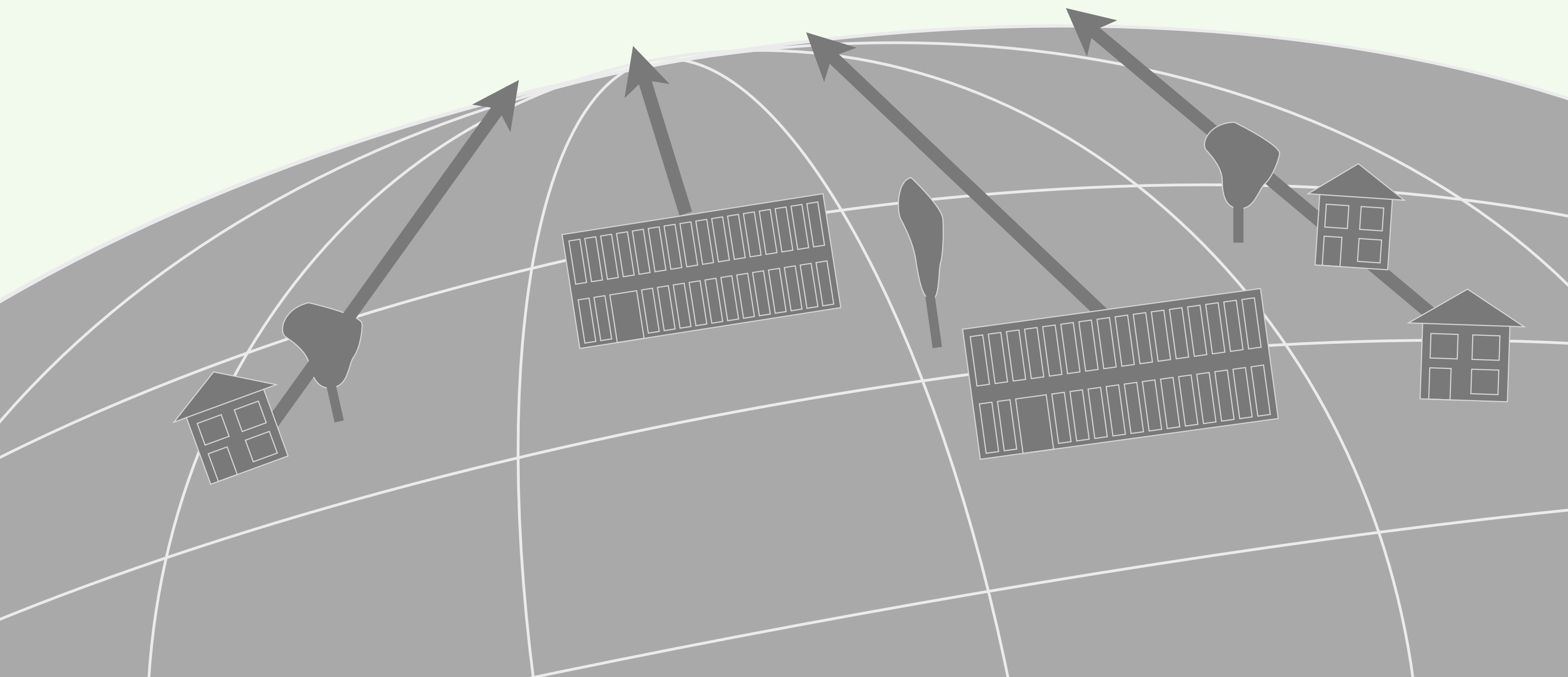
Distributed Smart Space Orchestration

Marc-Oliver Pahl



TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste

Distributed Smart Space Orchestration

Marc-Oliver Pahl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. phil. nat. Helmut Seidl
Prüfer der Dissertation:	1.	Univ.-Prof. Dr.-Ing. Georg Carle
	2.	Univ.-Prof. Gudrun J. Klinker, Ph.D.

Die Dissertation wurde am 11.3.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29.5.2014 angenommen.

Thesis Abstract – Marc-Oliver Pahl

Distributed Smart Space Orchestration

Spaces in which humans live are increasingly enriched by sensors and actuators. Examples are heating, ventilation, air-conditioning, lighting, or access control. Many of the devices involved are equipped with network interfaces. This ability for remote control gives the possibility to connect and orchestrate all networked devices in a space. However, many of the available interfaces form functional, vendor, or spatial silos not allowing for full interconnectivity, thereby preventing comprehensive orchestration of smart spaces.

A fundamental open challenge of pervasive computing research is to design a suitable programming abstraction that structures and facilitates the development of pervasive computing application at large, and that can be used in the real world. By analyzing different research areas, the thesis identifies requirements that have to be fulfilled by a suitable programming abstraction. A state of the art analysis shows the missing of a suitable concept.

A novel middleware paradigm, called μ-middleware is introduced. It provides the functionality of a context-provisioning middleware, and it enables direct coupling of so-called smart space services. The resulting abstraction is called Virtual State Layer (VSL).

The VSL uses context models as abstract interfaces to services. To facilitate the creation of context models a hybrid meta model is introduced. Context models are shared over a so-called Context Model Repository (CMR). The VSL and all of its services use a fixed interface of 13 methods. This implements interface compatibility between all services in a smart space. The ability of inter service communication, the sharing of abstract service interfaces, and the use of fixed methods to access services implement a service oriented architecture.

A major challenge in the above described scenario is overcoming the silos. So-called portability is implemented by introducing a crowdsourced, self-managing convergence mechanism for context models in the CMR. The described design implements a dynamically extensible, self-managing collaborative ontology for smart spaces.

To demonstrate the simplicity and the power of the introduced VSL programming abstraction, a pendant to the App economy for smartphones is introduced. The use of the VSL as basis for the solution allows the implementation of a security-by-design concept.

Foundation services to connect devices with the VSL, and to implement diverse pervasive computing scenarios are introduced. A quantitative analysis shows the scalability of the presented solution. A qualitative user study shows the real world usability of the solution.

The presented concepts complement the hardware maker culture that is emerging in 2014 with tools that enable the emergence of a software maker culture for smart spaces. With a real world emergence of such a culture, ubiquitous computing would finally become reality.

Preface

This document marks the end of my structured academic education. All people I met on my journey so far had their influence on me and therefore on this work. I want to use the following paragraphs to thank some of those who had the most direct influence on my PhD explicitly, and all others implicitly. I know your importance for me, independent of explicitly naming you here or not. Thank you all.

I am deeply thankful to my family, especially my parents Carmen and Wolfgang, and my brothers Dennis and Philipp. Your constant support enabled me to overcome all obstacles in the past. Thank you for always being there for me, and for supporting me in all steps I take. Thank you for your unconditional love. Thank you Wolfgang and Dennis for proof reading so much text within such a short time!

Thank you to all my friends for constantly supporting me. Each of you is very special for me. You bring sunshine in my life. Thank you for tolerating my "absence" and providing me with so many joyful moments. Thank you, Tobias Heer, Johannes Nübler, and Christian Michel for managing to finally push me into the necessary writing mode.

All my colleagues, as I consider you as friends, feel included in the last paragraph. Though none of you did research in the area I did, I learned a lot from you by spending so many hours with you every day. A special thank to you, Manfred Jobmann, for your many valuable comments on this final version. Implementing them definitely made this document better understandable. Thank you all.

Thank you, Georg Carle. Without you this document would not exist. Thank you for always trusting me and giving me the resulting freedom and resources to develop things my way. Thank you for directly and indirectly teaching me so many things – related and unrelated to the academic system. Thank you for the many hours of discussions we spent on various topics. Thank you for providing me with a working environment that enabled me to do my research as documented in this book, and so many more projects at university. Thank you for letting me create the iLab.

