (Section 7.1.2), and on the XML format of CommandInfos (see Appendix E). It is constructed so that messages for commands with textual parameter values consist entirely of human-readable characters. For commands with binary parameter values, the parameters are transmitted in binary.

7.1.4.1 Messages

RemoteConnect The RemoteConnect sub-protocol allows an application on one device, typically a PalCom browser or an Assembly Manager, to establish and close connections between services, running on that device or on others. It consists of two unicast messages: RemoteConnect and RemoteDisconnect. RemoteConnects asks a customer to establish a connection to a provider:

```
RemoteConnect ::= CustomerServiceID ProviderDeviceID ProviderServiceID
```

- *CustomerServiceID* is the ServiceID of the customer.
- *ProviderDeviceID* is the DeviceID of the provider.
- *ProviderServiceID* is the ServiceID of the provider.

This message is sent to the customer device, to the RemoteConnect selector given in its DeviceInfoReply. When receiving it, the device establishes a connection between the customer and the provider (see Section 7.1.2 for information about connection establishment). The customer DeviceID is not included in the RemoteConnect message, because the device knows its own DeviceID.

RemoteDisconnect RemoteDisconnect corresponds to RemoteConnect, but when receiving it the device closes the given connection, instead of opening a new one. The structure of the message is similar:

```
RemoteDisconnect ::= CustomerServiceID ProviderDeviceID ProviderServiceID
```

The message is sent to the customer device, to its RemoteConnect selector.

7.1.4.2 Commands

When connected to a service, commands can be sent to and from it over a connection. The commands are packaged like this:

```
Command ::= CommandInfo Param*;
```

- *CommandInfo* is defined in Section 7.1.3.

The concrete representation when sent over the network is that it is packed in a multipart message, with the CommandInfo as the first part and the values for the parameters as the remaining parts (see Section 7.1.2 for information about multipart messages).

7.2 Resource Descriptors

The PalCom Resource Descriptors are based on the Resource Metadata Model as has been elaborated in [58]. PalCom entities are able to confer information about their resource properties and dependencies. In order to do this a metadata approach is used, i.e., each PalCom resource description includes metadata that describes the functionality of the resource and its prerequisites for use (e.g., required resources by the resource to be available).

The basics of a metadata model as required for PalCom are sketched below:

The first type is called functional metadata whereas the second type is called prerequisite metadata.

Functional Metadata describes exported interfaces, types, and contracts for interface handling, or other information as for example quality of service or charging issues.

Prerequisite Metadata describes essential information for component deployment and distribution.

Prerequisite metadata are in general invisible and only accessed if required. But they are indispensable in advanced composition mechanisms as e.g. in the resource constrained composition explained in [58]. Different formats for storing and providing the metadata are conceivable, the method of choice is, as with the rest of the resource model, to use XML.

To give some examples for metadata:

- memory consumption
- processing power
- energy consumption
- network bandwidth
- latency
- provided interface
- required connection

Figure 7.3 gives an overview of the PalCom Resource Model and the use of Metaddata in the current implementation. Below, a brief explanation of the various elements can be seen.

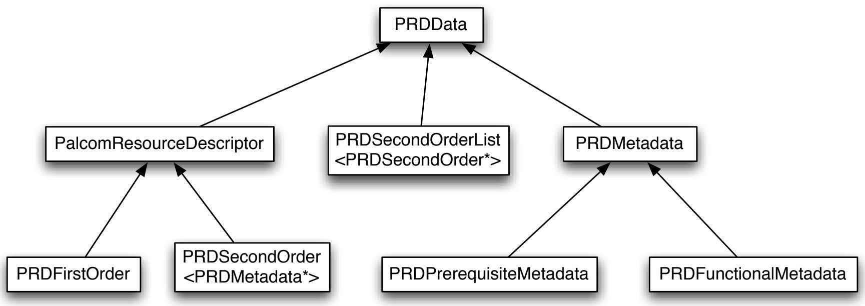


Figure 7.3: PalCom Resource Model Overview.

- PRDData: The base element for Resource Descriptors. PRD is an abbreviation for PalCom Resource Descriptor. All PRDData elements are XML representable and communicable.
- PRDMetadata, PRDFunctionalMetadata and PRDPrerequisiteMetadata: The common abstraction elements for metadata, functional metadata and prerequisite metadata. Metadata may be of atomic nature like *network bandwidth*, or they may be of compound nature like *provided interface*.
- PalcomResourceDescriptor: The common abstraction for Resource Descriptors. All of these are identifiable and have, as such, a name.
- PRDSecondOrder: The common abstraction for Second Order Resource Descriptors. These are the primary unit of discovery. Within the current implementation, not all second order resources have been experimented sufficiently with in relation to resource descriptors, that they are represented as such. Currently, the implementation sports the following Second Order Resource Descriptors: **PRDService**, **PRDConnection**, **PRDDevice** and **PRDAssembly**. A concrete example of a PRDSecondOrder is given further down in the section.
- PRDFirstOrder: The common abstraction for First Order Resources. At the time of writing, First Order Resources have not been experimented with sufficiently with that they are consistently exemplified in the PalCom Open Architecture. The model is constructed so as to acommodate for the presence of First Order Resources in relation to Second Order Resources - both as provided and required elements.
- PRDSecondOrderList: Second Order Resource Descriptor Lists within the current implementation are: **PRDServiceList** and **PRDAssemblyList**.

Currently, the PalCom Open Architecture uses these Resource Descriptors in announcing, discovering and reasoning about Second Order Resources. They are as such used in relation to e.g. making a formal description of a Service available as well as using such descriptions actively in constructing assemblies. The actual use i based on the parsing of various instances of PRDData, making the model easily extendable in practice.

Resources generally are characterized by their identity and a set of attributes. They either describe general properties providing required knowledge in dealing with the resource or specify resource domain specific attributes. The general properties are independent of the specific domain and indicate how the resource management systems deals with them.

- Disposable Resources: A resource is disposable if it is possible to identify a span of program execution over which a given resource instance is consumed. Outside of this span, the resource instance is available for (re)use. As a consequence, usage is not necessarily monotonic. A page of memory is a disposable resource; CPU time is not. An example of the usefulness of this attribute is in allowing unconsuming (i.e., returning to the pool of resources) of disposable resources only. The same operation for a non-disposable resource is erroneous.

- Revokable Resources: A resource is revokable if units of the resource previously granted to the resource consumer can be withdrawn without affecting the consumer's behavior, except possibly for its rate of progress. An example is physical memory: the OS can alter the size of the page frame pool it dedicates to a process's address space without the process noticing.
- Bounded Resources: A resource is unbounded if there is no fixed limit on the amount available. For example, in the absence of a constraint (perhaps issued by the underlying host platform), 'absolute CPU time'is an unbounded resource.
- Reservable Resources: After a successful reservation request of a reservable resource, it is guaranteed that the system is able to supply the reserved units of resource. This does not imply that a client may consume the resource, as that is also dependent on the resource usage policy of the client's resource domain. The definition is phrased in terms of resulting usage, rather than in terms of number of units requested. This distinction is important for disposable resources, since the sum of requested units might overstate actual usage. Any resource can be reservable although attempting to reserve an unbounded resource seems rather pointless (the API does not prevent it, though).
- Resource Units: The resource description should also provide a unit description, which may be expressed in several different systems (e.g., metric, US, etc.) and which can contain standard scaling prefixes (e.g., milli, kilo, etc.).
- Granularity of Resources: The granularity of a resource is the indivisible amount of the resource in a given implementation. For instance, a heap might be managed as a set of pages; in this case, although the resource's unit is bytes or kilobytes, the deliverable granularity is the underlying system's page size, e.g., four kilobytes.
- Measurement Delay of Resource: The measurement delay is is the maximum amount of time that can pass between resource consumption and updating the usage information. For example, controlling the number of open file descriptors can be done accurately at any time (measurement delay is zero), whereas controlling CPU time usage via sampling once a second has a measurement delay of one second. An important implication of measurement delay is the possibility of uncontrolled consumption during the delay interval. If this is undesirable, resource providers should make measurement delay of the resource in question as small as possible. The rationale behind granularity and measurement delay is to allow managing the tradeoff between accounting precision and its cost. Both attributes are important for resource providers; applications are typically insulated from directly dealing with them.

This constitutes the mandatory set of general attributes in the meta-data model. In addition to them there are domain specific resource attributes as e.g.: reliability measures for network connections, mass storage properties, etc.

While in many cases it is useful that only a minimal required set of resource meta-data is provided, some use cases require more in-depth information about dependencies of the resource itself. Such information is within the implementation represented directly in the descriptor, but instead of provided, it is generally fetched on-demand. The major concern in this respect is to provide an appropriate view on the resource. Appropriateness is thereby defined by the context of operation. A completely seamless offer of different views is, however, not feasible and will certainly impose an exceedingly high overhead in generation and processing, that means for practical purposes, a tree representation approach is used.

The runtime representation, and thus this tree representation, of the Palcom Resource Descriptors are instantiated from autogenerated classes made with the JastAdd JAVA compiler compiler system [76]. These are based on a grammar outlining the overall structure and attributes and augmented with various *aspects* addind runtime functionality to the structure. Below is an example sniplet of the grammar defining the structure of Service Descriptor (PRDService):

```
PRDService: PRDServiceListItem ::= LocalSID <Distribution:byte> <HasDesc:boolean>
```

```
                        <RemoteConnect:boolean> <Protocol:String> <Reliable:boolean>
                        <VersionName:String> <ServiceHelpText:String>
                        [PRDServiceFMDescription];

PRDServiceFMDescription: GroupInfo ::= LocalSID ControlInfo*;

abstract ControlInfo: PRDData ::= <ID:String> <Help:String>;
GroupInfo: ControlInfo ::= ControlInfo*;
CommandInfo: ControlInfo ::= <Direction:String> <Name:String> ParamInfo*;
ParamInfo: ControlInfo ::= <Type:String> <Name:String> <DataRef:int>;
```

As can be seen is PRDService a kind of Second Order Resource. It further has a set of attributes, ranging from a version string to a service identifier. Further, the service *may* have a PRDServiceFMDescription (provided interface), which is a compound functional metadata element which is fetched on-demand.

7.3 Reflection through H-Graphs

To accommodate for primarily the inspectability experimentability qualities, but also to a certain extent others, one of the design ideas pursued within the PalCom, is a pervasive use of a dynamic runtime data structure named Hierarchical Graphs (henceforth abbreviated H-Graphs). The primary benefit and novelty of this comes not as much from the datastructure itself, more it is a consequence of its use.

As such, the current use of H-Graphs within the PalCom Open Architecture sports several perspectives. Firstly, the H-Graph is used for structuring and accessing/inspecting most non-transient elements of any process in a runtime environment. That is, both functional entities as well as their data. Secondly, the H-Graph facilitates a simple message-passing and handling mechanism for inter H-Graph node messaging. Thirdly, the H-Graph act as a reflective mechanism facilitating, amongst others, inspection which pr. default allows for a very high degree of inspectability which then can be explicitly disabled.

The H-Graph thus enables introspection and remote manipulation of all nodes, aiding the inspectability and experimentability aspect of PalCom. It is important to state, though, that in relation to the Open Architecture, none of these perspectives are *solely* implemented through H-Graphs at the time of writing.

The notion of H-Graphs is historically tied to, the within PalCom previously used, concept of H-Maps (an abbreviation for Hirarchical Maps). Detailed information about H-Maps and their uses is available in [79]. Inspired by the Plan 9 OS from Bell Labs [43] and the Linux /proc file system, the previously used Corundum framework uses a hierarchical map to support the implementation of palpable components, services and devices. In particular, the use of the externally accessible hierarchical map supports construction/deconstruction of assemblies of services, and visibility and introspection of services and applications.

Although there is a historical relation, H-Graphs differ from H-Maps in several senses. Mainly, given a H-Graph node sibling reference concept and therefrom a secondary graph-like nature, the H-Graphs are not strictly hirarchical like H-Maps. Instead, they can be seen as hirerchically ordered graphs with a primary child-parent relationship and a secondary sibling relationship.

7.3.1 Abstract Description

In general the intention is for H-Graphs to be used instead of inventing ad-hoc data structures. It is, as stated, a dynamic graph-esque runtime data structure. The ontology of H-Graphs is an extension of the ontology of standard object graphs used in object-oriented programs. It consists of the following elements:

H-Graph Node H-Graph nodes are the vertices of the hierachical graph. H-Graph nodes have names and these names are required to be unique amongst siblings. H-Graph Node names are transient and thus allowed to change over time. H-Graph Nodes pass H-Graph Events to each other along the Child-Parent Relationship. The H-Graph Node / Object could be considered at the level where it is like an object with a set of extra mandatory operations.

Child-Parent Relationship The containment relation by which one H-Graph node is subordinated another in the hierachy. H-Graph nodes have no or one parent and no or more children. H-Graph nodes with the same parent are siblings. The Child-Parent Relationship thus defines the hierarchical nature of the H-Graph.

Path A sequence of names of H-Graph nodes at progressively lower hierachical levels. H-Graph nodes have a path at any given time. No two different H-Graph nodes can have the same path. Colloquially, Paths are just like those in a file system. Paths can be seen as symbolic references, allowing for symbolic-path relationships to other nodes in the hierarchy.

Sibling Relationship A type of edge that can only exist between the H-Graph Nodes at the same level in the hierarchy. Sibling relationships are based on the names of the node end points. Thus, they are maintained when possible if one of the node end points change. The Sibling Relationship thus defines the graph-like nature of the H-Graph, and allows for regular object graphs, each within a given level in the hierarchy.

H-Graph Event The type of message passed between H-Graph Nodes. H-Graph Events have a receiver and a sender Path giving the end points of its journey. H-Graph Events have a Command string designating the type of event. It may have a Return Command designating the type of an eventual return event. H-Graph Events may further carry a payload in the form of a Value.

Handler H-Graph node analogous of provided method(s). Handlers are associated with H-Graph nodes in a dynamic fashion, in the sense that they can be added and removed at runtime. Handlers are the end points that handle H-Graph Events based on their Commands and Values. As such, they provide the one part of the functional link to the H-Graph kontext. Handlers are named, though not uniquely, and have a description outlining the purpose of the Handler.

Invokator H-Graph node analogous of required method(s). Invokators are, like Handlers associated with H-Graph Nodes in a dynamic fashion. Invokators are the start points of H-Graph Events, in that they instatiate those and start passing them through the H-Graph structure via their H-Graph Node. As such, they provide the other part of the functional link to the H-Graph context. Invokators are named, although not uniquely, and have a description outlining their purpose.

Listener H-Graph node analogous of event callback(s). Like Handlers and Invokators are Listeners dynamically associated with a given H-Graph Node. Listeners are invokated on the basis of trigger events which are, basically, all node state changing events. Listeners invoke return events to the listening node and, as such, have a return Path and Command and may have pass a payload along.

Value Values are encapsulated non-H-Graph node entities associated with some given H-Graph node. Values have names. Value names are within a given H-Graph context unique.

The abstract H-Graph data type can be described through the following extended BNF:

```
HGraph ::= HGraphNode
HGraphNode ::= Name Path parent:HGraphNode HGraphNode* Value* Handler*
               Invocator* Listener* SiblingReference*
Name ::= String
Path ::= Name*
Handler ::= Name description:String Signature*
Invokator ::= Name description:String
Listener ::= Name description:String Path
```

```
Signature ::= Name parameters:String* returns:String?
Value ::= Name Object
SiblingReference ::= localname:Name globalname:Name
```

Figure 7.4 shows an informal depiction of the H-Graph node and its constituents.

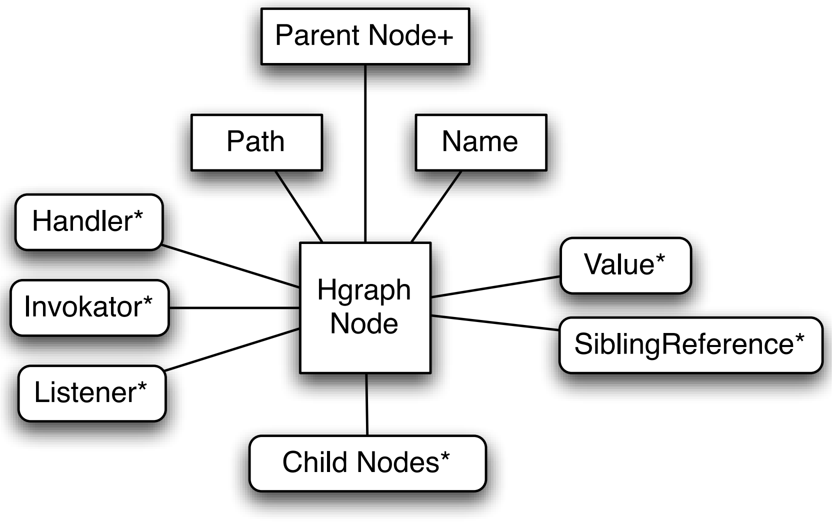


Figure 7.4: Informal pepiction of the H-Graph node and its constituents

7.3.1.1 Relationship with existing PalCom ontology and architecture

H-Graphs are in themselves not part of the PalCom ontology. Instead they are, as described in the introduction, a structural and functional addition to the current architecture implementation. Thus, the link to the PalCom ontology is through the various ontological elements implemented so as to have an H-Graph interface. These, including their relation, are outlined below:

Device Device form the root of the H-Graph of a PalCom Device. Apart from general information, attributes of the Device and other non-ontological elements, it holds the *services* and *middleware managers* subtrees. Within these, Services running on the Device and the three Middleware Managers are installed respectively.

Assembly Assemblies are installed under the Assembly Manager, which manages these. Within the Assembly H-Graph, amongst others, the state of the assembly as well as links to the locations of its constituting second order resources are maintained.

Service Services are commonly installed in the overall H-Graph under a *services* node which is a child of the Device H-Graph node. They themselves form the root of the service-local H-Graph. Apart from the Resource Descriptor, it holds the non-transient data for the service, and thus it is context-dependent how much resides within this H-Graph.

Synthesized Service If an Assembly has a Synthesized Service, it is installed in the Assembly H-Graph. Apart from that and its auto generated sub elements, the Synthesized Service looks more or less like a regular service from the H-Graph perspective.

Resource Descriptor Resource Descriptors are generally installed in the root of the resource which they describe.

Assembly Descriptor Assembly Descriptors are installed under the Assembly Manager.

Middleware Managers The three middleware managers - Resource Manager, Contingency Manage and Assembly Manager are installed under the *middleware managers* child of the Device. These three are at the time of writing at different implementational levels with relation to H-Graphs. Generaly, the intention is, as with Service etc., that the middleware managers store all their non-transient data in their sub H-Graph. As an example, the Assembly Manager has the assemblies it manages as children in its H-Graph.

Figure 7.5 shows the overall H-Graph structure and relation of ontological elements of a PalCom device. The given example has two services and an assembly. The figure further depicts, that the resource, contingency and assembly managers are connected through sibling references.

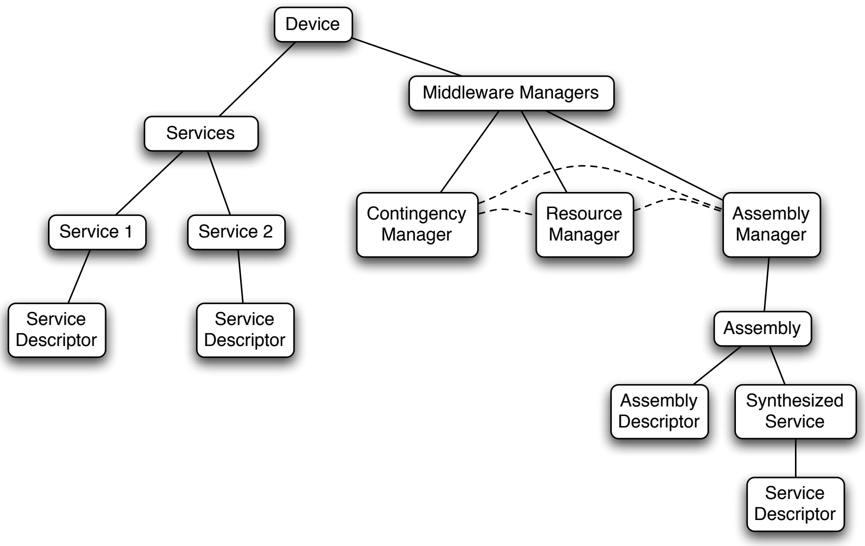


Figure 7.5: The overall H-Graph structure of an example PalCom device

7.3.2 Remotely inspecting PalCom devices through the H-Graph

From an implementational point of view, the remote inspection of some given node is facilitated by the *Hgraph Manager* and the *Hgraph Service*. Here, the Hgraph Service provide a service iterface to the Hgraph Manager. In order to remotely inspect a PalCom Device, an assembly between the Hgraph Service of the inspectee and some remote service with inspection capability will have to be created. For specifics on service interface, communication format and compliance levels of the H-Graph inspection mechanism, see section 10.3. In general, the communicated format is `hgraphXML` - an XML text format required by all H-Graph nodes to be able to return. It is then defined by the strategy and purpose of the inspecting instance how to interpret the returned data and incorporate that into the chosen representation and interaction model. At the time of writing, a command-line based inspection service has been made and a gui inspector based on the *Pal-Visualization* framework (see section 9.1.1) is under construction. For an example transcript of use of the command-line based inspector, see figure 7.6. For a preliminary mock-up of the gui based inspector, see figure 7.7.

```
Waiting for connection...
Waiting for connection...
## > dir

CompassDevice  :  #CompassDevice#
      middleware-managers#
      resourceDescriptor#
      hgraphmanager#
      communication#
      services#

## > cd services

##services > dir

services  :  #CompassDevice#services#
      Compass Service#

##services > cd Compass Service

##services#Compass Service > dir

Compass Service : #CompassDevice#services#Compass Service#
      Test Compass#
      ----values-------
      <name: "compass string"> <value: "18.0,0.0,0.0">
      <name: "frequency"> <value: "500">
      <name: "service status"> <value: "sending">
      -----------------

##services#Compass Service > cd Test Compass

##services#Compass Service#Test Compass > dir

Test Compass : ##services#Compass Service#Test Compass#
      ----values-------
      <name: "status"> <value: "ok">
      <name: "minimum change"> <value: "0.1">
      <name: "frequency"> <value: "500">
      <name: "last"> <value: "18.0,0.0,0.0">
      <name: "heading"> <value: "18.0">
      <name: "pitch"> <value: "0.0">
      <name: "roll"> <value=: 0.0">
      <name: "last sent string"> <value: "18.0,0.0,0.0">
      <name: "last processed time"> <value: "-1928682599">
      <name: "compass"> <value: "CompassService$CompassCommThread@f2c96c">
```

Figure 7.6: A transscript sniplet of use of the command line based H-Graph inspection service

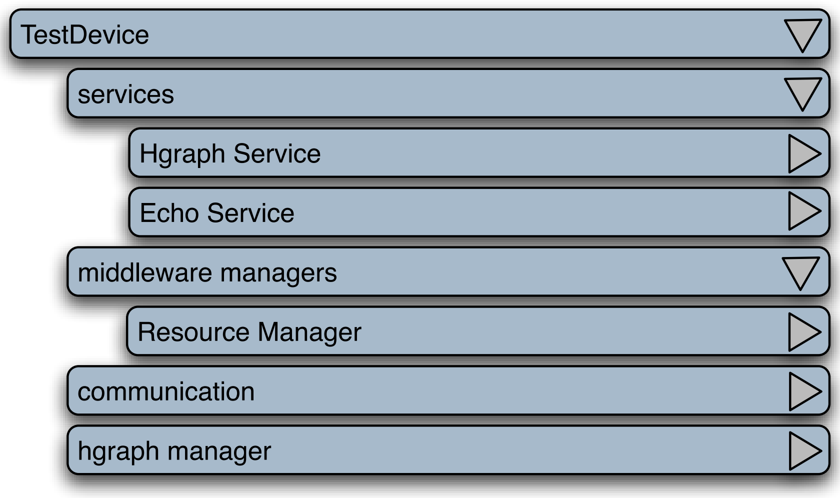


Figure 7.7: A mocked up example of how a simple end-user graphical inspection interface could look

7.4 Processes, threads and scheduling

Details on how processes, threads and scheduling is supported on the `pal-vm` can be found in previous WP3 deliverable [63, Section 8] and [63, Section 9]. As described there, the `pal-vm` supports both a *process* concept, with processes running in isolation from each other, without the possibility to refer to each other's objects, and a *thread* concept, where a scheduler can schedule multiple threads within one process. For threads, there is a library called `palcomthreads`. This section presents the `palcomthreads` library, which is implemented in Java and can also be used on the JVM. The most important parts of the `palcomthreads` library are *PalcomScheduler*, which implements a scheduler, *PalcomThread* for user threads, and the synchronization classes *Monitor*, *Event* and *Semaphore*.

7.4.1 Schedulers

Schedulers are responsible for giving CPU time to user-defined threads and to take care of the situation when there is nothing to do among the threads it handles. In the simple case there is only one object of the *PalcomScheduler* class, which the user needs (typically in the `main` program) to create, and give control to. The default scheduler implementation uses a common priority system to schedule processes. This can be used to give more attention to threads with shorter time-demands. The system is designed for later extensions so it can actually handle hierarchical systems of schedulers.

The `palcomthreads` framework comes with a default scheduler, but it is possible for the advanced user to implement and use his own scheduler, for example using another scheduling scheme.

7.4.2 User-defined Threads

The user defines sub-classes of *PalcomThread* for implementing the functionality that will be run in parallel with other threads. Technically this is implemented as a method `PalcomThread.run`, which eventually will be called by the scheduler.

Internally, the PalcomThreads are implemented using a *coroutine* (see [63, Section 9]). The PalcomScheduler attaches one coroutine at a time, and lets it run until it suspends itself (cooperative scheduling; preemptive scheduling is possible with this scheme, but not yet implemented).

7.4.3 Synchronization classes

For communication between the user-defined threads the following common synchronization mechanisms are available:

Semaphore The semaphore is a simple mechanism that can be used for mutual exclusion or signalling. Simple as it is it should be used with care since it is a bit low-level for most situations. It is normally used by creating objects directly of the class *Semaphore*.

Monitor *Monitor* is a more high-level synchronization mechanism that combines mutual exclusion and signalling in one construct. It is often very practical to use when synchronization depends on data or datastructures that can be inspected and updated by several processes. The user makes subclasses of this class and operations on instances of these classes, becomes mutually excluded.

Mailboxes *Mailbox* is also a high-level synchronization mechanism that typically used for communication of data, or "events" from one process (producer) to the another (consumer). Typically the construct is also used for signalling so the consumer is delayed until data is available. Every PalcomThread has a mailbox which is used for events to the process from the Scheduler as well as events to it from other processes. A Mailbox is technically a bounded FIFO buffer with synchronized operations for inserting and removing messages.

The messages sent to a Mailbox are subclasses of a common class *Event*. In some situations it is practical to include data in such a class, which is thus communicated to the consumer process, and in other situations the information is simply signalling the occurrence of an *Event* itself in which case an empty event is sufficient. The user can create new such subclasses and there are a number of subclasses defined by the PalcomScheduler. These pre-defined subclasses are used to communicate events from the PalcomScheduler to the PalcomThreads.

7.5 Runtime Engine

The PalCom Runtime Engine is realized as a Virtual Machine and supporting libraries. Figure 7.8 expands and highlights the "Runtime Engine" box of Figure 4.1. As can be seen the engine consists of either the `pal-vm` or a standard Java VM.

The essence of this is, that on a Desktop machine, one can choose to execute PalCom code on either the Java Virtual Machine (JVM) or the PalCom Virtual Machine (`pal-vm`). The Core Libraries, Middleware Management, and Utilities are all written in Java code, compatible with `pal-j`, which means that they can be compiled with either the standard Java Compiler `javac` and executed on JVM, or with `pal-j` and executed on the `pal-vm`.

For a specification of the Java VM, see, e.g., [6, 30].

For an elaborate explanation of the dual programming model provided by PalCom, see the PalCom Developer's Companion [71].

The rest of this chapter describes the `pal-vm`[1]. To kkeep the size of this report down, large parts of the VM specification have been left out. The reader is referred to the previous publically available WP3 Runtime Environment deliverable [63], which contains detailed specification on all parts of the VM.

[1]A note on notation: In the following, we often refer to the `pal-vm` as simply "the VM". This is not to be confused with the JVM, which we always refer specifically to, if needed.

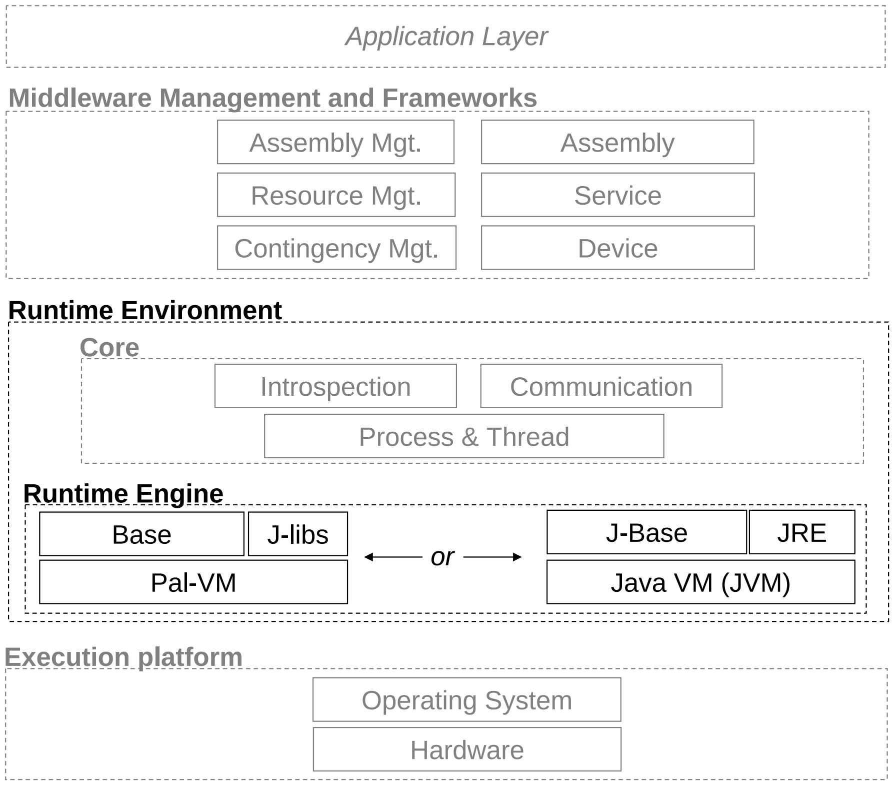


Figure 7.8: PalCom Runtime Engine

In the following we start by justifying the need for a novel VM in section 7.5.1, then an overview of the VM architecture is given in sections 7.5.2 and 7.5.3. These three sections are updates of material from the previous WP3 deliverable 22 [55].

We finally conclude this VM overview by giving an update on the performance (speed and size) of the current implementation in section 7.5.4.

7.5.1 Why build our own Virtual Machine?

Palpable Computing poses a number of special requirements on a Virtual Machine. From the PalCom challenges [42], through architectural prototyping and analysis [48, 4, 73], work analysis [46], scenario work [47, 52] application prototypes [49, 51, 53, 22, 27] and other activities in the various PalCom work packages, a number of such requirements have been identified:

- *Distributed and concurrent processing* – The VM must contain a model for distributed and concurrent processing. The VM must make it possible to define schedulers for concurrent processes supporting pre-emptive as well as non pre-emptive scheduling.
- *Device abstractions and network abstractions* – The VM must make it possible to define device abstractions and network abstractions.
- *Operating system requirements* – The VM must be able to be executed directly on a microprocessor without requiring an operating system. In addition it shall be possible to implement the VM on standard operating systems.

- *Reflective interface* – The VM must define a reflective interface to make it possible to observe and modify software on the fly directly on a device.
- *Resource aware* – Dynamically combining functionality as in an assembly without taking into account the resource capabilities and availabilities of the devices can rapidly lead to runtime errors. Thus, the VM must define ways to observe and manipulate first order resources such as memory, CPU load, battery stand and storage utilization as well as second order resources such as processes, components and devices [58][63].
- *Contingency handling* – The VM must provide the necessary support for fundamental contingency handling, including an exception model, and means for controlling processes, threads, services
- *Migration, update & versioning of components and services* – The VM should support multiple versions of components and services running simultaneously. Furthermore it should support update of a running service without stopping it. Migration of running services to another device (including state) is a subject of future research.
- *Language independence* – The VM must not be tied to one specific language. A family of programming languages including important object-oriented languages must be supported.
- *Portability* – The runtime must be portable to a new hardware platform in a reasonable time.
- *Small devices* – The open architecture and palpable runtime must support devices with a small amount of RAM (lower bound in the order of 256Kb) and slow, low-power microprocessors. For very small devices it may be considered to define a subset of the functionality provided.

None of the above mentioned requirements are unique to PalCom in themselves, but the combination constitutes a very strong requirement, that none of the currently available VM solutions provides.

The following sections describe the `pal-vm`, which has been designed and implemented to address these requirements.

7.5.2 VM Overall Architecture

The virtual machine is designed to support class-based languages. We first give an overview of the overall virtual machine architecture, and in the subsequent sections we details various aspects of the VM.

7.5.2.1 `Pal-vm` Overview

The virtual machine executes one or more processes. A *process* is an independent execution sequence using its own stack. A process contains an isolated collection of objects, in the sense that one process can not hold references to mutable objects, belonging to another process. A process may hold references to an immutable object belonging to another process, though from the point of view of the programmer this is indistiguishable, modulo performance, from the solution where each process has its own copy of immutable data. As such the issue of cross-process references to immutable data is an implementation detail not directly relevant to the system specification.

An *object* encapsulates state, stored in distinctly named fields, and behavior, defined by its class. A *class* is an object that defines a set of behaviors encapsulated into methods. A *method* is a parameterized sequence of bytecode instructions. A method can invoke other methods using virtual dispatching and can signal exceptional conditions to a handler.