Thank you, Gudrun Klinker, for your motivation, our fruitful discussions, and your kindness. Thank you, Helmut Seidl, for our exchanges in different contexts that I always enjoy. Thank you, Eike Jessen, for our joyful exchange, and for participating at my defense – your wonderful compliments make me very happy. Thank you Ernst Biersack for the nice chats, and for showing me a bit of your way of doing research.

Thank you, Rainer Nagel, for showing me your utopian facets of university. Your way encourages me to do things differently though it is often quite hard. Working differently often provokes opposition. Thank you, Ursula and Rainer Nagel, for making me a part of your life. I greatly enjoy the time with you and your extended family.

Thank you, Manfred Muckenhaupt, for showing me that following one's goals even at a possibly higher price can be the right way. Thank you for introducing me to a more reflected interaction with media. Though your "media science" turned out to be very different from what I expected to learn, I enjoyed and enjoy it a lot as it opened new perspectives to me.

Thank you, Michael Sperber, for your passion for teaching and computer science. I learned a lot from you, including that code must look good on t-shirts ;) You are a great inspiration.

Thank you Klaus-Jörn Lange for giving me my first student assistant position in 2001 that made me even more aware of my passion for teaching. Thank you, all participants of my two first classes. Several of you are close friends today. Thank you Wolfgang Küchlin for giving me the opportunity to hold my first real lecture in 2003. Thank you, my first students for showing your appreciation. You encouraged me a lot, and I enjoyed working with you a lot. Thank you, Andreas Schilling, Sieghart Stangler, and Michael Diehl for the projects we implemented at Universität Tübingen, and for the nice discussions that we had, which I always enjoyed. Thank you Herbert Klaeren, Wolfgang Straßer, Wolfgang Rosenstiel, and Uwe Baumgarten for our always-joyful exchange and your support.

Though my presence at conferences started very late, I had the opportunity to get in contact with some very nice people of whom I especially want to thank Karoly Farkas, Sajal Das, Christian Becker, Gregory Abowd, Roy Want, Mark Newman, Guy Pujolle, Ulf Körner, and Hubert Zimmermann for the very kind discussions we had. I enjoyed talking to each of you a lot. I hope we can continue exchanging, and I will get to know many more inspiring people from academia in the upcoming years.

Thank you, all my students I had the pleasure working with and learning a lot from. A special thank to those of you, I supervised the final thesis and projects of, which are in temporal order Frederik Fischer, Jose Valerio, Alexandro Rodriguez, Blaz Primc, Malte Buck, Robert Franz, Enrique Garcia, Carl Denis, Rene Brogatzki, Martin Riedl, Sibi Antony, Daniel Baumann, Pavel Podkopajev, Cuneyt Caliskan, Thomas Szyrkoviec, Nakul Chaudhari, Johann Dichtl, and Milad Khanibeik. Especially thank you, Deniz Ugurlu, for implementing the VSL knowledge agents with me. We both learned a lot.

Though not related to this document but strongly related to my PhD time, thank you to all contributors to my big teaching project, the iLab. Thanky you, Uwe Bilger, for starting the fire. Thank you first class of 2004 and all following classes for allowing me to learn so much about teaching. Thank you Johannes Riedl, Andreas Korsten, Joachim Schiele, Jakob Dettner, Tobias Beck, Stephan Günther, and more recently Benjamin Hof, Lukas Schwaighofer, and Florian Wohlfart for carrying the spirit further. Thank you, all tutors over all those years. Thank you, more than 1500 participants since 2004. It was a pleasure working with each single one of you. I enjoy a lot still having contact to several of you!

Last but not least thanks to "my" sports crew, especially Lothar, Maja, Gerhard, Andi, Konrad, and Caro. Without you I would not have survived this time ;)

München 2014, [illegible]

Contents

List of Figures

List of Tables

1. Introduction

Simplicity does not precede complexity, but follows it.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

In 1991, Mark Weiser formulated the vision of "ubiquitous computing" [Wei91]. In his vision, computing is *woven into everyday's fabric*, meaning that computing is possible nearly everywhere, and part of many aspects of life. Ubiquitous computing can be divided into the evolutionary steps *distributed computing*, *mobile computing*, and *pervasive computing* (Ch. 2).

Distributed computing and mobile computing are reality today: cloud computing is a recent manifestation of the distributed computing paradigm (Sec. 4.3). The use of smartphones is daily routine for many people in many countries in 2014 (Sec. 2.3). Smartphones allow mobile computation (Apps, Sec. 3.10), and mobile access to the Internet (Sec. 2.3.1).

Pervasive computing is only in parts reality in 2014. Smart Devices that allow sensing and actuating (parts of) the real world remotely via software are available off-the-shelf (Sec. 3.2). However, the systems are often isolated in silos which prevents using them for implementing pervasive computing scenarios (Sec. 3.3.1).

A fundamental open research question (Sec. 2.5.1) that has direct impact on the practical implementation of pervasive computing in the real world is: *how should a –so far missing– programming abstraction be designed to enable the implementation of pervasive computing scenarios in the real world.*

This thesis proposes an answer to the question, and shows how the proposed programming abstraction can be used to implement the technical foundation for an App economy for Smart Spaces.

A goal of this thesis is enabling the shift from "*programming environments to programming environments*" [Abo12], shifting the complexity from the technical aspects of implementing pervasive computing scenarios towards the logic behind the scenarios.

1.1 Objectives of the Thesis

The challenge described before can be reformulated to, *providing an abstraction that is so simple-to-use and at the same time so powerful that it allows developers to focus on solving their pervasive computing problems instead of having to deal with technical problems.*

To reach this goal, the overall objective of this thesis is to

<O.0> Design a programming abstraction for Smart Spaces that *structures and facilitates the development* of pervasive computing applications *at large*, and that can be *used in the real world.*

Fundamental tasks for implementing pervasive computing scenarios with software are monitoring sensors, and controlling actuators according to workflows that implement high-level goals. This allows splitting the overall objective <O.0> into four subsidiary objectives.

To implement pervasive computing scenarios with software (Sec. 2.4.1), it is necessary to provide software interfaces to the Smart Devices that can be found in Smart Spaces. Services that provide such interfaces are called *adaptation services* in this thesis. The first subsidiary goal for the programming abstraction is to

<O.1> Structure and facilitate the development of so-called *adaptation services.*

The services that implement the logic of pervasive computing scenarios are called *orchestration services* in this thesis (Sec. 2.4.1), leading to the second goal for the programming abstraction, to

<O.2> Structure and facilitate the development of *orchestration services.*

As Sec. 4.5 shows (Sec. 3.13.2), solutions for the objectives <O.1> and <O.2> exist in the research community, and outside research labs in 2014 (e.g. professional Building Automation Systems (BASs), Sec. 3.2.1). However, they fail (Ch. 4) to solve the remaining parts of <O.0>.

Pervasive computing aims to pervade many aspects of daily life with computing (Sec. 2.1). This involves *diverse* scenarios. The attribute "at large" of <O.0> expresses that the programming abstraction should

<O.3> Support *diverse* pervasive computing scenarios and not only few use cases.

Finally, pervasive computing needs to be deployed in the real world to become reality outside research labs and prototypes. The goal, *usability in the real world*, in the overall objective <O.0> can be specified as aim to

<O.4> Design a programming abstraction that can be *used by regular users* for implementing pervasive computing in the(ir) real world, and not only by researchers in laboratories. This includes empowering users to develop their own programs that implement their desired pervasive computing scenarios (Sec. 2.5.2).

1.2 Methodology

The thesis starts with an analysis of the implementation of ubiquitous computing in 2014 (Ch. 2). It **identifies** the *missing of a suitable programming abstraction* for pervasive computing (<O.0>).

For specifying the attribute "*suitable*", **requirements are analyzed** in Ch. 3. The result are 50 requirements that have to be solved to reach the objectives of the thesis.