A process can communicate with other processes, for example using publish/subscribe. Threads can communicate with other threads in the same process using shared objects.

A process is associated with a *main component*, and is instatiated from one of the classes in this component. A component defines dependencies on other components, which are also instantiated in the process. Therefore a process holds a collection of components, which in turn holds the executable code and data. A *component* is a packaging of classes, resources, and metainformation.

The virtual machine guarantees that the processes and threads executes independently, whereas control inside each process is managed programmatically, typically using coroutines. A *thread* is an independent execution sequence, much like a process, that shares objects with the process it belongs to. A *coroutine* is an object that has its own execution stack which can be suspended (cooperatively or using timer-based preemption) and restarted.

Processes on the `pal-vm` are described in more detail in [63, Section 8].

7.5.2.2 Classes, objects and methods

For each process, the virtual machine executes a sequence of *bytecode* instructions that manipulate the objects that belong to the process. Objects are conceptually always manipulated by-reference thus ensuring the basic integrity of the system (e.g., pointer arithmetic is not allowed). From an extrinsic point of view, an object can respond to a *message send*, which is a selector name and a list of arguments. The response is the execution of a method belonging to the object. From an intrinsic point of view, an object has state which is stored in named fields and can be accessed from the methods of the object. As a special case, some objects may have state that is only accessible in a special way (e.g., the name of a class is obtained through a primitive). Each object contains an explicit or implicit reference to its class, which defines the behaviour of the object and its fields. The methods that belong to the object are conceptually stored in its class.

A *class* is an object which stores a set of methods indexed by selector name. Furthermore, a class has a field which contains a reference to the superclass (if any) of the class it represents. The virtual machine handles a method send to a designated selector by searching through the set of methods in a class and all of its superclasses for a method matching this selector.

A *method* is a named sequence of bytecode instructions that can be executed by the virtual machine. A method takes a number of arguments, as indicated by its selector. The set of bytecodes is described in [63, Appendix B]. A specific bytecode is used to call native (compiled) functions that are built into the VM in order to implement primitive functionality.

All data manipulated by the virtual machine is in the form of objects. Although conceptually sound, this approach requires special support in the implementation of the virtual machine, since all integer values manipulated by the virtual machine also must be represented as objects. The typical approach is the use of *tagged values*, where the least significant bit is used to distinguish integer values from regular objects. This approach is classical in the implementation of untyped languages, and has shown to be applicable to embedded systems [2, 39].

7.5.2.3 Coroutines and Concurrency

The virtual machine supports concurrency at two levels: between processes and threads (which execute independently) and between coroutines inside each process. A coroutine is an object which has its own execution stack. Activating a coroutine causes the current execution stack be be suspended in favor of the execution stack of the coroutine. The activated coroutine executes until it either activates another coroutine (in which case this other coroutine continues execution) or suspends itself, in which case the previous coroutine continues execution. Coroutines can be used to implemenent special cooperative scheduling within a process.

See [63, Section 9] for details on scheduling and coroutines in the VM.

7.5.2.4 Exception mechanism

The virtual machine implements a generic and highly flexible exception mechanism. This mechanism allows exceptions to be propagated dynamically along the call chain from a callee to a caller that handles the exception. Exceptions are represented using objects (of any class), and handlers are written using blocks, which for example allows exception handing in the style of Java. For details, we refer to [63, Section 12].

7.5.2.5 Component Model

As mentioned above, the virtual machine executes one or more concurrently running processes. Each process is an isolated collection of objects, meaning that the objects stored in the process cannot refer to objects from other processes nor can other processes refer to objects stored in the process. Rather, external functions are used to implement basic communication primitives that allow serialized data to be exchanged, for example using publish/subscribe. A process is instantiated from a component, referred to as the *main component*. This component can contain dependencies on other components (expressed using metainformation). These components are also instantiated in the same process.

A persistent component is a deployment unit for classes and resources. It is a binary packaging of class descriptions (including methods and bytecodes), metainformation, and resources, typically generated by a compiler.

When creating a process, all needed components are instantiated from a specific class in the persistent component. The instantiated component are referred to as a *runtime component*.

For more details about processes, refer to [63, Section 8].

The binary component format is described in [63, Appendix E].

7.5.2.6 Language Interoperability

As mentioned in the requirements above, the VM must not be tied to one specific language. The bytecode set, and supporting compilers have therefore been designed with language independence in mind. In the current implementation Smalltalk and Java based components can be interpreted. This rizes the need for *language interoperability*, i.e., the possibility to call and/or inherit classes and methods written in other languages. The mechanisms for doing this in `pal-vm` are detailed in [63, Section 11].

7.5.2.7 Exception Handling

The `pal-vm` supports throwing and catching of exceptions via a generic and highly flexible exception mechanism. The mechanism allows any object to be thrown as an exception. On top of the support provided by the runtime environment, it is possible to build abstractions in library code which extend the exception mechanism to satisfy the requirements of specific languages.

See [63, Section 12] for details on the exception handling mechanism.

7.5.2.8 Resource Awareness

The `pal-vm` is augmented with low-level support for resource management and migration. In particular, primitives and system library code to provide:

- a measure of the amount of memory that has been allocated in a coroutine. By measuring this before and after an operation, device resource management can remain aware of the memory usage of that operation. Note that this measure includes memory that has been allocated but no longer in use (i.e., it could be reused after a garbage collection);
- a signal to a scheduler that is triggered when a certain amount of memory has been allocated. The signal should be resettable by the scheduler. This allows device resource management to limit the memory allocated by a process when performing an operation. This allows a scheduler to keep track of a whole thread or process, not just individual coroutines.
- the above measure and signal based on the number of bytecodes executed rather than the number of bytes allocated (bytecode lenght is not fixed).

The details of the mechanisms are given in [63, Section 13].

7.5.2.9 Reflection

Traditional VM-based object-oriented reflection as in Java and .NET is based on the existence of full metadata for all methods, parameters, etc. in the compiled deployment units. The `pal-vm` does not carry full metadata in components. Although some metadata is available on methods and fields, the types of fields and method parameter types are not included. In the case of Smalltalk components this is natural since the types are dynamic and therefore cannot be determined by the compiler or VM ahead of time. The actual objects that are used when the program is executed are of course all marked with their types (classes and interfaces). Thus, the lack of type annotations cannot lead to unpredictable behaviour as an exception will be thrown if the type assumptions of the programmer fail to hold at run time.

The advantages of not storing type annotation are twofold. Firstly, it saves considerable space which can be at a premium in palpable devices. Secondly, it makes it considerably simpler to load and update code on a device; since there are no type annotations on methods and classes, they cannot become inconsistent when related classes and methods are changed.

The details on the `pal-vm` reflection mechanisms are presented in [63, Section 14].

7.5.3 VM Runtime Evaluation Environment

A number of entities and structures are used during the evaluation of the interpreted bytecodes. This section provides an overview of these.

7.5.3.1 Globals and literals

The evaluation environment provides access to global variables through the current method, current class, current component and current process.

The evaluation environment stores all constants in the literal table, which maps an index to a compile-time constant. Constants can be integers, strings, symbols, constant arrays and constant hash maps. The literal table is method-local.

7.5.3.2 Message send and return

To illustrate the stack layout during message-send and return, Figure 7.9 shows the organization of the stack frames during the send of `add` to `q`. Some of the book-keeping information is removed (displayed as a gray box).

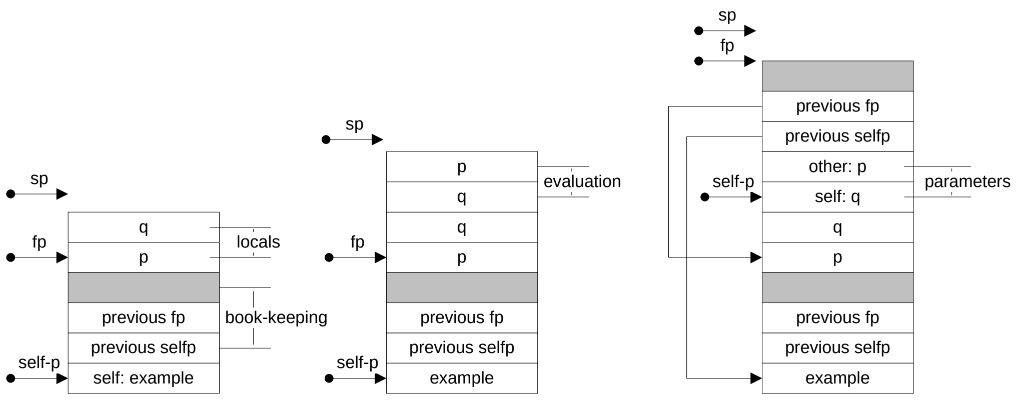


Figure 7.9: `Pal-vm` stack during call

The first part show the state of stack at the beginning of the `run` method in the `Example` process. The second part shows the state just prior to sending the `add` message, and the third part shows stack-frame that has been allocated on the stack as result of sending the `add` message. Notice that the top two elements of the evaluation stack for `run` have become the parameter-area for `add`.

Figure 7.10 shows the stack-layout during return from a message-send.

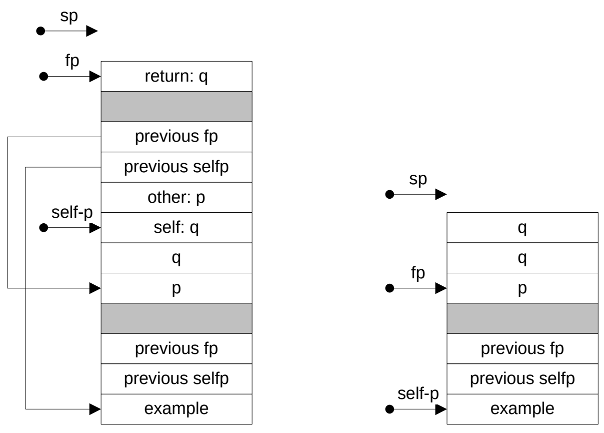


Figure 7.10: `Pal-vm` stack during return

The first part of the figure shows the stack just prior to returning. The self object has been pushed on the evaluation area of the stack-frame of the `add` method. The second part shows how the top stack-frame has been unlinked, and the return-value has been pushed on the evaluation area of the previous stack-frame – after first popping the parameters. All these steps are done automatically as a result of handling the `return` bytecode.

7.5.3.3 Bytecodes

The virtual machine executes a set of binary bytecodes. A list of these bytecodes can be found in [63, Appendix B]. These bytecodes are evaluated in an environment as described above. The organization of the self reference, locals and arguments in the stack frame is shown in Figure 7.11.

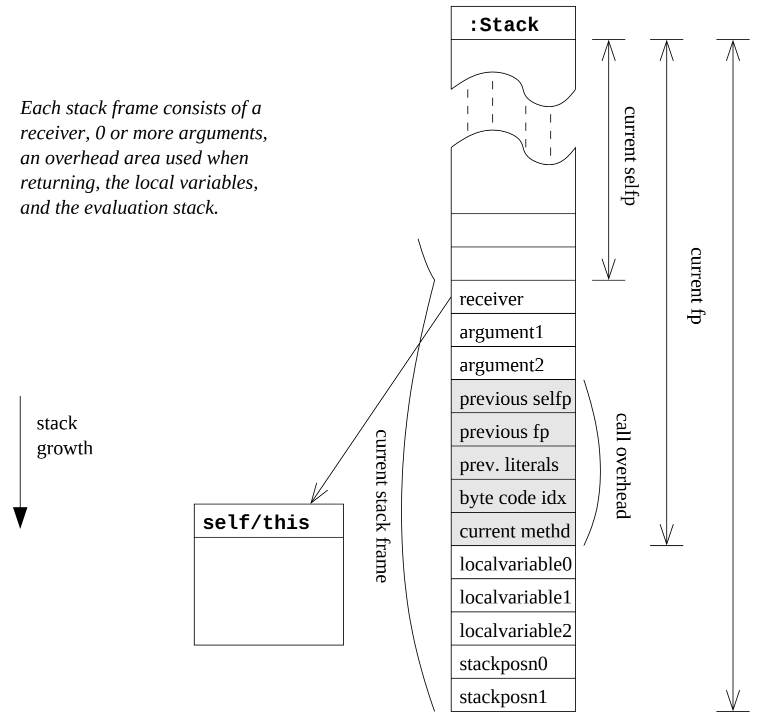


Figure 7.11: Structure of the stack

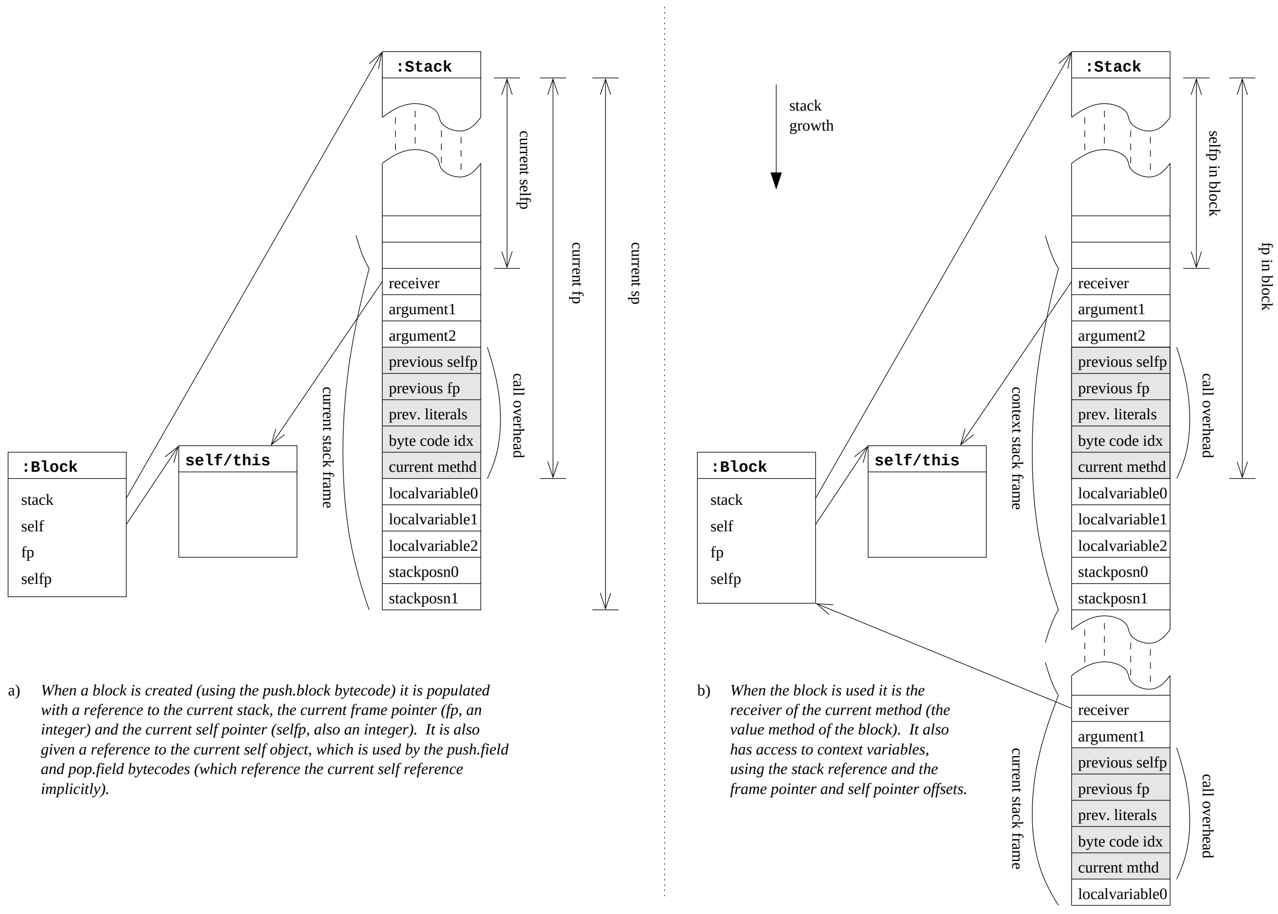
Each method invocation results in the creation of a stack frame on the current stack as shown in Figure 7.11. There are several stacks in the system (associated with different threads and coroutines). The stack is automatically resized on method invocation if it is too small for the new stack frame. The stack object contains offsets that describe the current frame pointer, `fp`, the position of the current receiver object, `selfp`, and the current stack pointer, `sp`.

7.5.3.4 Contexts

The bytecode set previously included several bytecodes designed to handle local variables not on the stack, see Deliverable 22 [55]. These bytecodes, which handled 'contexts' were intended mainly for use in Smalltalk code. Since that time, the main thrust of development has moved to Java code and so the context-oriented bytecodes were removed from the specification.

Blocks are now implemented with a reference to the stack and the offset of the frame where local variables and arguments from the context are available. This is illustrated in Figure 7.12a, where the block object is created, and in Figure 7.12b, where the block object is used.

In the future it may be found more efficient to allocate the blocks on the stack itself, in which case they need only an intra-stack offset to the context variables. In both cases it is a requirement that a block is not used (its method is not executed) after the stack frame which is its context has been popped from the stack (returned from).



a) *When a block is created (using the push.block bytecode) it is populated with a reference to the current stack, the current frame pointer (fp, an integer) and the current self pointer (selfp, also an integer). It is also given a reference to the current self object, which is used by the push.field and pop.field bytecodes (which reference the current self reference implicitly).*

b) *When the block is used it is the receiver of the current method (the value method of the block). It also has access to context variables, using the stack reference and the frame pointer and self pointer offsets.*

Figure 7.12: Implementation of blocks

Violations of this rule would lead to system instability. In order to ensure that this cannot happen, there are certain operations that are not permitted for blocks: They cannot be returned from methods, they cannot be thrown (as exceptions) and they cannot be written into objects (including arrays and hash maps). In addition, they cannot be written into local variables and arguments accessed through the context of a block. The VM contains checks to ensure that these rules are not violated, and throws an exception if they threaten to be violated. In the OSVM virtual machine[3] there were similar restrictions, but they were enforced by static typing in the compiler and name mangling of methods that took blocks as arguments. The solution with static typing may be faster than the runtime checks currently performed by `pal-vm`.

7.5.3.5 Internal bytecodes

In addition to the bytecodes documented in [63, Apendix B], the current implementation of `pal-vm` has some internal bytecodes. These are generated by the VM itself by bytecode rewriting, and are used for two main purposes:

Firstly, those primitives whose principal effects are on the state of the interpreter are simpler to implement as byte codes than as conventional primitives (which use a function pointer lookup to call implementation code external to the interpreter). The `call` bytecodes for these primitives are therefore overwritten with special internal bytecodes when the method is loaded into the VM.

Secondly, some performance improvements have been obtained by bytecode rewriting. Common operations that can be identified in the bytecode stream have been accelerated using special bytecodes.

These internal bytecodes are optional aspects of the current concrete implementation of the VM. They are not available to the compilers, nor do the compilers need to have support for them. If the VM implementation changes then the compilers do not need to be updated, as long as the external bytecodes remain constant. There are currently 48 internal byte codes.

7.5.3.6 Primitives vs. bytecodes

When designing the PalCom virtual machine specification, decisions had to be made whether to implement certain instructions with a new bytecode or with a new primitive. Certain rules of thumb were followed:

- Primitives have a predetermined effect on the expression stack of the interpreter. (All arguments are popped, the result is pushed.) If an instruction is to have a different effect on the expression stack then it must be implemented as a bytecode.
- Instructions that are very rare, for example the instructions used to implement the `name` method in the `Component` class will normally be implemented as primitives. This helps to keep the size of the interpreter down, by not cluttering the bytecode set with rarely used bytecodes.
- The Smalltalk compiler `pal-st` is a very simple one, and there is a straightforward correspondence between source code constructs and bytecodes. Instructions for which Smalltalk syntax exist are normally generated as bytecodes, whereas other instructions are generated as primitives.
- If there is doubt as to whether an instruction should be implemented as a bytecode or a primitive then it is normally implemented as a primitive. The VM can convert the primitive to a bytecode when the method is loaded if this is convenient, whereas the opposite conversion is more difficult: Some bytecodes require less space than a primitive call and do not use an entry in the literal table of the method. Thus, preferring primitives to bytecodes gives the VM implementer most flexibility in designing the VM implementation.
- Once an instruction has been implemented in one way it normally stays implemented that way. This preserves backwards compatibility and avoids the inconvenience of having to modify several compilers, an assembler and a VM at the same time.

Most of the available primitives are used only in one place, in a particular method of a particular class from the base library. This library, `ist.palcom.base`, is written in Smalltalk. The primitives can be used from other languages by calling the methods of the base class. The exception to this rule is the `Object throw` primitive, which throws an exception. This is emitted by the Java compiler when the Java `throw` keyword is used.

7.5.3.7 Reading and writing fields of objects

There are no bytecodes for reading and writing the fields of objects other than the current (this/self) object. See Section 10 in Deliverable 22 [55] for the motivation for this.

7.5.3.8 Meta-class Hierarchy

As mentioned in the `pal-vm` architecture overview 7.5.2, the `pal-vm` heap contains instances and classes. A class, however, is also an object and must therefore be an instance of some class, which in turn is *also* an instance of some class. This leads to infinite recursion unless the recursion is stopped by a class instance that does not have a unique class. Figure 7.13 is an *object diagram* that shows the relationship between class-instances in the `pal-vm`. Each class instance has a *super pointer* pointing to the class it inherits from, and, by virtue of being an object, it also has a *class pointer* pointing to the class of the object.

Since the source-code for a class in both Java and Smalltalk can define both *instance-methods* and *class-methods*, each class in the source-file is compiled to two different class-instances, holding these two sets of methods.

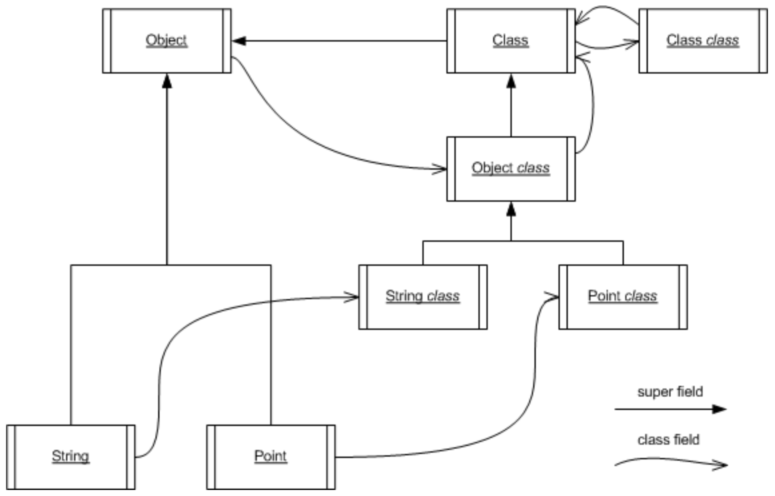


Figure 7.13: `Pal-vm` Meta-class Hierarchy

In the figure, two source-code defined classes are shown (`String` and `Point`) along with their relationship with system classes.

Object
: The class that all *instances* inherit from. Contains the instance-methods of `Object`: `equal`, `hashCode`, `class` etc.

Class
: The class that all class instances inherit from. Since a class-instance is also an object, the super of `Class` is `Object`. Contains the instance-methods of `Class`: `name`, `new`, etc.

Object *class*
: The class of `Object`. The super is `Class`. Contains class methods for `Object` (e.g., deserialization support).

Class *class*
: The class of `Class`. Contains no methods, since a class-instance does not have any class-methods. To stop recursion, the class of the class `Class` is simply `Class`.

String
: The class of string-instances. Contains instance-methods for strings.

String *class*
: The class of `String`. Contains class-methods for strings.

Point
: The class of point-instances. Contains instance-methods for points.

Point *class*
: The class of `Point`. Contains class-methods for points.

As mentioned, to keep the size of this deliverable down, we will not repeat the detailed VM specifications found in [63], but instead refer to that document.

7.5.4 VM Performance and Size

In previous WP3 deliverable [63, Section 16], an overview of the current state of `pal-vm` performance was given. The following section constitute an update on this after year four of the project.

7.5.4.1 VM size

The goal for the `pal-vm` is to be able to run on devices with as little as 256kbytes of memory. There are a number of designs decisions, that will help keep the size of the running VM low. These were detailed in deliverable 22 [55], and deliverable 40 [63].

In the current implementatio the size of the VM itself is around 300kB, depending on the configuration used (Release, Debug, Profiling, Checked, see the Developer's Companion [71]), and the specific processor it is compiled for. This number has not changed significantly over year four of the project; there has been no efforts put into attempting to reduce this size.

The runtime memory consumption of the VM has been improved by implementing a multi-generation garbage collector instead of the original semi-space collector. The gains vary depending of use, but are in the order of 45-50%, much as expected, since the semi-space algorithm is wasting half the memory on the unused semi-space. For platforms like the UNC20 where the operating system only allows for allocation of memory chunks up to 1Mb, the semi-space algorithm also limited the maximum usable memory to 500kb, whereas the generational algorithm may allocate multiple older generations, allowing full utilization of the 8Mb actually available on the UNC20 developer board we have used (see [63, Section 19]).

7.5.4.2 VM speed

As explained in [63, Section 16.2], we have done some benchmarking on the `pal-vm` performance. For comparison purposes, we translated the small BenchPress benchmarks into Java (from Smalltalk), taking care to use 'natural' constructs, in order that the benchmark code should reflect typical use of the Java language. This enabled us to run the benchmarks using the industry standard `javac` compiler and the `HotSpot` VM [20].

Since `pal-vm` is interpreter-based for compactness and simplicity, we compared ourselves with the interpreter built into `HotSpot`. This interpreter is a highly optimized commercial product that has been in very widespread use and continual development for over a decade.

Over the past year of developement we have gained a little performance improvement through a number of minor optimizations. The current performance of the PalCom tools is around 56% of that of the Sun tools, whereas we were around 50% at the end of year three.

As the main focus of the PalCom project is not performance, this is good enough that we do not expect speed to impede the use of our tools.

7.5.4.3 Binary Component Size

The binary `pal-vm` components (as specified in [63, Appendix E]) has been designed for optimal performance rather than minimum size. For instance the Symbols are embedded as standard strings in the components. On a device with small storage this may result in insufficient space to store the deployed component files. This can be avoided in various ways: The most obvious would be to design a more compact layout of the components. An alternative would be to load the components from a remote device, as they are needed. The former method would save bandwidth compared to the latter, but still requires space for the components both in the storage and in the memory of the device. The "remote component server" method could be implemented by designing a "ComponentService" on a large device, and use the palcom transport protocols to transmit the binary components to the smaller device. In the current implementation we have chosen a variant of the "minimal component" approach: Rather than designing a hard-to-debug homemade "optimal" compaction layout for the binary components, we simply apply the GNU GZip algorithm on the components. This reduces the size of the binary components a factor 3-6, and is probably close to being as optimal as any specially designed format we could come up with.

The `pal-vm` has been extended with automatic detection of and unpacking in memory of gzipped components, just as the compilers have been augmented with support for compressing the components.

Since there is a small load-time overhead in the uncompression of the components, this feature is not turned on by default, but requires the compilers to be called with the `-z` option.

Naturally the "ComponentServer" approach could still be applied, even for the compressed components, but this has not been implemented.

Chapter 8

Execution Platform

The Execution Platform seen in Figure 4.1 essentially consists of device hardware and optional operating system(s) running on that hardware.

The execution platform is a prerequisite, but this architecture does not mandate any details regarding technology choices or specific configuration option, however, communication capabilities are necessary for most PalCom deployments.

Besides some device-specific hardware microprocessor, the Execution Platform may contain an operating system. The specification does, however, not mandate the presence of an operating system, and if one is present, it does not mandate a specific operating system. The PalCom Runtime Environment should be implementable on most modern operating systems.

Many First Order Resources are directly related to the Execution Platform, e.g. memory, cpu-cycles, communication bandwidth. The Runtime Engine provides primitives for accessing these First Order Resources, possibly through an Operating System.

The reference implementation for the PalCom architecture has been carried out on a number of different hardware platforms. When using the standard Java virtual machine based Runtime Environment, all computing platforms that support J2SE can in principle be used.

For smaller devices the special `pal-vm` has been developed. This will run in small memories, but has some restrictions on supported Java and Smalltalk features, c.f. [63, Section 17]. Currently `pal-vm` can be used on Intel based Linux, ARM-7 and ARM-9 based Linux, Intel based Windows XP or Vista, Intel and PowerPC based Mac OS X 10.3 or later.

The Java edition for mobile devices, Java ME, does not directly support a sufficiently large subset of the J2SE for the current implementation of the PalCom Open Source reference implementation. However, a current case study within PalCom with a Sun SPOT device [35] indicates that the differences are not fundamental. In that case study, we have had PalCom services running on the Sun SPOT, on top of the Open Source libraries with a thin Java ME adaptation layer in between. The Sun SPOT runs CLDC, which is the Java ME configuration for connected, limited devices. CLDC is also used by smart phones.

Chapter 9

Utilities

As indicated in Figure 4.2, a number of utility libraries are included in the PalCom reference implementation. These are not mandated by the architecture specification per se, but since the functionality provided by these libraries relates to, e.g. the visibility/invisibility PalCom challenge and will be useful in most PalCom applications, we give a short presentation of these in the following.

9.1 Displays

9.1.1 Remote Visualization

Within a palpable environment, the challenge of visualization is a formidable one; Devices of variying sizes and with variying input capabilities compounded by the quality of assemblability calls for a highly dynamic and loosely coupled gui and interaction model. One accomodation strategy for this challenge has been explored within Pal-Visualization, which is a remote visualization framework.

The main driving concern behind this strategy is twofold: Firstly, that from an overall perspective, devices and services cannot be assumed to have the capability of producing a graphical user interface. Neither from a hardware perspective in the terms of owning a display, nor from a software perspective in being able to commuicate a gui. Secondly, Devices and services that *do* have display capability will have to have the responsibility of displaying information not only pertaining to themselves, but also to other devices and services not having this capability themselves. Given that, these can not generally be assumed to have any knowledge about the specifics of the guis they will display during their lifetime.

Based on this, Pal-Visualization have adopted a three element service/assembly level remote visualiation strategy. These three elements are at runtime associated in an assembly and all communication between them is facilitated at that level:

- Display service: Is the service interface to a *display manager* which is a generic mechanism managing the currently available and shown guis (detailed below). The display service has a generic interface used when showing and manipulating all associated gui configurations.
- Displayed service: Is the service which is to have gui displayed. Nothing special in the ways of guis and displaying needs to be assumed about the service, as all communication, both functional and informational, is to be facilitated through its regular service interface.

- Accomplice service: Is the man-in-the-middle, so to say. It is within the context of the accomplice service, that the concrete gui for some given application is prodced. It has a generic part pointed at display services, and an extensible part pointed at spacific displayed services. On the generic side, it provides a service interface to an *accomplice manager* which is a generic mechanism maintaining an internal representation of the gui which is to be displayed. On the extensible side, the accomplice service should also provide a parallel interface to the displayed service(s) it should maintain gui for. It should further contain functionality mapping between the internal gui representation and the service interface directed at the displayed service.

At the heart of this approach lie the accomplice and display managers. Amongst them, they maintain a mirrored internal representation of the gui which is to be displayed. For the sake of the display manager, this may be several representations, depending on how many different accomplices its service is assembled with and it thus displays. See figure 9.1 for a detailing of how the remote visualization communication flow commences. As indicated by

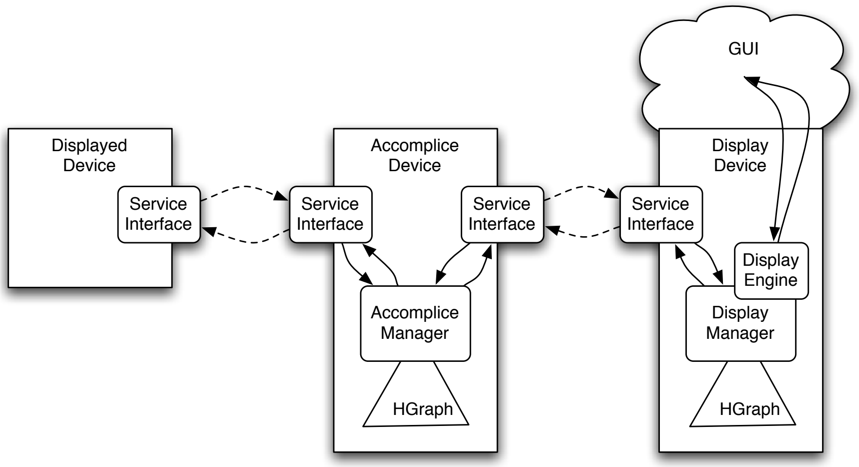


Figure 9.1: Remote visualization communication flow

the figure is the gui representation mirrored between the accomplice and display managers realized as an H-Graph. See section 7.3 for more on the concept of H-Graphs. The reason for choosing this representation is firstly, that it enables accommodation for the inspectability and dynamicity which is sought in this application and secondly, that this structure bears strong resemblence to a traditional gui structure, in that they both are hierarchically structured. The performance of this strategy clearly hinges on the network bandwidth. Owing to the relatively static nature of classical GUIs, it is however quite competitive once the initial tree has been transferred from the accomplice. Afterwards, changes are communicated as atomic as possible.

The mirroring is facilitated by installing a `Listener` listening on state changes in each node on both the accomplice and display side. Once triggered, the listener calls back to its respective manager, which sees to it, that the state change is mirrored. On the display side, the display manager is accompanied by a *display engine*. The display engine governs the relationship between the internal gui representation and the actual gui shown. Depending on the device, such an engine may be based on one of a wide array of display technologies, amongst others Java AWT, Java SWING, SWT and JOGL. The current implementation has a fully functional SWING engine. It is still an open issue of how to relate equal widgets, e.g. *button*, to each other across different engines and on different displays with relation to size, etc.