To confirm the relevance of the identified requirements, and to ensure that no solution exists that solves the requirements, relevant **state of the art** is analyzed in Ch. 4. The analysis shows that other researchers, and the commercial solutions that are on the market consider the requirements relevant. The state of the art solves only parts of the requirements. It typically focuses on aspects that belong to the adaptation of Smart Devices (<O.1>), and to the implementation of orchestration workflows (<O.2>).

Then the VSL programming abstraction is **designed** in Ch. 5. A mapping of requirements and solutions is directly provided via markers in the text, and at the side of the text. The programming abstraction solves most of the requirements. Some requirements are implementations specific and therefore covered in Ch. 6, or do not have to be solved by the programming abstraction as they can be solved using the programming abstraction (Ch. 7).

To show the feasibility of the concept, an **implementation** of the programming abstraction as autonomous Peer-to-Peer (P2P) system is provided in Ch. 6. With this chapter all requirements but the provisioning of an App store are fulfilled.

The usability of the implementation of the programming abstraction, and the feasibility of pervasive computing in the real world with the provided components are **demonstrated** in Ch. 7. With the DS2OS, the technically necessary components for implementing an App economy for Smart Spaces are introduced. The implementation of DS2OS demonstrates how the VSL programming abstraction structures and facilitates the development of pervasive computing services. In addition, DS2OS introduces the technical infrastructure that is needed for a *real world deployment* of the solution. All identified requirements are fulfilled at this point of the thesis. Another comparison to the state of the art highlights the differences between the new approach and the existing work.

To **demonstrate** the *real world usability* of the proposed approach, implementations for services that are fundamental for implementing pervasive computing scenarios are introduced in Ch. 8. The chapter demonstrates how the VSL programming abstraction facilitates the implementation of representative software components.

For an independent confirmation of the *real world usability*, a **quantitative, and a qualitative evaluation** of the solution are presented in Ch. 9. The quantitative **measurement in a testbed** assesses the suitability of the programming abstraction for implementing time critical scenarios. This assessment is relevant for the practical real world usability of the presented solution. The scalability is assessed as it is relevant for a *real world use* of the presented solution in large scenarios. A **user study** is conducted to confirm the *suitability* of the VSL programming abstraction for enabling users to implement their own pervasive computing scenarios in software.

Fig. 1.1 shows the structure of the thesis.

1.3 Major Contributions

This thesis has several contributions. The most important ones are listed at the beginning of each chapter where they are introduced.

The major contribution of this thesis is a novel programming abstraction for pervasive computing. It is called *Virtual State Layer (VSL)*. It comprises several novelties as described in Ch. 5, and summarized in the abstract at the beginning of that chapter. The most important concepts of the VSL are

- *the use of context models as abstract service interfaces* (Sec. 5.4.1),
- *Virtual Context* (Sec. 5.4.2),
- a *dynamically extensible, self-managing collaborative ontology* (Sec. 5.2.18) that is extended to a *crowdsourced ontology* in Ch. 7 (Sec. 7.4),
- the *separation of service logic and service state* (Sec. 5.6.5), and
- a *hybrid meta model* that is used for context modeling in the VSL (Sec. 5.2).

The use of *context models as abstract service interfaces* structures and facilitates the creation and composition of Smart Space services.

Virtual Context enables direct coupling of services over a descriptive (context) interface. The introduction of this novel concept enables the transparent dynamic functional extension of a middleware that implements the VSL programming abstraction.

The *collaborative ontology* implements a crowdsourced standardization of abstract service interfaces. Such standardization is required for providing *interface portability.* Interface portability enables services to run in different Smart Spaces and not only in the Smart Space, they were designed for as it is typically the case in 2014.

The *separation of service logic and service state* structures and facilitates services. It reduces the code size as common functionality can be provided by the VSL and does not have to be implemented in services. It allows focusing on the service logic instead of implementation details (Sec. 1).

The *hybrid meta model* enables user-based collaborative creation of the VSL ontology. It combines simplicity, expressiveness, and performant processing.

The major side-contributions of this thesis are listed in the following. They are ordered by their relevance for implementing pervasive computing in the real world. All contributions are novel.

- a *novel middleware paradigm* that is called μ-middleware and enables dynamic transparent extension with “core functionality” (Sec. 6.2),
- a *security concept* for software orchestrated Smart Spaces (Sec. 2.4.1) that enables the provisioning of security-by-design (Sec. 5.5, Sec. 6.5, Sec. 7.3),
- the design of the technical components that enable the impleentation of an *App economy for Smart Spaces* (Ch. 7),
- a concept for *automated integration of Smart Devices* into software orchestrated Smart Spaces (Sec. 8.3.1),

- a concept for *autonomous management of context on different levels of abstraction* (Sec. 8.3.2),
- a *coupling mechanism between distributed VSL operated Smart Spaces* that gives local access control for functionality that is transparently available in a remote site (Sec. 8.3.4),
- a *generic user interface that can be used for accessing all functionality* of a VSL Smart Space (Sec. 8.3.5),
- a set of *building blocks that enable the implementation of complex pervasive computing scenarios with simple Event-Condition-Action (ECA) rules* (Sec. 8.4), and
- *teaching material* that teaches students about Smart Space Software Orchestration (Ch. 9).
- A requirements analysis for a suitable programming abstraction for future Smart Spaces (Ch. 3).
- A survey of existing state of the art pervasive computing middleware that exceeds existing surveys (table 4.1).
- A combined reference architecture for existing pervasive computing middleware (Fig. 4.3).

1.3.1 Chapter Summaries

Fig. 1.1 shows the structure of the thesis that is briefly described in this section. Different to the description of the methodology in Sec. 1.2 this section focuses primarily on the content of the chapters and not their purpose. As described in Sec. 1.4, each chapter has a short and a longer summary on its first page. Following the short summaries are printed.

Chapter 2 (Ubiquitous Computing) puts this work into the context of ubiquitous computing, and provides a high-level analysis of the implementation of its parts in 2014. It is shown that ubiquitous computing research comprises distributed computing, mobile computing and pervasive computing. While the distributed computing and mobile computing parts of the ubiquitous computing vision are reality in 2014, pervasive computing is only reality in limited domains, or research prototypes.

Heterogeneity of Smart Devices, and the missing of a suitable programming abstraction are identified as inhibitors for a implementation of pervasive computing in the real world.

It is discussed that user-participation is desired for the development of services that implement pervasive computing scenarios.

The terms Smart Space, Smart Device, Smart Space Orchestration, and context are defined. As pervasive computing comprises several sub domains, this work is put into context to Internet of Things (IoT), Cyber-Physical System (CPS), and Ambient Intelligence (AmI). Context-aware computing is introduced as basic paradigm behind pervasive computing.

Chapter 3 (Analysis) analyzes requirements on implementing Smart Space Orchestration in the real world. Knowing the requirements is relevant for the design of a *suitable* programming abstraction as most requirements have to be supported by the abstraction directly (Ch. 5).

In a bottom-up, and a top-down analysis the problem space is structured. Bottom-up, Smart Devices from different domains, solutions for bridging heterogeneity, middleware, and μ-kernel operating system design are analyzed. Top-down, psychology, AI, autonomic management, Service Oriented Architecture (SOA), and context modeling are analyzed. The App economy is analyzed as a successful example of the recent implementation of a subdomain of ubiquitous computing in the real world.

The analyzed fields give background for the design of the VSL programming abstraction in Ch. 5ff.

The 50 identified requirements are summarized and discussed in Sec. 3.11ff.

Chapter 4 (State of the Art) starts with an analysis of distributed user-based software development, virtualization as basic paradigm of cloud computing, network management, and building management.

The main part of this chapter analyzes relevant state of the art pervasive computing middleware. The selection is based on different surveys. The assessment criteria are also based on several surveys and extended with the requirements from Ch. 3.

Chapter 5 (The Virtual State Layer Programming Abstraction) introduces the VSL programming abstraction. It is based on the context-provisioning middleware concept.