In order to use the Pal-Visualization remote display framework, there are several things to do and knobs to turn. Firstly, an `AccompliceManager` needs to be installed on the device at which one wishes to use as accomplice. This may be done as part of the accomplice service, or it may be instaled as an indiginous part of the device, servicing several accomplice services. One or more accomplice services will need to be created. This can be done by extending the `AbstractAccompliceService`, or creating a new service with the same interface. Apart from the generic service interface, the developer will also need to create a service interface facilitating communication with the displayed service. This interface should be backed by functionality that provide the link and translation between it and the internal representation as governed by the accomplice manager. Secondly, the developer should choose the use model for the display part. The simplest solution within the given implementation, is to use the `GenericSwingDisplayDevice` - a device having a display service and a display manager using a SWING engine. Alternatively, the display service and display manager can be installed on some preexisting service, also giving the developer the choice of engine. It is possible to create a new display engine, thus taking over the responsibility of mapping between internal representation and actual gui, by implementing the `DisplayEngine` interface.

All code is available in the *pal-visualization* and *pal-visualization-services* projects. One use of the Pal-Visualization remote display framework is the Overview Browser detailed in ref.

9.1.2 Nano-X

In order to support (limited) display capabilities on smaller devices running `pal-vm`, but not supporting e.g Java Swing libraries, wrapper classes in Smalltalk have been made for the Nano-X graphics library. The Nano-X Window System supports a number of embedded devices, and will display on virtually any raster-display attached to these devices. For instance it could be used for the Active Surfaces prototype, if a display was needed there, by running on the UNC20, with an LCD display connected[1].

To cite from [18]: *The Nano-X Window System is an Open Source project aimed at bringing the features of modern graphical windowing environments to smaller devices and platforms. Nano-X allows applications to be built and tested on the Linux desktop, as well as cross-compiled for the target device. The Nano-X Window System was previously named Microwindows, but has been renamed due to conflicts with Microsoft's Windows trademark. There are two APIs implemented in the system, a Win32 API and an Xlib-like API.*

Nano-X supports a number of small displays on a number of platforms. To quote from [19]: *Nano-X currently runs on 32-bit Linux systems with kernel framebuffer support, under the X Window system, or through the popular SVGAlib library. In addition, it has been ported to 16-bit Linux ELKS, real-mode and protected mode MSDOS, and the RTEMS real time operating system. Screen drivers for 1, 2, 4, 8, 16, 24 and 32 bits-per-pixel have been written, as well as a VGA 16 color 4 planes driver. The Nano-X system has been ported to a number of Handheld and Pocket PC's, as well. The graphics engine is capable of running on any system that supports readpixel, writepixel, drawhorzline and drawvertline, and setpalette. Blitting support is optional, but if implemented allows enhanced functionality. All bitmap, font, cursor and color support is implemented on top of these routines. Support for 8, 15, 16, 24 and 32 bit truecolor systems as well as 1, 2, 4 and 8bpp palletized systems is implemented. .*

PalCom Smalltalk wrappers have been made for the following widgets in Nano-X: CheckBox, Cursor, Label, PushButton, RadioButton, TextBox, TextLine, and Window.

A simple code example written in Smalltalk, and which could be executed on `pal-vm` on a display-equipped UNC20 can be seen in Figure 9.2. The example works with the graphics widgets in a familiar way: It begins with creating the singular `nanox` object from the `Nanox` class, which constitutes the "connection" to the display. It then creates a couple of color objects and uses these to create a white and blue window. The it attaches an OK pushbutton and a checkbox with four choices initialized from the four strings in the `buttons` string array. It finally puts a text label on top of this and starts listening to events. This is a fairly trivial example with no interaction possibilities, but it is easy to add eventhandling, e.g. detect pushes on the buttons, by implementing specific methods. Please refer to the online tutorials in the PalCom Open Source toolkit [41] for more extensive examples.

To illustrate the simple graphics that this can create, Figure 9.3 shows the output of running this program – the screenshot is from a desktop computer using the X-Windows binding of Nano-X.

[1] see [63, Section 19] for a descsription of the Active Surfaces and the use of `pal-vm` on the contained UNC20s

```
smallApp = (
  | nanox pbutid1 checkbox1 white blue window buttons |
  run = (
    nanox := Nanox new.
    white := Nanox color: 255 blue: 255 green: 255 red: 255.
    blue  := Nanox color: 255 blue: 255 green: 0 red: 0.
    window := nanox newWindow: 1 owner: self posx: 0 posy: 0 width: 640
      height: 480 bordersize: 4 interiorcolor: white bordercolor: blue events: 0.
    ( window id = -1 ) ifTrue: [ 'Cannot create window' println. ] ifFalse: [ ] .
    pbutid1 := nanox newPushButton: self window: window bgcolor: white fgcolor: blue
      text: 'OK' posx: 30 posy: 230 width: 30 height: 30.
    buttons := Array new: 4.
    buttons at: 0 put: 'choice 1'.
    buttons at: 1 put: 'choice 2'.
    buttons at: 2 put: 'choice 3'.
    buttons at: 3 put: 'choice 4'.
    checkbox1 := nanox newCheckBox: self window: window bgcolor: white fgcolor: blue
      buttons: buttons posx: 10 posy: 45 width: 160
    height: 162 alignment: 0.
    nanox newLabel: window bgcolor: white fgcolor: blue text: 'Make a choice:'
      posx: 10 posy: 40 width: 80 height: 30  box: true.
    nanox listen.
  ))
```

Figure 9.2: Simple Nano-X Code Example

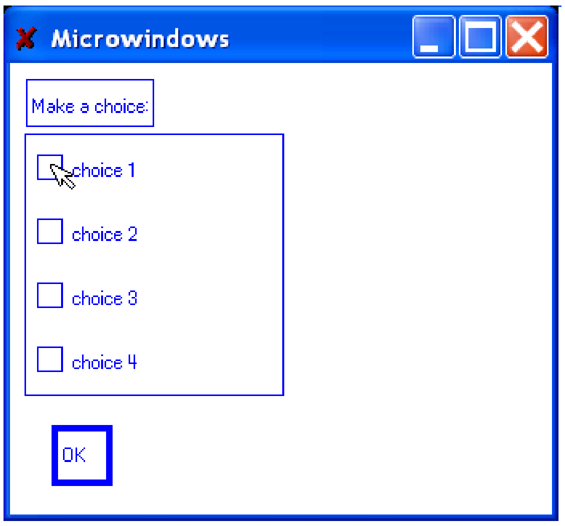


Figure 9.3: Simple Graphics using the Nano-X Window System

9.2 Storage

The PalCom storage components are utilities available for arbitrary storage of data for use by PalCom Services and PalCom Assemblies. The storage service has several abstractions and could be used from the applications as well as services and assemblies layers. The Storage component serves both as a basic component in the Palcom framework but has also some specific features to meet the Palcom Challenges. Construction - de-construction is one example where actual device configuration needs to be stored on a device. Another example is Change - Stability where a Palcom application naturally need to handle dynamic as well as persistent data. However, even more important, data also need to move seamlessly between devices and hence need the storage components facilitate such data migration.

In the first section we will outline the Basic Palcom Storage Service. The Basic Storage component is layered in several layers of functionality that provides different kinds of abstractions to the core services:

- Palcom File System (PFS) - Base layer for Palcom Storage Service
- Palcom IO (PIO) - I/O Abstactions of Palcom File System
- Palcom Persistent Service (PPS) - Serialization of Palcom Objects
- Palcom Tags (PTS) - Tagging of Persistent Palcom Objects

The next section will breifly describe a scenario of use of the Storage component. This will clearify some aspects of the use of the Storage component API.

The last section will describe some work in progress that extend the Storage Component into seamlessly handle the distributed nature of Palcom systems.

9.2.1 Basic Palcom Storage Service

9.2.1.1 Palcom File System (PFS)

At the bottom line, the most fundamental need for a Palcom Storage Service is to use storage in a unified way regardless of underlying hardware. Hence the Palcom File System component plays the most central role among all Palcom Storage Services.

The Palcom File System makes it possible to access and use the storage service in a unified way regardless of underlying hardware. Most central here is the `Storage` interface and the abstract class `AbstractStorage` that defines the minimal implementation of a Palcom file system, i.e. `open`, `close`, `read`, `write` (see Figure 9.4). However implementations could extends this to also more advanced file system functions, such as structured trees and random access. In the current implementation it is possible to run the PFS on a RAM-drive (`ByteStorage`), regular posix filesystem (`FileStorage`), as well as on a flash-memory (`FlashStorage`).

```
public abstract class AbstractStorage implements Storage {
  public abstract int open(String file, int mode) throws StorageException;
  public abstract int close(String file) throws StorageException;
  public abstract int read(String file) throws IOException;
  public abstract int write(String file, int b) throws IOException;
}
```

Figure 9.4: AbstractStorage, specifying a minimal implementation of a Palcom File System

```
public interface Persistent {
  /** Called to persist an object. */
  byte[] persist() throws IOException;
  /** Called to resurrect a persistent object. */
  void resurrect( byte[] data ) throws IOException;
}
```

Figure 9.5: The Persistent interface

9.2.1.2 Palcom Persistent Service (PPS)

Most application programmers of Storage Services will most likely use the Persistent Services. The interface Persistent provides here functions to write / read objects to a `Storage` even if serialization is not available (see Figure 9.5. Furthermore, this layer provides a couple of data abstractions that make it simple to use the Storage from an application, e.g. persistent lists and a simple OODB.

In the example in Figure 9.6 it is illustrated how to use the `Persistent` interface in a class to store objects in the `Storage`.

```
public class PersistentNote implements Persistent {
    protected String   note;
    public PersistentNote(String note) {
        this.note = note;
    }

    public byte[] persist() throws IOException {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        bout.write(getNote().getBytes().length);
        bout.write(getNote().getBytes());
        bout.flush();
        return bout.toByteArray();
    }

    public void resurrect( byte[] data ) throws IOException {
        int stringLength = bin.read();
        b = new byte[stringLength];
        result = bin.read(b);
        if(result>=0) { note = new String(b); } //else EOF reached
    }
}
```

Figure 9.6: Use of the Persistent interface to create a Persistent class

9.2.1.3 Palcom IO (PIO)

The Palcom IO provides an abstraction for Java programs that make the use of regular Java and Palcom IO transparent. In its current implementation three classes have been defined: `StorageInputStream` and `StorageOutputStream` extends the basic InputStream and OutputStream. To handle random access storage StorageAccessStream provides also: seek, position and available. The example in Figure 9.7 demonstrates how Palcom Storage Streams could be used.

```
try {
    Store store = new ByteStorage("/");
    StorageOutputStream out = new StorageOutputStream(store,"file.dat");

    String str = new String("123");
    out.write(str.getBytes(), 0, str.length());
    out.close();
} catch (StorageException e) {
    ...
} catch (IOException e) {
    ...
}
```

Figure 9.7: Use of the PIO classes

9.2.1.4 Palcom Tags (PTS)

The Palcom Tags provides palpable properties to the Palcom storage by enabling data in the storage to be tagged, linked and searched using a variety of methods such as: timestamps, ownership, and keywords. The interface Taggable is an extension of the Persistent interface. The Tags class then makes it possible to set, get tags as well as store a set of tags persistently.

```
public interface Taggable extends Persistent, Cloneable {
    /** Handle object id, i.e. makes objects unique */
    void setid(int id);
    int getid();
    public Object clone();
}
```

Figure 9.8: The Taggable interface enables tags on Persistent objects

9.2.2 Scenario, example of use

In the following we detail a scenario with relation to the "Stone" device[2] from WP9: A pregnant woman has diabetes and must save her blood sugar measurements on the Stone. The G.P. makes a blood type measurement on a pregnant woman and wants to store it on the Stone. The measurement is associated with the right date on the calendar, and with the G.P. A service on the Stone, the DiabetesService, receives the measurements and tags and stores them in the Stone. See Figure 9.9

[2]Personal PalCom mobile storage device

```
public class DiabetesService extends AbstractService {

    public DiabetesServicevDeviceContext(context, String urnLeaf)
      throws IOException {
// create storage
    }

    public void add(int id, String test) {
       // add test(id) to storage
    }

    public void get(int index) {
       // get test: index
}

    public void tag(int id, String tag) {
       // tag obj(id)
}

    public void find(String tag) {
        // find all objs with tag
}

    // Standard Palcom methods like, start(), stop(), etc ...
}
```

Figure 9.9: The Taggable interface used to create a Persistent class

9.2.3 Palcom Storage Extended Service

The Extended Services is also work in progress but aiming at extending the Storage Service by providing features that seamlessly handle the distributed nature of Palcom systems. These extensions include:

- Utilities, additional storage functionality for Palcom systems.
- Bridge locally stored information with networked information, ie distributed storage
- Extend the Tags features to improve the palpable storage, i.e. making information available in time and at place by using palpable models of interactions with stored information

9.2.3.1 Palcom Storage Utilities (PSU)

The Palcom Storage Utilities is still work in progress but will contain functions for storage handling such as: encryption, compression, and backup.

9.2.3.2 Distribution

Handling distributed data and information is a core services in a Palcom network. When executing a task on the local device, files in this virtual file system simply point to the local files, but when a task is delegated for remote execution, the storage managers of the client and the surrogate should be linked, so that remote files may be read transparently. This Palcom distributed system will be kept as simple as possible and is designed specifically for the purpose of cyber foraging (the transient and opportunistic use of servers by mobile devices). It will use on-demand synchronization of file data to reduce the amount of data transferred, and temporary files are only be synchronized if the task is migrated. One plausible implementation model in a massive and dynamic network such as a Palcom assembly is Birman's Bimodal multicast and epidemic information diffusion [9]. In this model information spread in a network based on a probabilistic function.

9.2.3.3 Palpable Storage

Making the information stored in the Palcom Storage palpable is yet another challenge. In the current implementation it's possible to tag information but as the Web suffers from a lack of organized information so will a Palcom network. One solution to this problem could be to use a model based on Tim Berners-Lee semantic web idea [8] that utilizes ontologies to make information explicit. A new twist on the approach would be to form these ontologies in use by utilizing nearby, in time and space, information, and thus allowing devices to process and integrate information automatically when needed.

Chapter 10

Compliance Levels

In this chapter we define four interoperability levels with the PalCom infrastructure. These can be used to categorize alternative implementations of palpable architectures, as well as measures for interoperability with the open source PalCom reference implementation.

10.1 Assembly participation

The most basic type of compliance with PalCom is for a Device to host Services that can participate in a PalCom Assembly. The requirements for Assembly participation compliance are the following:

- Implementation of the PalCom Wire Protocol, as described in Section 7.1.2.
- Support for addressing as defined in Section 7.1.2.
- Implementation of the *announcing* ("responding") part of the PalCom Discovery Protocol, as described in Section 7.1.3, using the XML formats of messages defined in Appendix E. For Assembly participation compliance, it is not necessary for a device to be able to trigger discovery of other devices, only to respond to heartbeats and requests.

It is optional which technology/technologies to support in the Media Abstraction Layer, but in order to interoperate with the largest parts of the implementation in the PalCom toolbox, UDP is recommended.

10.2 Assembly hosting

The next level of PalCom compliance is the ability to host an Assembly Manager which can execute Assemblies. In addition to the requirements for Assembly participation, the following is needed for Assembly hosting:

- Implementation of the complete Discovery protocol, including the messages needed for triggering discovery (heartbeats) and requesting resource descriptors from Devices.
- Implementation of an Assembly Manager that can execute Assembly descriptors as described in Sections 5.2 and 6.2, following the format given in Appendix A of Deliverable 49 [68].

10.3 Inspection

In order to comply with the inspection strategy currently emplored within the Palcom Open Architecture (H-Graphs), there are two levels at which to ensure compliance; Given that the inspection mechanism is exposed to the outside of a Palcom Device by a Service, the top level of compliance is the service interface of the H-Graph Service. Apart from that, it is also a necessity to comply with the exchange format communicated by that Service. The code sniplet in figure 10.1 outlines the compliance requirements on the service interface level. A service facilitating H-Graph inspectability should have the following commands: *Return value*, returning an eventual value for the preceeding command invokation. The format of the return value is `hgraphXML`, the exchange protocol outlined below. There is one exception to this, which is the return value of the *cd* command. Here, the return value is simply either `<true/>` or `false/`. *Cd*, which is taking a stringified H-Graph path as parameter, checks whether the queried path exists in the local H-Graph. *dir* takes a stringified path again and returns a limited view i.e. only a listing og H-Graph children of the node in question. *dir full* acts like dir with the exeption, that all Handlers, Invokators, Listeners and Values pertaining to the node is also returned. *dir recursively* is like *dir* with the exception, that the returned value is the result of a recursive call - giving all offspring of the given node.

```
private static ServiceProxy createServiceProxy(String id){
  ServiceProxy tmp = new ServiceProxy(id);

  Command cd = new Command("cd", Command.DIRECTION_IN);
  cd.addParamAndInfo(new Param("directory path", "text/plain"));

  Command dir = new Command("dir", Command.DIRECTION_IN);
  dir.addParamAndInfo(new Param("directory path", "text/plain"));

  Command dirFull = new Command("dir full", Command.DIRECTION_IN);
  dirFull.addParamAndInfo(new Param("directory path", "text/plain"));

  Command dirRecursively = new Command("dir recursively", Command.DIRECTION_IN);
  dirRecursively.addParamAndInfo(new Param("directory path", "text/plain"));

  ...

  Command returnValue = new Command("return value", Command.DIRECTION_OUT);
  returnValue.addParamAndInfo(new Param("return value", "text/hgraphXML"));

  ...

  return tmp;
}
```

Figure 10.1: Code sniplet taken from the H-Graph Service outlining the service interface level compliance requirements for the facilitation of H-Graph inspection. Pay special attention to the return type of the `returnValue` command. HgraphXML defines the exchange protocol.