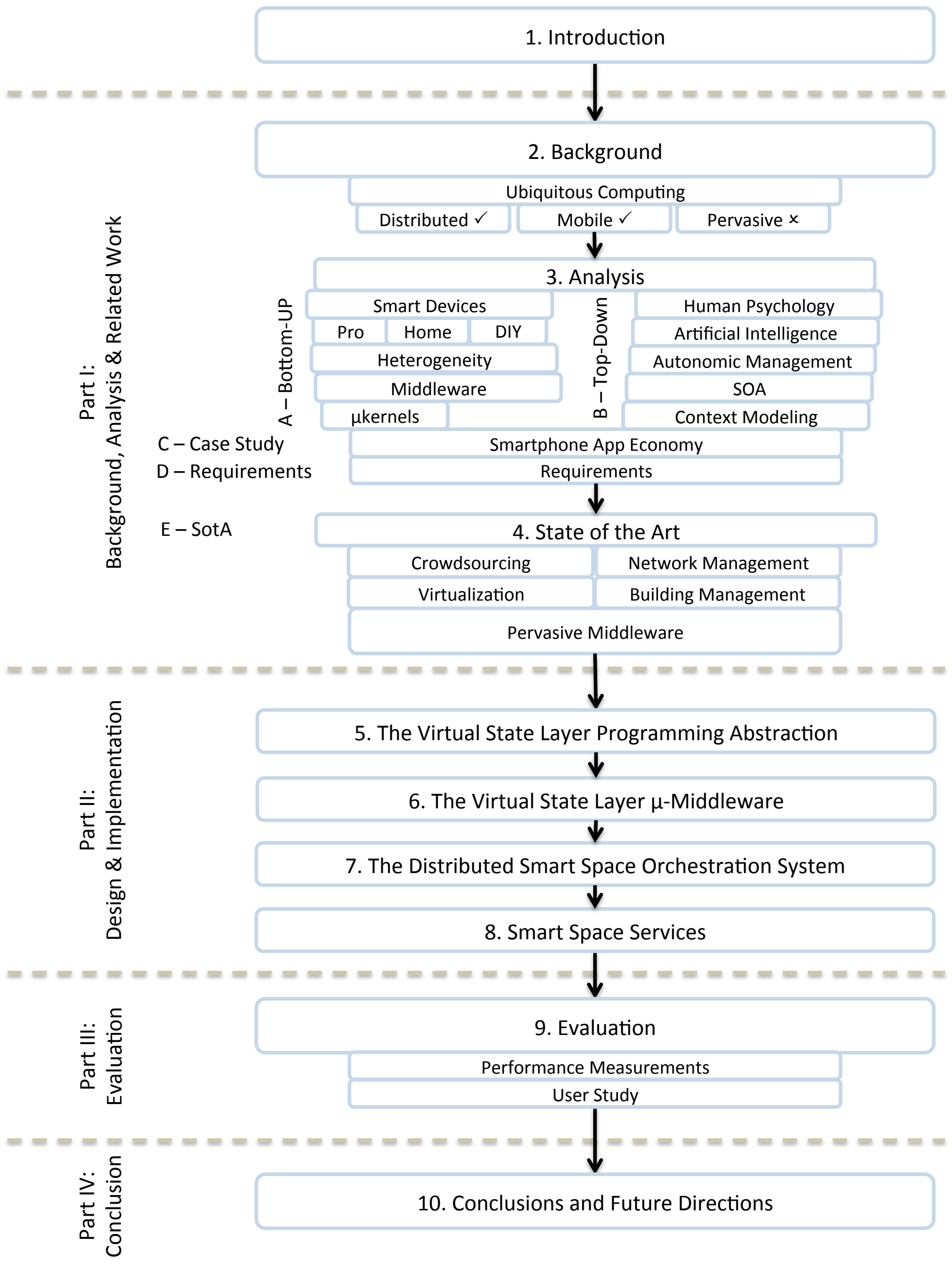


Figure 1.1: The structure of this thesis.

A hybrid meta model is presented. It is used for creating context models that are shared over a global Context Model Repository (CMR) that becomes a self-managed collaborative ontology. A collaborative automated convergence mechanism for context models is introduced. The context models are used to offer context storage for services over a fixed Application Programming Interface (API) with 13 methods.
Via a novel concept, called Virtual Nodes, it becomes possible to use context models as abstract service interfaces. This enables a SOA of Smart Space services.
It is shown how the programming abstraction implements distribution transparency and autonomy, and how it fosters portability of services and security-by-design.

Chapter 6 (The Virtual State Layer μ-Middleware) introduces the μ-middleware concept. An implementation of the VSL programming abstraction as distributed, self-managing P2P system is presented.
Programming language specific connectors are introduced as concept for supporting developers.
Implementation details of the basic services that were introduced in Sec. 5.7 are given for illustrating the advantages of the μ-middleware design.

Chapter 7 (The Distributed Smart Space Orchestration System) introduces the DS2OS framework. DS2OS consists of different VSL services plus an App store. Together they provide the technical foundation for an App economy for Smart Spaces (Sec. 3.10).

All components of DS2OS besides the Smart Space Store (S2Store), the App store for Smart Spaces, are implemented as regular VSL services to demonstrate the advantages of the programming abstraction.

DS2OS provides autonomous service management within a site, and service deployment from the central S2Store.

Different mechanisms that support crowdsourced development, and real world deployment of Smart Space services are introduced. The collaborative convergence mechanism from Ch. 5 is extended to a crowdsourced convergence mechanism.

A fully distributed, simple-to-use security concept for Smart Space services is introduced.

DS2OS is assessed using the criteria from the state of the art analysis (Sec. 4.5).

Chapter 8 (Services) demonstrates how the VSL programming abstraction facilitates the implementation of services. Implementations of six relevant service classes demonstrate the practical use of the VSL programming abstraction.
As additional structuring element for Java services, a service template is introduced.
It is shown how the VSL abstraction allows automated integration of Smart Devices, automated correlation of context on different levels of abstraction, remote access, generic user interfaces, and the implementation of complex scenarios using simple-to-configure ECA rules.

Chapter 9 (Evaluation) evaluates the VSL quantitatively and qualitatively.
The *latency* of service coupling over the VSL is measured using the two coupling mechanisms, *value change notification* and *Virtual Context*.
The *scalability* of the VSL is evaluated in a large testbed, and a smaller one that can be expected to be closer to future real-world deployments. The self-management overhead of the VSL is evaluated.
A user study is presented that shows the real world usability of the VSL.

Chapter 10 (Conclusions & Future Directions) concludes the thesis and provides an outlook how the future development of the presented solution could be, and how the research will be continued.

1.4 Document Structure

Fig. 1.1 shows the structure of the thesis.

The objectives of this thesis (Sec. 1.1) are marked with the sign <O.0>. Ch. 2 and Ch. 3 identify requirements that are marked with the sign <R.1>. In the chapters Ch. 5ff the fulfillment of a requirement is marked in the text and at the margin next to the text by showing the respective requirement identifier (<R.1>). *<R.1>*

Each of the following chapters starts with a short summary, key results, key contributions, a list of the identified requirements (<R.1>) that are addressed in the chapter (where applicable), and a longer summary of the chapter. They are marked as follows:

Summary This is the executive summary of the chapter.

Important terms are introduced in definitions:

Definition ***Definition***

Definitions are identified by the shown special formatting.

The listings in this thesis are only labeled when they are referenced later. This is done intentionally, not to disturb the reader as the chapters 5ff contain several listings on some pages.

At the end of the document, an index, a glossary, and an acronym list are provided.

Part I

Background, Analysis & Related Work

2. Ubiquitous Computing

Bringing computers into the home won't change either one, but may revitalize the corner saloon.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

Short Summary The chapter puts this work into the context of ubiquitous computing, and provides a high-level analysis of the implementation of its parts in 2014. It is shown that ubiquitous computing research comprises distributed computing, mobile computing and pervasive computing. While the distributed computing and mobile computing parts of the ubiquitous computing vision are reality in 2014, pervasive computing is only reality in limited domains, or research prototypes.
Heterogeneity of Smart Devices, and the missing of a suitable programming abstraction are identified as inhibitors for a implementation of pervasive computing in the real world.
It is discussed that user-participation is desired for the development of services that implement pervasive computing scenarios.
The terms Smart Space, Smart Device, Smart Space Orchestration, and context are defined. As pervasive computing comprises several sub domains, this work is put into context to Internet of Things (IoT), Cyber-Physical System (CPS), and Ambient Intelligence (AmI). Context-aware computing is introduced as basic paradigm behind pervasive computing.

Key Results Pervasive computing is the missing building block for the implementation of ubiquitous computing. Devices and networking technology are available. Only the software is missing.
The missing of a suitable programming abstraction for pervasive computing is identified as key challenge by researchers, and by the market.
User-participation is desirable when creating software for Smart Spaces.

Key Contributions The missing of a suitable programming abstraction for pervasive computing is identified as major open challenge for pervasive computing to become the third generation of computing.

Summary The vision of ubiquitous computing is introduced (Sec. 2.1). Ubiquitous computing can be decomposed into distributed computing, mobile computing, and pervasive computing (Sec. 2.1.1). The temporal and systematical correlation between the three computing paradigms is shown.

The implementation of the three paradigms in 2014 is analyzed using the three criteria (1) devices, (2) software, and (3) network which Weiser introduced as assessment criteria in his 1991 article.

Distributed computing is briefly introduced as its technology was available already in 1991 and it still is (Sec. 2.2).

Mobile computing is analyzed in detail as its devices and networking technology are part of the hardware base for pervasive computing and as such relevant for this work (Sec. 2.3). Mobile computing technology is used to connect pervasive systems. Mobile devices are suitable user interfaces for pervasive computing, and can be used as sensors or actuators. Network, devices, and software of mobile computing are reality for a broad audience in 2014.

For pervasive computing the devices and the networking technology are available in 2014 (Sec. 2.4). The implementation of Smart Space Orchestration lacks an abstraction that provides a unified system view to the heterogeneous and distributed hardware of Smart Spaces.

The missing of a suitable programming abstraction for pervasive computing is identified as key challenge by the research community (Sec. 2.5), and by the market (Sec. 2.4.4). It is the topic of this thesis (<O.0>).

User-participation is desirable when creating software for Smart Spaces (Sec. 2.5.2).

IoT, CPS, and ambient intelligence are used to describe research that focuses on different aspects of pervasive computing (Sec. 2.6). Context-aware computing is the software paradigm behind pervasive computing applications (Sec. 2.6.4).

2.1 The Vision

Mark Weiser coined the term “ubiquitous computing ” in 1991 as head of the Computer science laboratory at the Xerox Palo Alto Research Center (PARC) [Wei91, Wei93]. He describes ubiquitous computing as a paradigm that “takes into account the human world and allows the computers themselves to vanish into the background” [Wei91]. See Fig. 2.1.

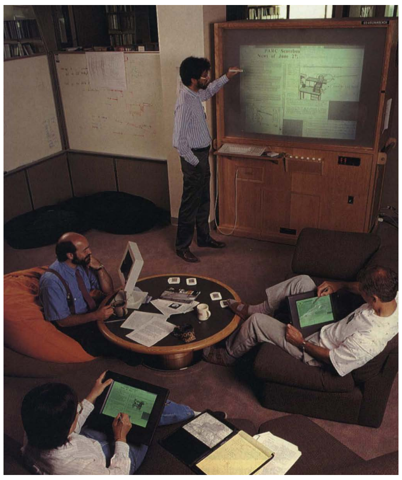


Figure 2.1: Ubiquitous computing at Xerox PARC in 1991. In the background is a so-called board that is used with a pen. The men in the foreground on the right use so-called tabs. [Wei91]

Weiser envisions human-centric interaction with computers that happens always and everywhere in the background. Technologies should “weave themselves into the fabric of everyday life”. This contrasts the machine-centric human-computer interaction that dominated in 1991 (Sec. 2.5). Weiser uses the term “embodied virtuality” for describing the interaction of computers with the reality outside their “electronic shells” [Wei91].

Weiser formulates three technical requirements for the implementation of ubiquitous computing [Wei91]:

1. “*cheap, low-power computers that include equally convenient displays,*
2. *software for ubiquitous applications, and*
3. *a network that ties everything together*”

The 1991 paper [Wei91] and the 1993 paper [Wei93] introduce prototypical solutions for all three challenges. The prototypes became reality until 2014.

The man on the bottom and the man on the right of Fig. 2.1 use so-called PARC tabs. Tabs are reality as tablet computers in 2014, e.g. in form of the Apple iPad[1] that was released in 2010.

Weiser describes two application scenarios, an *office scenario* [Wei93], and a *home scenario* [Wei91]. The *office scenario* introduces tools for Computer Supported Cooperative Work (CSCW). An example for a commercial CSCW tool is Adobe Connect[2] in

[1]http://www.apple.com/ipad/
[2]http://www.adobe.com/products/adobeconnect.html

2014. The office scenario contains so-called *active badges* for tracking entities. Radio identifiers are commercially available as Radio Frequency IDentification (RFID) tags and regularly used for tracking goods in 2014 [VD10, ISO08, YZYN08] (Sec. 2.6.1).

For the *home scenario*, Weiser describes the awakening of a character, called *Sal*, in an ubiquitous computing world in the future. In this vision, computing interacts with the physical environment [Wei91]. *Sal* is informed about events in the environment. Her alarm clock is coupled with the coffee machine.

The vision of computers that interact with people's regular environments ("everyday's fabric") is not reality in 2014 [BLM+11, DMA+10, KFKC12, MH12, ABI12] (Sec. 2.4). Sec. 3 shows that the software basis for such pervasive computing is still missing.

Concerning the networking technology, the paper presents infrared technology for wireless networks. Such networks are regularly available in 2014 and in many use cases replaced by (longer range) radio-based network technology (e.g. WiFi, mobile phone networks) (Sec. 2.3).

2.1.1 Dividing the Vision

Weiser's vision can be decomposed into distributed computing, mobile computing, and pervasive computing [LY02a, SM03, Sat01]:

- *Distributed computing* was already reality when Weiser formulated his vision in 1991. It focuses on different aspects of providing services on distributed hosts (Sec. 2.2).
- *Mobile computing* focuses on providing ubiquitous access to computing resources (Sec. 2.3).
- *Pervasive computing* focuses on the augmentation of men's daily life with computers (Sec. 2.4).

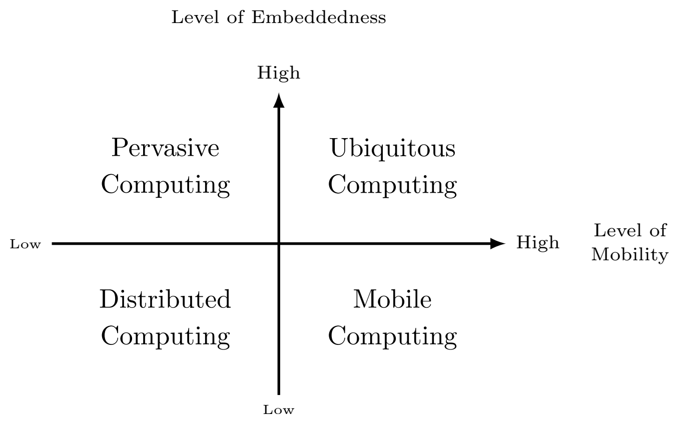


Figure 2.2: Dimensions of ubiquitous computing according to [LY02a, SM03, Sat01].