The mentioned exchange protocol owes to the way that the HgraphNode interface requires H-Graph nodes to be able to return an hgraphXML representation of themselves. See figure 10.2.

Tag names of the hgraphXML document type are the actual types of the H-Graph nodes. The name and path are stored as attributes under `name` and `path`. Handler, Invokator, Listener and Value representations are written as child tags with types `int_Handler`, `int_Invokator`, `int_Listener` and `int_Value`. These are reserved type names.

```
public interface IHgraphNode{
  ...
  public String toXML();
  public String toXMLFull();
  public String toXMLRecursively();
  ...
}
```

Figure 10.2: Part of the `IHgraphNode` interface being the basis of the inspection exchange protocol.

10.4 Full Interoperability

The final and most complete compliance level we call "Full Interoperability". It implies the above compliance levels, and further demand binary compatibility:

- Full PalCom Interoperability for an alternative JVM based implementation of a palpable architecture is obtained if *Java CLASS files and JAR files produced in the PalCom implementation can be executed in the alternative implementation*
- Full PalCom Interoperability for an alternative Pal-VM implementation is obtained if *PalCom binary PRC files can be executed in the alternative Pal-VM implementation*.

Part III

Appendices

Appendix A

PalCom Open Architecture Highlights

This page presents the main elements of the PalCom architecture, including those considered to make it unique in comparison with the state of the art – highlighted in **bold**.

- Open Architecture
  - Public specification
  - Open-source reference implementation
  - Open communication protocols
  - Open-ended composition and development mechanisms
  - Interoperation with non-PalCom services
- Service Oriented Architecture
  - Loosely coupled independent communicating services on network-enabled devices
  - PalCom Services instantiated from components
  - Services can be stateless or stateful and are network discoverable
  - **Compositions of services into assemblies, including non-PalCom services**
- Dual VM Platform-Independent Runtime Environment
  - Standard Java VM or PalCom Virtual Machine (Pal-VM) with multi-lingual support
  - **Pal-VM exposes explicit low-level resource consumption enabling resource awareness**
  - Pal-VM loads platform independent components compiled from Java and/or Smalltalk
  - Hosts services and assemblies
- Communication
  - Open communication protocols
    - Human-readable representation, yet highly compact and performant
    - Pluggable Media Manager adaptors to any protocol
  - **Adaptive connector selection according to situation and needs of user**
  - Support for publish/subscribe (multicast), point-to-point messaging (one-shot), point-to-point communication channels (e.g., streaming, client/server)
  - **Transparent support for multiple bearers (BlueTooth, Ethernet, Infrared, . . .)**

- Lightweight Resource Management Middleware
  - **Assemblies are the “applications”**
    - Glued together from high-level resources (services/devices/actors/communication channels)
    - Open specification of human-readable assembly descriptors (XML)
    - Scripting to create active compositions of services, devices, etc.
    - Maintains mandatory/optional loosely/strongly specified bindings
    - **Constructed using visual browser or programmatically**
  - Resource Management
    - Persistent discovery of services/devices
    - Provides resource descriptors for Assembly Manager
    - **Awareness** of low-level resources
  - Contingency Management
    - Maintenance of assemblies in cooperation with Assembly Manager
    - Resilience through **reactive and proactive problem compensation**
- Inspectability through open data representation on network nodes
  - **Hierarchical Graph datastructure (H-Graphs)**
  - Uniform access to all data
  - Remotely accessible
  - Challenges the classical notion of Object-Orientation encapsulation
  - **Reflection** $\Rightarrow$ **Inspection** $\Rightarrow$ **Visibility** $\Rightarrow$ **increasing understandability**

Appendix B

A Survey of Service Composition Mechanisms in Ubiquitous Computing

Composition of services, i.e., providing new services by combining existing ones, is a pervasive idea in ubiquitous computing. We surveyed the field by looking at what features are actually present in technologies that support service composition in some form. Condensing this into a list of features allowed us to discuss the qualitative merits and drawbacks of various approaches to service composition, focusing in particular on usability, adaptability and efficiency. Moreover, we found that further research is needed into quality-of-service assurance of composites and into contingency management for composites—one of the concerns differentiating service composition in ubiquitous computing from its counterpart in less dynamic settings.

B.1 Introduction

The ability to seamlessly compose the services from various devices in a more or less ad-hoc manner is a frequently emphasized feature of ubiquitous computing (cf. e.g. [44, 72, 75]). Given the abundant mention of this feature in visions and scenarios, one should expect a sizeable body of research and development work dedicated to its realization. However, in practise we have found far fewer cases than expected where service composition for ubiquitous computing has been a main focus of research efforts. By "main focus" we mean that actually designing, implementing and evaluating a composition technology has been a main part of the contribution of the work. With that said, however, work has been carried out in the area. This paper presents an initial survey of it and draws out the experiences and lessons it holds for continued development of the field. In particular, there are several reasons that prompted us to do this work.

Firstly, we believe, with our colleagues whose work we survey, that there is a real need for good composition mechanisms. This need arises on a variety of temporal scales, including, for instance, the user on fieldwork who is prompted by immediate circumstance to compose a set of services that allows a display on a GPS device to temporarily replace a broken one on an otherwise functional cell-phone. Or the administrator who needs to tailor commercially acquired services to the needs of people in his or her particular organization. Towards the goal of developing a good service composition mechanism, our contribution is to help understand what "good" might mean when talking about service composition for ubiquitous computing, or rather to give an account of advantages and disadvantages of various approaches and identify areas that need further attention from the research community.

Secondly, we wish to support the claim of service composition for ubiquitous computing as an area of study in itself. Composition is a natural concept around which to structure ubiquitous systems and individuate parts of it

[0]**Note:**This chapter is adopted from a survey being prepared for publication by Jeppe Brønsted, Klaus Marius Hansen and Mads Ingstrup.

that are suitably subject to e.g. contingency management, application, or storing, as witnessed by its frequent use as a new and more open-ended replacement for the traditional application concept of desktop computers. This makes it a viable topic of research in itself.

The paper is organized as follows. In section B.2 we describe the approach we have used to arrive at the categorization of the surveyed composition mechanisms used to compare their features in table B.1. Based on this comparison, we move on to a more qualitative discussion in section B.3, before concluding in section B.4.

B.2 Service composition categorization

We only consider architectures that enable composition of services. The following definition of a service is used to decide which technologies we include in the survey:

> A *service* is a unit of runtime software that is accessible by others

This allows for variation in the rationales of service definitions used in the surveyed papers. A further constraint is that the technologies should be targeted towards ubiquitous computing. As a minimum, this means the range of devices the technology works with is not limited to desktop PCs, and that it allows for distributed deployment of the composed services. Finally, we take composition to be a grouping of services interacting with a certain purpose.

In the following, we outline our categorization of service composition features and give examples of middleware that implements those features. We do not propose a rigorous, would-be normative definition of which features are of relevance to service composition in ubiquitous computing. Rather we studied features what are present in systems that can be said to support service composition for ubiquitous computing.

The overall purpose of the survey is to compare and position each architecture relative to the others in terms of the quality trade-offs they engender. In order to enable this we examine what features each composition mechanism has within a select set of concerns.

The categorization framework presented in this section is divided into three main parts that have been selected to inform the discussion on qualitative trade-offs, and to expose variation points in the technologies surveyed. The first part concerns by who, when, and how service composition is specified. The second part deals with runtime properties of the technologies and the third part concerns how the composite is deployed and how it has been evaluated. In table B.1, the categorization framework is applied to the surveyed technologies. In the following sections, the first word in italics describes a column in table B.1.

B.2.1 Specification

Specified by The first step in establishing service composition is to specify what should be composed and how. The mechanism used for this can be aimed at developers as a tool for constructing general applications, or it can be aimed at the user to let him tailor an application for his particular needs. If the developer specifies the composite he can only try to anticipate what services will be available at runtime, whereas if the user specifies the composite he can design the composite from the available services and compose services and devices in new unanticipated ways as in [75, 12].

In Service Composition for Mobile Environments (SCfME) [10] which encompasses protocols for service composition in mobile ad hoc network environments, the composition is specified by application developers through a DAML description [5] of a “Description-Level Service Flow” (DSF) in which sequences of services may be described.

Another example is ICrafter [44] in which users may combine services from different devices and have an aggregated user interface generated.

With UbiDev [31] an application developer supplies an ontology, classifiers, and user interfaces for services in an application. The classifiers map resources on devices into concepts in the ontology. A service is then defined as an atomic action that transforms an input resource yielding a new resource as output. Services are constrained by the concept (from the ontology) that it accepts as argument, and the concept it produces as output.

Specified at The composite can be specified at development time before the composition framework is up and running or at runtime. Note that if the composite is specified before runtime, it implies that it is done by a developer since the user, at that time, has no way of interacting with the system. If the composite can be specified at runtime, there is no need for restarting the composition framework.

Both ICrafter and SCfME provide for runtime composition specification. In ICrafter, users may explore services available at runtime through a general purpose Interface Manager (IM). With the interface of the IM the available services can be explored and a subset selected. An aggregate interface is then requested for the selected set of services. In SCfME an application developer may supply new DSFs at runtime and have these bound dynamically to an "Execution-Level Service Flow" (ESF) containing references to actual services.

UbiDev is more static in the sense that an ontology and corresponding classifiers need to be provided prior to deploying the application. To change the possible service compositions, either the ontology or the classifiers need to be changed, something that appears to be only possible at development time.

Specified in The specification can be specified by providing source code, configurations, or interacting with a tool. If the composite is specified by user interaction, an underlying representation has to be available to make the specification persistent.

In ICrafter, the composition is specified by interacting with a tool in which services are rendered. Services are described with a SDL (Service Description Language) that includes a simple type system oriented towards UI generation.

In SCfME, the service composition is described via a configuration (a DAML XML document) and in UbiDev the specification is partly programmatic since application developers need to supply classifiers.

Level The services in the composition specification can be represented as instances, types, or implicitly. In the case of instances, a particular service is bound to the composite by e.g. an URN [36]. If the services are specified by types there can be several candidates for each service specified. Finally, if the user instead of specifying how services are connected, requests a task from the framework that has to be resolved into a composition of services, we regard the services as being specified implicitly.

With respect to the level of description, SCfME is characteristic of an approach in which the composition is described at a type level and resolved at runtime by the service composition implementation to an instance level. ICrafter, on the other hand, operates only on an instance level in that it is specific services that users select. Finally, UbiDev operates on an implicit level based on the ontology and classifiers specified. One example is a generic messaging service, *document_to_display* that can be adapted to the context by UbiDev. If the user context is a personal phone, UbiDev will compose the services *ascii_to_wav*, *wav_to_adpcm*, and *adpcm_to_voice* to provide the *document_to_display* service.

Quality of Service (QoS) A few of the technologies have support for specifying quality of service requirements in the composite specification. If, e.g., the services are specified as types the selection of the actual instances can be guided by the quality of service specification.

In the Amigo service composition mechanism [77] services are described as semantic Web services (using OWL-S [32]) in which atomic processes have QoS attributes with values obtained from runtime measurements. An example of a QoS requirement would be that latency should be less a threshold. At runtime, an abstract specification of a composite service is matched against possible realizations, the latencies for the realizations are calculated, and the best composition is chosen (also subject to other constraints) [37].

B.2.2 Runtime

Contingencies The technologies surveyed provide different levels of support for contingency handling. The trivial option is to have no support for contingencies at all. A slightly more advanced solution is to detect the contingency and alert the user, and a natural extension of this approach is to resolve the contingency automatically. It should be noted that in some cases, it is impossible to resolve contingencies automatically and in these situations, the user should be in the loop.

As an example, in SCfME the state of the execution of a composition is check-pointed by sending partial results back to the service requester. If the Composition Manager fails, the service requester is notified and may create a new concrete ESF based on the original abstract DSF and its latest checkpoint and finds a new composition manager.

In UbiDev, since the composition is implicit and dynamic, contingencies such as partial failures will be handled automatically as classifiers find new ways of realising compositions.

Resource use Since many devices for ubiquitous computing have limited resources its relevant to look at resource use. Since its not feasible to make comparable measurements for all the technologies, we have made a rough estimate of what kind of system the composition mechanism requires. One type is a server platform and another is a typical PDA. A third category would be a smaller sensor/actuator platform such as Motes (`http://www.tinyos.net/`).

A number of solutions apply semantic Web technology (e.g., SCfME and Amigo) which means that at least parts of the middleware will not scale to low-end devices. Other than that most service composition mechanisms seem to aim at PDA-type device and only DSCiPC [25] integrates sensor/actuator platforms.

Scalability As with resource use we only provide an estimate of how the composition mechanism scales.

In Amigo, a part of the service composition mechanism was to build a “global” automaton. It is not clear what the scope of that automaton is, but together with the use of OWL-S this means that the scalability of Amigo is low.

All devices in one.world run the one.world system platform. Services are composed in a decentralized manner and thus no node has the sole responsibility of the composite. This makes it possible to create composite services consisting of arbitrarily many services.

System	Composition Specification					Runtime			Deployment		
	Specified by	Specified at	Specified in	Level	QoS	Contingencies	Resource use	Scalability	Infrastructure	Topology	Evaluation
Amigo	End-users	Runtime	End-User int.	Implicit	Runtime	Automatic	PDA+Server	Low	Fixed	Centralized	Sce. perf.
Aura	End-users	Runtime	Configuration	Types	Runtime	Automatic	PDA+Server	Unknown	Ad Hoc	Centralized	Example
Daidalos	Unknown	Runtime	Configuration	Types	Runtime	Runtime	Unknown	Average	Unknown	Centralized	Example
DSCiPC	App. Devs	Runtime	Configuration	Implicit	Runtime	Automatic	PDA+Mote	Average	Ad Hoc	Decentralised	Ex., Perf.
GAIA	App. Devs	Dev. time	Configuration	Types	None	Automatic	PDA+Server	Unknown	Fixed	Centralized	Example
ICrafter	End-users	Runtime	End-User int.	Instances	None	None	PDA	Average	Fixed	Centralized	Example
Obje	End-users	Runtime	End-User int.	Instances	None	Detection	PDA	High	Ad Hoc	Decentralized	Example
one.world	App. Devs	Runtime	Source Code	Instances	None	Detection	J2SE	High	Ad Hoc	Decentralized	Sce., Usab., Perf.
Ozone (WSAMI)	App. Devs	Dev. time	Source Code	Types	Runtime	None	PDA	High	Ad Hoc	Decentralized	Ex., Perf.
PalCom	End-users	Runtime	Configuration	Instances	None	Automatic	PDA	Low	Ad hoc	Centralized	Scenario
Paths	App. Devs	Runtime	Configuration	Implicit	None	None	PDA	Average	Fixed	Centralized	Scenario
QuAMobile	App. Devs	Runtime	Configuration	Types	Runtime	Automatic	PDA+Server	Unknown	Ad Hoc	Decentralized?	Scenario, Perf.
SCfME	App. Devs	Runtime	Configuration	Types	None	Detection	PDA	Low	Ad hoc	Decentralized	Performance
SpeakEasy	End-users	Runtime	End-User int.	Instances	None	Detection	PDA	High	Ad Hoc	Decentralized	Ex., Usab.
TCE	End-users	Runtime	End-User int.	Instances	None	None	PDA	High	Ad Hoc	Centralized	Example

Figure B.1: Categorization of service composition mechanisms. The references used for each technology are as follows:Amigo [77, 37], Aura [15, 74], Daidalos [78], DSCiPC [25], GAIA [72], ICrafter [44], Obje [12], one.world [17, 16], Ozone (WSAMI) [24], PalCom [75], Paths [28], SCfME [10], SpeakEasy [11, 13, 38], TCE [34, 33], UbiDev [31], QuAMobile [1]

B.2.3 Deployment

Infrastructure In ubiquitous computing environments services might not be connected at all times. Devices may enter and leave the network and therefore it is important whether the composite has to have a fixed connection during operation or whether devices can enter and leave on an ad hoc basis.

In SCfME devices are peers and communicate via ad hoc protocols. In particular, a Group-based Service Routing (GSR) on-demand protocol in which the route is constructed during service discovery is used for routing service invocations.

In contrast, ICrafter assumes a fixed infrastructure in which a global communication infrastructure which resembles a tuplespace may be constructed.

Topology Some of the mechanisms require that a central node acts as a coordinator for the composite and thereby imposing a centralized structure in the composite, whereas other mechanisms allows for a decentralized structure.