Fig. 2.2 shows the connection of the three sub fields of ubiquitous computing in terms of embeddedness and mobility. Distributed computing provides a low level of embeddedness and mobility. Mobile computing provides a high level of mobility. Pervasive computing provides a high level of embeddedness. The implementation of ubiquitous computing requires all three parts to become reality (Fig. 2.4). Following, the implementation of the three dimensions in 2014 is assessed according to Weiser's three requirements on *hardware*, *software*, and *network*.

2.2 Distributed Computing

"*A distributed system is a collection of independent computers that appears to its users as a single coherent system.*" [Tan04]. Software that provides a so-called *unified system view* is called middleware. Middleware is introduced in Sec. 3.4. Networked computing systems were already reality in 1991, they are ubiquitous in 2014 (Sec. 4.3, Sec. 3.2).

2.2.1 Devices, Network, and Software

Most computers have a network interface in 2014. Regular computers can be used as part of a distributed system. Devices for distributed computing are available in 2014.

Ethernet over copper or fiber is the predominant networking standard for distributed computing in 2014 [SRV08]. Networking technology for distributed computing are available in 2014.

Cloud computing is a recent software paradigm for distributed computing (Sec. 4.3). It drives many popular web services that are available over the Internet (Sec. 2.3.3). Various middleware for implementing distributed systems with distributed hardware is available in 2014 (Ch. 4). Software for distributed computing are available in 2014.

2.2.2 Assessment

Using Weiser's criteria (1) devices, (2) network, and (3) software, distributed computing is reality in 2014 as it was in 1991.

2.3 Mobile Computing

Mobile computing describes the shift from fixed workstations with specific purposes to the mobile use of computing resources [LY02a]. Heterogeneous ways of data processing and transmission, a high level of mobility, and large-scale services and infrastructures are characteristic for mobile computing [LY02b].

Mobile computing technology is available and a part of many people's *daily lives* in 2014. The following analysis shows that Weiser's requirements for (1) devices, (2) software, and (3) network are fulfilled for mobile computing and that the technology is commonly used in 2014. As the network is fundamental for mobile computing it is presented before the analysis of devices and software.

2.3.1 Network

Wireless networking technology progressed since Weiser's paper. The success of the Internet [vEF12] led to demand-for, and the invention-of various network access technologies. For Personal Area Networks (PANs) (∼10m) technologies like IEEE 802.15.4 (including ZigBee), or IEEE 802.15.1 (Bluetooth) are specified. For Local Area Networks (LANs) (∼150m) wireless Ethernet is specified as IEEE 802.11{a,b,g,n}. For Metropolitan Area Networks (MANs) (<50km) IEEE 802.16 (including WiMAX) is specified. For Wide Area Networks (WANs) (<150km) cellular networks provide mobile access. Ubiquitous wireless broadband Internet access is reality in many areas in 2014 (Fig. 2.3).

Especially cellular networks are interesting for (ubiquitous) mobile access to the computing resources of the Internet as they do not require near-by infrastructure. Cellular networks make broadband Internet access at high bandwidth and low latency [BCL12, ITU11] ubiquitous via technologies such as third generation (3G) Universal Mobile Telecommunications System (UMTS), or fourth generation (4G) Long-Term Evolution (LTE) networks [Cis12].

According to [ITU13] there are more mobile phones than the world's population in 2013. About 75% of the population in the developed world have a subscription to mobile broadband in 2013 (30% of the world's overall population) [ITU13]. Internet access is a major reason for subscribing to mobile broadband [WM10]. About 77% of the population of the developed world are actively using the Internet in 2013 (39% of the world population) [ITU13]. Via cellular networks, the Internet is an ubiquitously accessible resource in areas that are covered with a mobile signal. Since 2007 (release of the iPhone) the access to the global network via mobile devices has grown continuously. The raise of mobile data traffic is expected to continue growing exponentially [Cis12, ITU11].

Most parts of Europe and Northern America are covered with mobile signals. Fig. 2.3 shows the mobile signal coverage as it is measured by users. The actual coverage can be expected to be higher as the samples are sparse in many regions. Combined with satellite phones the mobile network coverage can be expected to be high in many parts of the world. Mobile access to the global Internet is available and daily routine for many people in 2014 [vE13, Cis12, ITU11].

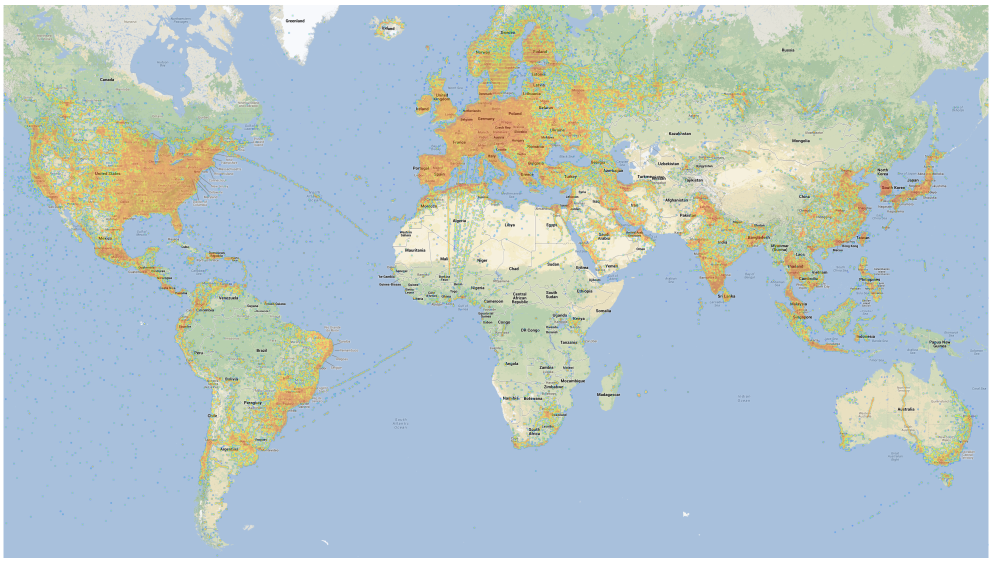


Figure 2.3: World coverage with cellular signal *measured via crowdsourcing* (excluding satellite coverage). The signal strength is indicated by the colors (red=strong – blue=weak). Created on Aug 23, 2013 with data from http://opensignal.com/.

2.3.2 Devices

Different device classes for mobile computing exist in 2014, including tablets, smartphones, ultrabooks, and laptops. Smartphones and the development of the cellular broadband networks are important enablers for real world mobile computing [WM10].

The amount of mobile connected devices per user is expected to be two or more for a quarter of the mobile users until 2016 [Cis12, HLH+11] (Sec. 2.5.1). The sales of mobile computing devices including smartphones, and tablets are significantly higher than those of fixed PCs in 2013 [MTC+13, RvdM13, VSCK11, HLH+11, Wan09]. People are actively using mobile computing and invest in the technology.

The German Online Study [vEF12], a yearly representative survey in Germany, shows that the access to the Internet via smartphones has grown from 8% in 2010 to 22% in Germany in 2012 to 41% in 2013 [vE13]. In the second quarter of 2013 more smartphones than other mobile phones were sold world wide showing that mobile Internet access is a world wide trend [vdMR13, Cis12, MW12].

Mobile devices are available and their use is daily routine for many people in 2014.

2.3.3 Software

As mobile runtime environment, laptops allow mobile computing since the 1970s [KG77]. A big amount of software is available for laptops. They run software from stationary desktop computers while providing mobility.

"Full" mobile computing in the sense that people have their computing devices at hand emerged with the development of smartphones and the App economy (Sec. 3.10) since 2008. An important success factor for the first commercially successful smartphone, the Apple iPhone, was the access to the regular Internet in 2007. Until 2007, the smartphone Operating Systems (OSs) of Apple's competitors typically provided access to adapted World Wide Web (WWW) pages only using the Wireless Application Protocol (WAP) profile [WM10].

The resources offered by mobile devices are constantly growing. Mobile phones are almost as powerful as desktop PCs some years ago. Cloud computing (Sec. 4.3) and low-latency high-bandwidth access networks provide mobile devices with access to a powerful computation back-end [FLR13]. Web sites like Google, Facebook, Youtube, Yahoo, Baidu, Wikipedia, Twitter, or Amazon are applications that run directly in the Internet. All named applications are heavily used [Ale13]. In 2012 the Internet was mostly used to extend the resources of smartphones for videos, music, social media, web traffic, and email [Cis12].

Applications that run directly on a smartphone are called Apps. Apps often do not require an Internet connection. In combination with mobile devices such as smartphones Apps enable mobile computing everywhere for regular users (Sec. 3.10).

In 2008, the Apple App store opened for third party developers. This enabled crowd-sourced software development for smartphones (Sec. 3.10). A whole new economy for software development emerged with billions of Apps and downloads [PMC$^+$12, VSCK11]. Apps are an important decision criterion for smartphone buyers [Eri12]. This shows the acceptance and broad use of mobile computing software.

Software applications for mobile devices are available and used daily by many people in 2014.

2.3.4 Assessment

Solutions that address Weiser's challenges [Wei91] for (1) devices, (2) software, and (3) network exist and are widely used. Mobile computing is reality in 2014.

2.4 Pervasive Computing

This thesis is about the implementation of pervasive computing in the real world.

Pervasive computing focuses on the penetration of human environments with computing technology. Typical tasks are monitoring and controlling environments via Smart Devices [LY02b, Woo09, SM03].

Fig. 2.4 shows the inclusion of distributed computing, and mobile computing in pervasive computing. The innermost part of the figure is a *computer* as basic entity for computing. When a *computer* is embedded in a device that serves more purposes than being a computer, it is called *embedded system* [VD10].

By adding *networking support* the *computer* becomes a *distributed system*. Adding *mobility support* (e.g. wireless communication technology and portability), it becomes a *mobile system*.

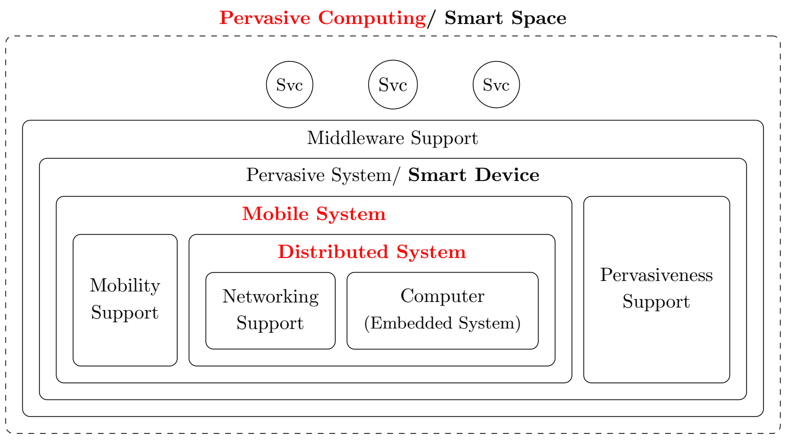


Figure 2.4: Parts of pervasive computing illustrating the relationship between distributed mobile and pervasive computing. The visualization combines information from [SM03, Sat01].

Sensors and actuators enable computing devices to interact with their physical environment. This is called *pervasiveness support*. A networked computing device with pervasiveness support is a *pervasive system*. It is called *Smart Device* in this thesis. *Networking support* is required for *Smart Devices* [Tan04, III98, Sat01].

As discussed in Sec. 3.2, *mobility support* is not required for *Smart Devices*. *Mobility support* is interesting for accessing pervasive functionality in a space remotely, e.g. for controlling the heaters of a building remotely. Via the described ubiquitous Internet connection (Sec. 2.3) this can happen from many places with the solutions presented in Ch. 8.

For making distributed *Smart Devices* accessible, *middleware* is used (Sec. 3.4). *Middleware* facilitates the access to computing resources of a distributed system as described in Sec. 2.2. The main contribution of this thesis is a novel class of middleware that is called μ-middleware (Sec. 6.2). It facilitates the access to distributed Smart Devices.

Using the middleware as platform, services (*Svc*) can implement pervasive computing scenarios for implementing *pervasive computing*. A physical space that contains the described functional entities is called *Smart Space* in this document.

Pervasive computing is the topic of this document. Sec. 2.4.1 defines several important terms that are used throughout this document. The terms can also be found in the glossary at the end of the document. More terms are defined when needed.

The implementation of pervasive computing in 2014 is assessed according to Weiser's criteria in the order *hardware*, *network*, and *software*. This order is chosen as the former two are reality in 2014, while the *software* is not. A more detailed analysis of the current implementation state of pervasive computing follows in Ch. 3.

2.4.1 Definitions–Smart Space, Smart Device, Smart Space Orchestration

The terms *Smart Space*, *Smart Device*, *Smart Space Orchestration*, and *context* are not clearly defined in literature. Their intended meaning in the context of this document is defined here.

Definition ***Smart Space***

> (Real world) spaces that contain Smart Devices and support Smart Space Software Orchestration are called Smart Spaces.

Another term for *Smart Space* [CD05, KKR$^+$13, HWN10, BSG12] is *intelligent environment* [Coe98, Kru09].

Definition ***Smart Device***

> A Smart Device is an embedded system with pervasiveness support via sensors or actuators, and networking support that enables remote control.

An alternative name for a *Smart Device* is *pervasive system* [SM03, Kru09].

Definition ***Smart Space (Software) Orchestration***

> The novel term Smart Space Software Orchestration describes the management of Smart Devices via software. For better readability the term Smart Space Orchestration will be used as synonym to Smart Space Software Orchestration in this document. Smart Space Orchestration typically has a goal that it reaches by orchestrating multiple Smart Devices.

Smart Space Orchestration of Smart Devices allows dynamic composition of existing hardware to reach a certain goal such as saving energy (Sec. 2.4.4) [Ber96, HIM05a, RCKZ12, DMA$^+$12, DHKT$^+$13, SHB10]. The term orchestration is used similar to the web service context (Sec. 3.8). A conductor performs a composition by orchestrating his orchestra consisting of musicians playing their instrument. Similar, a software service reaches a Pervasive Computing goal in a Smart Space by orchestrating the (adaptation) services (Sec . 8.3.1) that manage their Smart Devices.

In Ch. 6 of this thesis, a middleware is presented that enables Smart Space Orchestration. It enables services to orchestrate distributed Smart Devices via unified abstract interfaces. See *Svc* in Fig. 2.4.

Definition ***Context***

> Context (from Latin *con-texere*, to weave together) is information that is relevant for an entity to reach its goal.
> In this document, the term *context* is used to describe the virtual representation of information that is relevant for a service to implement Smart Space Orchestration (*virtual object* in Fig. 3.12). Such *context* representation is typically structured by so-called context models (Sec. 3.9).

The previous definition connects the term context with services as services are in the focus of Smart Space Orchestration. In other literature, context has different dimensions including computing context, physical context, time context, or user context [ST94, YZYN08, Dey01, HIR02, BCQS07, CK00, BCFF12]

2.4.2 Devices

Smart Devices are the interface between physical environments and software (Sec. 3.2). They define which interaction with the environment can be managed by software.

In professionally used environments (e.g. office buildings), Smart Devices are available for application domains such as Heating, Ventilation, Air-Conditioning (HVAC), lighting, or security in 2014. Devices in so-called Building Automation Systems (BASs) typically communicate over standardized protocols such as X10, KNX, LonTalk, or BACnet [KNSN05, KFKC12, BLM+11] (Sec. 3.2.1).