SCfME is decentralized in that after an ESF is created, any node in the system can be used as a Composition Manager that handles the execution of the composition. The Composition Manager may be dynamically changed during the execution.

Based on finite state automata for individual OWL-S processes, the Amigo service composition mechanism builds a global finite state automaton for all services. In this sense, the Amigo service composition mechanism relies on a central component.

Evaluation Most of the technologies presented are evaluated by demonstrating that the composition mechanism can be used to implement various ubiquitous computing applications. *Example* applications invented by applications developers typically demonstrate a wide range of features whereas application *scenario*s developed in cooperation with users, have emphasis on posing relevant challenges. In some of the projects, it is evaluated how usable the composition mechanism is for users and/or developers, and, finally, some projects present measurements of various performance parameters.

In SpeakEasy, an example application, *Casca*, supports collaborative work by letting users share files, devices, and services on an ad-hoc basis, and thus demonstrates various features of the composition mechanism. Usability is evaluated by observing users interacting with the SpeakEasy service browser to accomplish a task.

The PalCom service composition mechanism have been used to implement a set of application scenarios that have been developed in cooperation with end-users using participatory design techniques. The scenarios range from enabling landscape architects to visualize digital data in the context of the physical world to supporting rehabilitation of children with Downs syndrome.

B.3 Qualitative aspects of service composition

In the following we discuss composition mechanisms from an architectural perspective taking the categorization presented as an outset. In doing so, we look at (architectural) software qualities [23] of the composition mechanisms that are of particular relevance to ubiquitous computing.

B.3.1 Usability

Usability is defined in one the predominant quality frameworks as "concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides." [7, p. 90]. This includes understandability, learnability, operability, and attractiveness [23].

Although the traditional view is that usability is first and foremost determined by the user interface of systems, there are several ways in which the architecture may strongly influence the usability of a system, e.g. the support for undo functionality [7]. Similarly, the service composition mechanism influences usability in a number of ways.

Our main criteria for assessing the usability of a composition mechanism is whether the composite services built with a given mechanism suit the users optimally given what services are available for composition. A strong determinant of this is how the composite services come to exist. Another is whether a composite, once created, can adapt (or be adapted) to change of circumstance, such as due to contingent availability of services and resources or malleability of users' intentions with the composition. We discuss the first issue below and the second in the next subsection.

Regarding the first, we may question whether developer-specified, users-specified, or e.g. automatic synthesis based on interpretation of high-level sentences [14] leaves the user better off. The PalCom assembly concept is based on an elaborate conception of this issue. It is held that autonomic, perhaps AI-based, composition or developer-specified composition based on anticipation of the user's needs are fine in so far as the resulting composite does what the user wants it to. However these approaches are for a variety of reasons not flawless and complete enough to work consistently and so they need to be supplemented by architecture and tool support that empowers the user to create composites or modify existing ones to suit their purpose. [21]

Thus the way a composite service is specified is influential on usability by (1) deciding whether the users can specify composite services themselves, (2) whether they can do so at runtime and (3) how it is done. Only 5 of the 12 surveyed technologies enable end-users to specify composite services[1]. Lack of support for end user composition appears in several cases to be due to the focus of the research project in question rather than a fundamental constraint of the architecture it produces: In 3 of the remaining 7 composition mechanisms the composite is specified in a configuration file and at runtime. These are arguably only a composition tool away from supporting end-user composition, albeit the configuration file format may be more or less constraining depending on its level of support for scripting.

In general, the considered composition mechanisms may be geared more towards the user or the developer. In ICrafter, for instance, the composition is explicit mainly in the tools used by the user, rather than at the architectural level. Speakeasy, in contrast, does not provide tools for the end user to do composition, but consists of a more general framework that developers can use in their applications. Concerning operability we must assume users have a greater range of tasks they can perform if the composition mechanism allows them to specify new composites, and that this also allows for greater adaptability in the short term.

B.3.2 Adaptability

Adaptability is the "capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered" [23, p. 11]. (In [7, p. 80] this is related to runtime 'modifiability').

The majority of the surveyed composition mechanisms (9 out of 12) allow service composition to be specified at runtime. Furthermore, most mechanisms do service discovery and coordination based on specification of needed types of services. This all supports adaptability which is arguably a central quality of ubiquitous computing systems

[1]PalCom Assemblies, listed in table B.1 as being specified in a configuration file, has at least one tool for allowing users to compose services (it uses the parsed configuration file as a living representation, and generate a file one when it is saved)

that are often characterized as being able to work in a heterogeneous, changing system context. SCfME, e.g., has the possibility of providing a high-level (type-based) description of a needed service at runtime.

The three dimensions of *specification* imply different temporal scales at which adaptation can be supported. Source-code and developer based specification that happens at development time necessitates a much longer turn-around time for adaptations than do end-users operating at runtime.

On the other hand, adaptability is less well supported for contingencies such as services disappearing or failing. An example, though, is UbiDev in which the Service Manager is responsible for rebinding services upon partial failure.

Another aspect of adaptability is in deployment where supporting an ad hoc infrastructure in a decentralized topology has the potential of being most adaptive. Obje, one.world, SCfME, and SpeakEasy all support this combination whereas none of these have automatic contingency management.

B.3.3 Efficiency

Efficiency is the "capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions" [23, p. 10]. This pertains to both time behaviour and resource utilization such as memory use. Again, this is arguably a central quality of ubiquitous computing in that embedded systems often play a central role.

While the cited articles do not report sufficiently detailed performance data to enable a quantitative comparison of their efficiency we can nevertheless infer a tradeoff between performance and adaptability.

Among the most dynamic and adaptable technologies (those with ad-hoc and decentralized infrastructure that enable composition at runtime: 6 out of 16) we can expect a performance penalty to be associated with the additional logic necessary to handle setup/changes at runtime. However if that logic is only activated when strictly necessary there is no apparent reason why a more dynamic and adaptable system cannot, in a concrete case, perform as well as one into which many assumptions have been hardcoded.

Apart from that case where the tradeoff is potential, it is more certain that a specification working at the level of types (or is implicit) must use additional resources on resolution compared to an instance-based specification. For this tradeoff 10 of the 16 surveyed technologies have chosen the more flexible but poorer performing solution of type-based or implicit service specification. However lack of performance data prevents a quantitative assesment of that performance penalty.

Further towards favouring the adaptability end of the adaptability-performance tradeoff, the specification of composites may support quality-of-service requirements which implies a more complex and therefore costlier resolution step. In addition to this, 6 of the 16 technologies have some support for monitoring QoS at runtime. This enables detection of non-compliance with a QoS requirement which can initiate reconfiguration to ensure better reliability, but again at a cost of increased resource consumption.

It is characteristic that none of the surveyed technologies appear to scale to low-end devices such as sensors even though this may be relevant. In addition, few of the composition mechanisms are highly scalable with one.world and SpeakEasy as exceptions.

B.3.4 Security

Security is "the systems's ability to resist unauthorized usage while still providing its services to legitimate users" [7, p. 85]. More specifically, a system should provide non-repudiation, confidentiality, integrity, assurance, availability, and auditing.

General security issues in service oriented computing also apply for service composition. In some situations, users and services should be authenticated (by e.g. using certificates) and communication should be encrypted. Also ubiquitous computing security issues are relevant in the context of this paper. Only few of the surveyed technologies explicitly handles these issues explicitly.

Directly in relation to service composition, one problem is, given a set of services, to select the services that should be part of the composite so that the security requirements of the application are met. In systems where the level of the composite specification is *instances* (6 of 15 in table B.1), the responsibility of matching the security requirements to a set of services is left to the user of the composition mechanism. Of the 6 composition mechanisms, none supports the user in this process.

When the composite specification is abstract (denoted *types* and *implicit* in table B.1) the system itself has to select each service in the specification from a set of candidates. 6 of the 15 surveyed technologies have support for quality of service requirements in the composite specification and some of these technologies explicitly state that security requirements can be specified as QoS requirements. E.g., in the QoS model in Amigo [77, 37] security is represented by three boolean parameters: confidentiality, integrity, and non-repudiation.

B.4 Conclusions and Future Work

This paper has presented an initial survey of service composition in ubiquitous computing. Quite a few service composition mechanisms exist of which we have only reported on a few here. In doing so, however, it has been characteristic that no papers have focused mainly on service composition. The characteristics that we have outlined show that there are indeed opportunities for doing so.

In particular, service composition is entangled with several complex features such as service discovery and matching, contingency management, and reconfiguration and therefore provides a good frame around which to explore those features and their interrelation. This, however, is an area where further research is needed: the surveyed technologies only begin to scratch the surface of contingency management and efficient resource management. Likewise, more work is needed to explore the relationship between manual and autonomic execution of functionality. Concerning the notion of service composition, we found very limited signs of research that leveraged the result of previous research in the area. Such leverage is arguably easier to achieve with a common conceptualization of the research area, and we consider this survey a step towards that in addition to providing an overview.

The obvious further work on this survey is to make it more complete in several ways. One is by including further examples of relevant ubiquitous computing systems. Another is by considering other forms of service composition (such as the ones used for web services) and discuss their relevance to ubiquitous computing. Both of these directions we believe will benefit the development of quality service composition mechanisms in ubiquitous computing.

Appendix C

Wire Protocol: Message Node Cross Reference

```
Message == Heading DataNode?
DataNode == MultiPart | SingleMessage

Heading == <HeartBeat> <HeartBeatAck> <HBInfoRequest>
<HBInfoReply> <HeartAttack> <FormatVersion>
<RoutingR> < RoutingS> <Mark*> <Connection>
<Reliable> <AckMessage> <ResendMessage> <Chopped>
<SingleShot> <PubSub>

Multipart == '+';<l>;<data>
SingleMsg  == 'd';<l>;<data>

FormatVersion == 'v';l;<Wire-ver-identifier>;<Discovery-ver-identifier>
HeartBeat == 'h';l; < CachedInfo>; <StatusInfo>
HeartBeatAck == 'H';l;<CachedInfo>; <StatusInfo>
HBInfoRequest == 'i';l;
HBInfoReply  == 'I';l;<DeviceID>;<DiscoverySelector>
HeartAttack == 'X'
RoutingR == 'r';l;ShortID
RoutingS == 's';l;ShortID
Mark == 'M';l;<data>

Connection == 'c';l;<f>;...
Where f can be:
  Radiocasted = <f>='b';selS
  Open = <f> = 'o';selR;selS
  OpenReply = <f>='p';selS;selN
  UsedID = <f>='u';Authentication
  Close = <f> = 'c';selR;
  Re-open = <f> = 'r';selR;
  Message over connection = <f> = 'm';selN
Reliable == 'R';l;<Seq>
AckMessage == 'A';l;<Seq>
ResendMessage == 'B';l;<Seq>
```

```
Chopped == <'-'>;<l>;<M-id>;<part>;<parts>

Single-shot message  == 'S';l;selR;selS
PubSubMessage == 'P';l;<Topic>
GroupJoin == 'g';l;<GroupID>
GroupMessage == 'G';l;<GroupID>
GroupLeave == 'q';l:<GroupID>
```

Appendix D

Wire Protocol: Example Messages

Note: all the lengths are written as l, but in a real transmission they need to be calculated. Spaces are inserted in between message-nodes in the examples below for readability.

1. Message over an existing connection, using the selector `selN`:

   `'c';l;'m';selN 'd';l;<DATA>`

2. Large message, this is part 2 of 5. The messages are given the id '11'.

   `'c';l;'m';selN '-';l;11;2;5 'd';l;<DATA>`

3. Reliable message, this is message 97 over the connection:

   `'c';l;'m';selN 'R';l;97 'd';l;<DATA>`

   (a) The previous message received was 96, so the receiver responds with;
   `'c';l;'m';selN 'A';l;97`

   (b) The previous message received was 94, so the receiver responds with:

   ```
   'c';l;'m';selN 'B';l;95
    'c';l;'m';selN 'B';l;96
    'c';l;'m';selN 'B';l;97
   ```

4. Local Heartbeat

   (a) A device receives a heartbeat from an address it has not heard from before:
   `'h';l;1234 or 'H';l;1234`

   (b) It reacts by (besides broadcasting its own HeartBeatAck) sending the address a HBInfoRequest:
   `'i';l;`

   (c) The unknown device responds with its DeviceID in a HBInfoReply and a version-node (assuming DiscoverySelector="1" , wire-version ="W2" , and discovery-version="D1") :
   `'I';l;<DeviceID>;1 'v';l;W2;D1`

5. Single-shot message

   `'S';l;selR;selS 'd';l;<DATA>`

6. Discovery heartbeat from remote Node via Router

(a) A local device receives a heartbeat by broadcast via the Router, originating from the remote device. The router has mark id 'abc' and uses `'<R>'` to identify the mark type as a router mark:

    `'H';l;1234 's';l;ShortID 'M';l;<R>;abc`

(b) The local device reacts by sending the address a HBInfoRequest as the remote device is previously unknown:

    `'i';l; 'r';l;ShortID`

(c) The Router sends the message on to the remote device and routes the response back to the local device. The response contains DeviceID in a HBInfoReply and a version-node from the remote device:

    `'I';l;<DeviceID>;1 'v';l;W2;D1 's';l;ShortID 'M';l;<R>;abc`

7. Message to a remote Palcom Node (DeviceR with ShortID) via a Router:

   `'c';l;'m';selN 'r';l;ShortID 'M';l;<R>;abc 'd';l;<DATA>`

Appendix E

Discovery Protocol: Concrete Message Representation

The messages sent over the network in the discovery protocol are considered included in the following context:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
```

Whenever parsing the content of a discovery message using an XML parser, the message is to be handled as preceded by this line.

For the concrete XML representation of messages, consider the following grammar for PRDServiceFMDescriptionReply:

```
PRDServiceFMDescriptionReply ::= PRDServiceFMDescription
PRDServiceFMDescription ::= LocalSID Help
(GroupInfo | CommandInfo | StreamInfo)*
GroupInfo ::= Help (CommandInfo | GroupInfo)*
CommandInfo ::= ID Direction Name Help (ParamInfo)*
ParamInfo ::= ID Type Name Help
```

The concrete XML representation for an example PRDServiceFMDescriptionReply is like this:

```
<SDA>
  <SD ls="134" h="This is a service">
    <GI h="This is a group">
      <CmdI id="c1" d="I" n="Do it" h="This is a command">
        <PI t="text/plain" id="p1" n="Param1" h="This is a param"/>
      </CmdI>
    </GI>
  </SD>
</SDA>
```

This example PRDServiceFMDescription contains one group, which contains one command with one parameter. There are help texts for the group, the command and the parameter. The LocalSID of the PRDServiceFMDescription is ”134”. The Command has ID ”c1”, direction in and the name ”Do it”. The parameter has the type ”text/plain”, ID ”p1” and the name ”Param1”.

Message/element	*XML tag name*	*Field*	*XML attr*	*Child elements*
PRDDeviceRequest	DQ[1]	-	-	-
PRDDeviceReply	DA	ServiceCache	svc	PRDDevice
		ConnectionCache	cc	
		AssemblyCache	ac	
		SelectorCache	slc	
		ServiceStatusCache	ssc	
PRDDevice	D	Name	n	-
		DeviceID	did	
		DeviceVer	v	
		DeviceStatus	ds	
		DiscoverySelector	dis	
		RemoteConnectSelector	rcs	
PRDServiceListRequest	SLQ	Parent	p	-
PRDServiceListReply	SLA	CacheNo	c	PRDServiceList
PRDServiceList	SL	DeviceID	did	ServiceInfo,
		Parent	p	SubListInfo
PRDService	S	LocalSID	ls	-
		Name	n	
		Distribution	d	
		HasDesc	hd	
		RemoteConnect	rc	
		Protocol	p	
		Reliable	r	
		ReadableVersionName	vn	
		ServiceHelpText	h	
PRDSubList	SubL	ReadableName	n	-
		Kind	k	
		ListLocalSID	ls	

Table E.1: XML tag and attribute names. Continued in Table E.2.

The XML tag and attribute names for PalCom discovery messages have been chosen as abbreviations of the message names and fields in the grammar, in order to be compact when transferred over the network.

The XML structure for all messages follows the pattern in this PRDServiceFMDescription example. There are XML elements for the messages, and for other non-terminals in the grammar, and XML attributes for simple fields in the grammar. The exception to this rule is LocalServiceID, which is written as one XML attribute, even though it is structured into two parts (for compactness). If the optional DeviceID part is present, it is separated from the ID part by a '/' character.

Tables E.1 and E.2 list all messages as defined in this document, with their XML tag and attribute names. The HeartBeat message is formatted at the Wire protocol level, and has no XML representation.

Message/element	*XML tag name*	*Field*	*XML attr*	*Child elements*
ServiceInstanceIDRequest	SIIQ	LocalSID	ls	-
ServiceInstanceIDReply	SIIA	LocalSID	ls	ServiceInstanceID
LocalServiceIDRequest	LSIQ	-	-	ServiceInstanceID
LocalServiceIDReply	LSIA	LocalSID	ls	ServiceInstanceID
SelectorRequest	SelQ	LocalSID	ls	-
SelectorReply	SelA	LocalSID Selector	ls s	-
ServiceStatusRequest	SSQ	LocalSID	ls	-
ServiceStatusReply	SSA	LocalSID ServiceStatus StatusHelpText	ls ss h	-
PRDConnectionListRequest	CLQ	-	-	-
PRDConnectionListReply	CLA	-	-	PRDConnection-List
PRDConnectionList	CL	DeviceID	did	PRDConnection
PRDConnection	C	ProviderDeviceID ProviderServiceID CustomerDeviceID CustomerServiceID	pdid psid cdid csid	-
ServiceID	SID	CreatingDeviceID CreationNumber UpdatingDeviceID UpdateNumber PreviousDeviceID PreviousNumber	cdid cn udid un pdid pn	-
ServiceInstanceID	SIID	DeviceID InstanceNumber	did in	ServiceID
PRDServiceFMDescription-Request	SDQ	LocalSID	ls	-
PRDServiceFMDescription-Reply	SDA	-	-	PRDServiceFM-Description
PRDServiceFMDescription	SD	Help LocalSID	h ls	GroupInfo, CommandInfo, StreamInfo
GroupInfo	GI	Help	h	CommandInfo, GroupInfo
CommandInfo	CmdI	ID Direction Name Help	id d n h	ParamInfo
ParamInfo	PI	ID Type Name Help	id t n h	-
StreamInfo	StrI	Direction StreamType	d t	-

Table E.2: XML tag and attribute names. Continued from Table E.1

Bibliography

[1] Sten L. Amundsen and Frank Eliassen. A resource and context model for mobile middleware. *Personal and Ubiquitous Computing*, 2006. published online first.

[2] J. R. Andersen, L. Bak, S. Grarup, K. V. Lund, T. Eskildsen, K. M. Hansen, and M. Torgersen. Design, implementation, and evaluation of the resilient SmallTalk embedded platform. In *Computer Languages, Systems & Structures*, volume 31, pages 127–141, 2005. `http://dx.doi.org/10.1016/j.cl.2005.02.003`.

[3] J.R. Andersen, L. Bak, S. Garup, K. V. Lund, T. Eskildsen, K. M. Hansen, and M. Torgersen. Design, implementation and evaluation of the resilient smalltalk embeded platform. `http://www.esug.org/data/ESUG2004/ESUG2004-RT-Resilient.pdf`.

[4] Peter Andersen, Jakob Eyvind Bardram, Henrik Brbak Christensen, Aino Vonge Corry, Dominic Greenwood, Klaus Marius Hansen, and Reiner Schmid. Open architecture for palpable computing: Some thoughts on object technology, palpable computing, and architectures for ambient computing. In *ECOOP 2005 Object Technology for Ambient Intelligence Workshop*, Glasgow, U.K., 2005. `http://www.ist-palcom.org/publications/files/OT4AMI-2005.pdf`.

[5] Anupriya Ankolekar, Mark Burstein, Jerry R. Hobbs, Ora Lassila, David Martin, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry Payne, and Katia Sycara. DAML-S: Web service description for the Semantic Web. In I. Horrocks and J. Hendler, editors, *Proceedings of the First International Semantic Web Conference (ISWC 2002)*, volume 2342 of *LNCS*, pages 348–363, Sardinia, Italy, June 2002. Springer-Verlag, Berlin Heidelberg.

[6] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, third edition, 2000.

[7] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practise*. Addison-Wesley, 2nd edition, 2003.

[8] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[9] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999. `http://citeseer.ist.psu.edu/birman98bimodal.html`.

[10] Dipanjan Chakraborty, Anupam Joshi, Tim Finin, and Yelena Yesha. Service composition for mobile environment. *Mobile Networks and Applications*, 4(10):435–451, August 2005.

[11] W. Keith Edwards, Mark W. Newman, Jana Sedivy, and Shahram Izadi. Challenge: recombinant computing and the speakeasy approach. In *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 279–286, New York, USA, 2002. ACM Press.

[12] W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy, and Trevor F. Smith. Bringing network effects to pervasive spaces. *IEEE Pervasive Computing*, 4(3):15–17, 2005.

[13] Edwards, K., et al. Using speakeasy for ad hoc peer-to-peer collaboration. In *CSCW '02: Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 256–265. ACM Press, 2002.

[14] Keita Fujii and Tatsuya Suda. Dynamic service composition using semantic information. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, New York, New York, U.S.A, November 2004. ACM.

[15] D. Garlan, DP Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: toward distraction-free pervasive computing. *Pervasive Computing, IEEE*, 1(2):22–31, 2002.

[16] R. Grimm. One. world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[17] 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.

[18] Greg Haerr. The nano-x window system, 2005. `http://www.microwindows.org/`.

[19] Greg Haerr. The nano-x window system faq, 2005. `http://www.microwindows.org/faq.html`.

[20] The java hotspot performance enging architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[21] M. Ingstrup. *Towards distributed declarative architectural reflection.* PhD thesis, University of Aarhus, 2006.

[22] Mads Ingstrup and Klaus Marius Hansen. Palpable assemblies: Dynamic service composition for ubiquitous computing. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, pages 632–638, Tapei, Taiwan, 2005. `http://www.ist-palcom.org/publications/files/PalpableAssemblies-SEKE05.pdf`.

[23] ISO/IEC. Software engineering—product quality, part 1–4, 2001. ISO-9126-1,-2,,3,-4.

[24] 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+, 2004.

[25] Swaroop Kalasapur, Mohan Kumar, and Behrooz A. Shirazi. Dynamic service composition in pervasive computing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(7):907–918, 2007.

[26] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management.* Wiley, 2004.

[27] Margit Kristensen, Morten Kyng, and Esben Toftdahl Nielsen. It support for healthcare professionals acting in major incidents. In *Proceedings of the 3rd Scandinavian conference on Health Informatics*, pages 37–41, Aalborg, Denmark, 2005. `http://www.ist-palcom.org/publications/files/Major%20Incidents%20paper%20SHI2005-v1.pdf`.

[28] Emre Kýcýman and Armando Fox. Using dynamic mediation to integrate cots entities in a ubiquitous computing environment. In *Proceedings of HUC2000*, number 1927 in LNCS, pages 211–226, 2000.

[29] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[30] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. `http://java.sun.com/docs/books/vmspec/`.

[31] S. Maffioletti, MS Kouadri, and B. Hirsbrunner. Automatic resource and service management for ubiquitous computing environments. *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 219–223, 2004.

[32] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, et al. Bringing Semantics to Web Services: The OWL-S Approach. *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, pages 6–9, 2004.

[33] Ryusuke Masuoka, Yannis Labrou, Bijan Parsia, and Evren Sirin. Ontology-enabled pervasive computing applications. *IEEE Intelligent Systems*, 18(5):68–72, 2003.

[34] Ryusuke Masuoka, Bijan Parsia, and Yannis Labrou. Task computing – the semantic web meets pervasive computing. *The Semantic Web - ISWC 2003*, 2870/2003:866–881, 2003.

[35] Sun Microsystems. Sunspotworld - home of project sun spot, 2007. `http://www.sunspotworld.com`.

[36] R. Moats. URN Syntax. RFC 2141 (Proposed Standard), May 1997.

[37] S.B. Mokhtar, J. Liu, N. Georgantas, and V. Issarny. QoS-aware dynamic service composition in ambient intelligence environments. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 317–320. ACM Press New York, NY, USA, 2005.

[38] Mark W. Newman, Jana Z. Sedivy, Christine M. Neuwirth, W. Keith Edwards, Jason I. Hong, Shahram Izadi, Karen Marcelo, Trevor F. Smith, Jana Sedivy, and Mark Newman. Designing for serendipity: supporting end-user configuration of ubiquitous computing environments. In *DIS '02: Proceedings of the conference on Designing interactive systems*, pages 147–156, New York, NY, USA, 2002. ACM Press.

[39] Osvm object-oriented software platform. `http://www.esmertec.com/solutions/M2M/OSVM/`.

[40] PalCom. Palpable computing: A new perspective on ambient computing. annex i – description of work. Technical report, PalCom Project IST-002057, 2003. `http://www.ist-palcom.org/publications/review1/contract/PalComDoW-v5-0-Final.pdf`.

[41] PalCom. PalCom Open Source Dissemination Toolkit, 2007. `http://www.ist-palcom.org/download`.

[42] PalCom. Palpable computing: A new perspective on ambient computing. annex i – description of work, update jan. 2007. Technical report, PalCom Project IST-002057, 2007. `http://www.ist-palcom.org/publications/review3/contract/PalComDoW6.pdf`.

[43] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Tricky, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[44] S.R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. *Proceedings of Ubicomp*, 1, 2001.

[45] PalCom Project. Palcom external report 10: Deliverable 10 (3.1): Open issues in language design for palpable/ambient computing. Technical report, PalCom Project IST-002057, December 2004. `http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-10-[3.1]-Open-Issues-in-Language-Design-for-PalpableAmbient-Computing.pdf`.

[46] PalCom Project. Palcom external report 12: Deliverable 12 (10.1): Initial work analysis report. Technical report, PalCom Project IST-002057, December 2004. `http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-12-[10.1]-Initial-Work-Analysis-Report.pdf`.

[47] PalCom Project. Palcom external report 16: Deliverable 7 (7.1): Multimedia material. Technical report, PalCom Project IST-002057, December 2004. `http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-7-[7.1]-Multimedia-Material.pdf`.

[48] PalCom Project. Palcom external report 8: Deliverable 6 (2.1): Palcom open architecture overview. Technical report, PalCom Project IST-002057, December 2004. http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-6-[2.1]-Report-on-Architecture-Requirements-And-Design.pdf.

[49] PalCom Project. Palcom external report 9: Deliverable 11 (8.1): Exploratory prototypes. Technical report, PalCom Project IST-002057, December 2004. http://www.ist-palcom.org/publications/review1/deliverables/Deliverable-11-[8.1]-Exploratory-Prototypes.pdf.

[50] PalCom Project. Palcom external report 20: Deliverable 15 (5.1): Palcom component model. Technical report, PalCom Project IST-002057, March 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-15-[5.1]-PalCom-Component-Model.pdf.

[51] PalCom Project. Palcom external report 21: Deliverable 16 (6.1): Palpable mixed-media device prototypes. Technical report, PalCom Project IST-002057, March 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-16-[6.1]-Palpable-mixed-media-prototypes.pdf.

[52] PalCom Project. Palcom external report 22: Deliverable 17 (7.2): Mock-up and prototype scenarios. Technical report, PalCom Project IST-002057, March 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-17-[7.2]-Mock-up-and-Prototype-Scenarios.pdf.

[53] PalCom Project. Palcom external report 25: Deliverable 18 (8.2): Prototypes. Technical report, PalCom Project IST-002057, March 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-18-[8.2]-Prototypes.pdf.

[54] PalCom Project. Palcom external report 27: Deliverable 21 (2.4.1): Specification of programming models and language support for palpable computing. Technical report, PalCom Project IST-002057, August 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-21-[2.4.1]-programming-models-and-language-support.pdf.

[55] PalCom Project. Palcom external report 28: Deliverable 22 (2.3.1) specification of virtual machine and reference implementation on selected embedded device(s). description of use of vm in application prototypes. Technical report, PalCom Project IST-002057, October 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-22-[2.3.1]-virtual-machine.pdf.

[56] PalCom Project. Palcom external report 29: Deliverable 23 (2.4.2): Specification of component & communication model. Technical report, PalCom Project IST-002057, October 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-23-[2.4.2]-component-communication-model.pdf.

[57] PalCom Project. Palcom external report 31: Deliverable 32 (2.2.1): Palcom open architecture - first complete version of basic architecture. Technical report, PalCom Project IST-002057, December 2005. http://www.ist-palcom.org/publications/deliverables/Deliverable-32-[2.2.1]-palcom-open-architecture.pdf.

[58] PalCom Project. Palcom external report 32: Deliverable 24 (2.5.1): Design issues in resource management. Technical report, PalCom Project IST-002057, October 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-24-[2.5.1]-resource-awareness-and-management.pdf.

[59] PalCom Project. Palcom external report 34: Deliverable 25 (2.6.1): End-user composition tool. Technical report, PalCom Project IST-002057, October 2005. http://www.ist-palcom.org/publications/review2/deliverables/Deliverable-25-[2.6.1]-end-user-composition-tool.pdf.

[60] PalCom Project. Palcom external report 45: Deliverable 35 (2.5.2): Basic contingency management model and prototype demonstrating ra & cm mechanisms. Technical report, PalCom Project IST-002057, April 2006. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-35-[2.5.2]-basic-contingency-management-model.pdf.

[61] PalCom Project. Palcom external report 50: Deliverable 39 (2.2.2): Palcom open architecture. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/deliverables/Deliverable-39-[2.2.2]-Palcom-Open-Architecture.pdf.

[62] PalCom Project. Palcom external report 52: Deliverable 37 (2.1.2): Palpability: Revised conceptual framework for palpable computing. Technical report, PalCom Project IST-002057, November 2006. http://www.ist-palcom.org/publications/deliverables/Deliverable-37-[2.1.2]-palpability-revised-SectionI.pdf.

[63] PalCom Project. Palcom external report 54: Deliverable 40 (2.3.2): Runtime environment. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/deliverables/Deliverable-40-[2.3.2]-runtime-environment.pdf.

[64] PalCom Project. Palcom external report 55: Deliverable 41 (2.4.3): Components & communication. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/deliverables/Deliverable-41-[2.4.3]-components-and-communication.pdf.

[65] PalCom Project. Palcom external report 56: Deliverable 42 (2.5.3): Resource & contingency management. Technical report, PalCom Project IST-002057, December 2006. http://www.ist-palcom.org/publications/review3/deliverables/Deliverable-42-[2.5.3]-resource-contingency-management.pdf.

[66] PalCom Project. Palcom external report 57: Deliverable 43 (2.6.2): End user composition. Technical report, PalCom Project IST-002057, January 2007. http://www.ist-palcom.org/publications/deliverables/Deliverable-43-[2.6.2]-EndUserComposition.pdf.

[67] PalCom Project. Palcom external report 62: Deliverable 47 (2.14.4): Dissemination. Technical report, PalCom Project IST-002057, September 2007. http://www.ist-palcom.org/publications/deliverables/Deliverable-47-[2.14.4]-dissemination.pdf.

[68] PalCom Project. Palcom external report 64: Deliverable 49 (2.6.3): End user composition. Technical report, PalCom Project IST-002057, November 2007. http://www.ist-palcom.org/publications/deliverables/Deliverable-49-[2.6.3]-end-user-composition.pdf.

[69] PalCom Project. Palcom external report 65: Deliverable 50 (2.7.3): Application prototypes final versions. Technical report, PalCom Project IST-002057, November 2007. http://www.ist-palcom.org/publications/deliverables/Deliverable-50-[2.7.3]-prototypes-final.pdf.

[70] PalCom Project. Palcom external report 68: Deliverable 53 (2.1.3): Palpability. Technical report, PalCom Project IST-002057, December 2007. http://www.ist-palcom.org/publications/deliverables/Deliverable-53-[2.1.3]-palpability.pdf.

[71] PalCom Project. Palcom external report 70: Developer's companion. Technical report, PalCom Project IST-002057, September 2007. http://svn.ist-palcom.org/svn/palcom/trunk/doc/tex/companion.pdf.

[72] Manuel Román, Brian Ziebart, and Roy H. Campbell. Dynamic application composition: Customizing the behavior of an active space. In *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, page 169, Washington, DC, USA, 2003. IEEE Computer Society.

[73] Ulrik Pagh Schultz, Erik Corry, and Kasper V. Lund. Virtual machines for ambient computing: A palpable computing perspective. In *ECOOP 2005 Object Technology for Ambient Intelligence Workshop*, Glasgow, U.K., 2005. `http://www.ist-palcom.org/publications/files/OT4AMI-prevm.pdf`.

[74] J. P. Sousa, V. Poladian, D. Garlan, B. Schmerl, and M. Shaw. Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics—Part C: Applications and Reviews*, 36(3):328–340, 2006.

[75] David Svensson, Görel Hedin, and Boris Magnusson. Pervasive applications through scripted assemblies of services. In *Proceedings of 1st International Workshop on Software Engineering of Pervasive Services*, 2006.

[76] Lund University. Jastadd open source java-based compiler compiler system. `http://jastadd.org/`.

[77] M. Vallee, F. Ramparany, and L. Vercouter. Flexible Composition of Smart Device Services. In *The 2005 International Conference on Pervasive Systems and Computing (PSC-05), Las Vegas, Nevada, USA., June*, pages 27–30, 2005.

[78] Yuping Yang, Fiona Mahon, M. Howard Williams, and Tom Pfeifer. Context-aware dynamic personalised service re-composition in a pervasive service environment. In *Proceedings of Ubiquitous Intelligence and Computing*, volume 4159 of *LNCS*, pages 724–735. Springer, 2006.

[79] Peter Ørbæk. Programming with hierarchical maps. Technical report, University of Aarhus, Daimi PB-575, 2005. `http://www.ist-palcom.org/publications/files/Palcom-Hierarchical-Maps.pdf`.