Professional BASs are often too expensive and too complex for private households [KNSN05, BLM+11]. Smart Devices for sensing and actuating physical environments are available for consumers as embedded systems off-the-shelf at affordable prices in 2014 [CD12, DMA+12, BLM+11]. They typically offer remote control functionality for various purposes including electrical power, heating thermostats, light switches, presence sensors, or computer equipment. Consumer devices are typically controlled via vendor proprietary protocols or web interfaces that can both be accessed remotely using Hyper Text Transfer Protocol (HTTP) over Transmission Control Protocol (TCP)/ Internet Protocol (IP) over wired or wireless Ethernet for instance (Sec. 3.2.2).

Besides buying off-the-shelf hardware, it is possible to built so-called Do-It-Yourself (DIY) customized hardware with standard parts in 2014. Platforms such as the Arduino project [Ban08] enable the remote control of various kinds of sensors and actuators over a network (Sec. 3.2.3).

Smart Devices use heterogeneous communication protocols. The resulting heterogeneity in the device access is a problem for implementing Smart Space Orchestration with existing hardware. The heterogeneity makes it difficult to integrate devices in complex scenarios. For reasons discussed in Sec. 3.3.1, so-called *silos* of vendors or functional domains emerged [DHKT+13, DMA+12].

Smart Devices for various purposes in the professional and in the consumer domain are available in 2014.

2.4.3 Network

Physical network connectivity between Smart Devices and computers that run services is necessary for Smart Space Orchestration. Smart Devices can be connected over wired or wireless connections (see Sec. 2.3.1). Networking technology for connecting pervasive systems is available in 2014 (Sec. 2.3.1, Sec. 2.2).

The emergence of Smart Spaces is likely to happen by retrofitting existing spaces with new Smart Devices. Networking technology for retrofitting spaces into Smart Spaces is available on the market in 2014 with wireless technology (Sec. 2.3.1) and communication technology that uses existing cables such as Power Line Communication (PLC).

In Weiser's article the term "network" probably covers not only physical connectivity but also syntactic connectivity and semantic connectivity [Wei91]. As mentioned in Sec. 2.4.2 and analyzed more in depth in Sec. 3.3.1, heterogeneity is an inherent property of Smart Devices (Sec. 4.4.2).

Technology for connecting Smart Devices is available and used daily by many people in 2014. However, solutions for bridging heterogeneity on the semantic level is required for implementing pervasive computing with diverse Smart Devices (Sec. 3.3.5, Sec. 5.6.3).

Providing interoperability, and portability of services from heterogeneous Smart Devices are a key challenge that is addressed in this thesis (Sec. 5.6.3).

2.4.4 Software

Smart Space Orchestration allows dynamic reconfiguration and flexible interaction with existing hardware [KKR+13, Ber96, HIM05a, CD12]. In software orchestrated Smart Spaces, software implements the pervasive computing scenarios.

In 1991, Mark Weiser described a scenario where a person (Sal) wakes up and different systems (e.g. the coffee machine) interact to assist her starting the day [Wei91] (Sec. 2.1). Implementing the described scenario is possible but challenging in 2014 (Ch. 3).

In research projects many use cases with a complexity similar to or higher than that in the scenario Weiser describes were implemented in the past [SM03, Har03]. Domains that were interesting for research projects [KKR+13, HWN10, Abo12, WL05] include health care and well-being [KD07, WL05], e-Learning and campus life [HKLC08, AM00], tourism and traveling [CKKC07], office and other business applications [CPC+04, WL05], security and safety [BCG12, MH12, KKK06, WL05], energy saving [Dar10, WN07, PNKC13, WL05], advertising and e-commerce [SM10, WL05], entertainment [APV06, DMA+12], user convenience [HMEZ+05, CD07, Hel05, WL05], gaming [BHLM02], and social community applications [ERS10, GPJB07].

Typically, the described research projects implement their solutions as specialized software that works with exactly the hardware used in the project's laboratory environment. The resulting software does often not support the implementation of other use cases (Sec. 4.5.2). Designing software for project-specific scenarios is time intense for each project, and inflexible as the resulting diverse middleware systems can often neither be reused nor transferred into real environments [KSSR05].

The successful implementation of diverse pervasive computing scenarios in research projects shows the feasibility and the power of Smart Space Orchestration [Kru09]. For many of the use cases that are presented in the cited research projects, standard hardware can be used for implementing the scenarios in 2014.

In real world private households the situation is similar to the one of the research projects (Sec. 3.2.2). Enthusiastic DIY hackers build customized hardware-/ software-setups for their specific hardware and application use cases that run inside their environments and cannot be transferred to other environments [BLM$^+$11, MH12].

There are not many automated private households in 2013 [TPR$^+$12, BLM$^+$11] but analysts see high demand and the need for new solutions as "*no company is able to provide all the parts, so telecom, cable, security, and utility providers are all looking to smart devices vendors, managed software providers, local installation specialists, and others to support the broad rollout of home automation services*" [ABI12].

Silo solutions for professionally used buildings, research projects, and enthusiastic home owners show that Smart Space Orchestration can be implemented with standard components in 2014 [Abo12, BD07].

2.4.5 Assessment

Solutions that address Weiser's challenges [Wei91] for (1) devices, (2) software, and (3) network exist for pervasive computing. Because of the missing software support, and the high complexity, pervasive computing is only used in limited domains by a limited amount of people. The situation of pervasive computing in 2014 is analyzed in detail in Ch. 3.

The key problem for Smart Space Orchestration in 2014 is the missing software development support for overcoming the existing diversity of Smart Devices (<O.0>). Portable services cannot be created as a common layer of abstraction is missing. This thesis introduces such a layer (Ch. 5, Ch. 6) and shows how it can be used for implementing Smart Space Orchestration in the real world (Ch. 7, Ch. 8, Ch. 9).

2.5 Revisiting the Vision – A Science Perspective

The conference for ubiquitous computing (UbiComp) and the conference for pervasive computing (PerCom) are the main conferences in the area of ubiquitous computing in 2014. At the 2012 UbiComp, two papers about the state of ubiquitous computing research 20 years after Weiser's vision were presented [Abo12, DM12]. They show the relevance of the problems that are addressed in this thesis for the science community.

The first paper [Abo12] (Sec. 2.5.1) is about the need for a suitable programming abstraction that pushes the technical challenges of implementing pervasive computing in the background and the applications in the foreground.

The second paper [DM12] (Sec. 2.5.2) is about pervasive computing strongly lacking user participation. The authors criticize that pervasive computing technology and its applications are created by few people and imposed on the rest of the world.

2.5.1 Celebrating an intellectual disappearing act

Gregory Abowd is a computer scientist and ubiquitous computing pioneer at Georgia Institute of Technology, Atlanta. He revisited Weiser's vision from a historic perspective on the evolution of computing in his article "What next, Ubicomp? Celebrating an intellectual disappearing act" [Abo12].

Abowd states that ubiquitous computing is about to become the third computing generation after *mainframe computing*, and *personal computing* [Abo12, Kru09]. Similar to the short analysis from Sec. 2.4, his conclusion is that –though ubiquitous computing has matured over the years– a powerful simple-to-use interface is "*unfinished business*" [Abo12].

Abowd characterizes the advent of a computing generation with a focus shift in people's attention. A computing generation is considered as implemented when its technology shifts into the background and is taken for granted. For that to happen, a powerful simple-to-use abstraction is essential.

Picking Apple's HyperCard as an important enabler for *personal computing*, Abowd puts a Wikipedia quote into his article that could also be used to express the goal of this thesis by replacing Hypercard with the Virtual State Layer (VSL) programming abstraction: "*Many people who thought they would never be able to program a computer started using HyperCard for all sorts of automation and prototyping tasks*" [Abo12].

This leads to the first requirement on a programming abstraction for pervasive computing:

<R.1> **Simplicity-to-use** is required for a programming abstraction for pervasive computings.

A Little History of Computing

Abowd's article contains a summary of past computing generations and their characteristic properties. Such a review can help identifying trends. Following, the review from the paper is extended with additional material from other reviews [Abo12, Kru09,

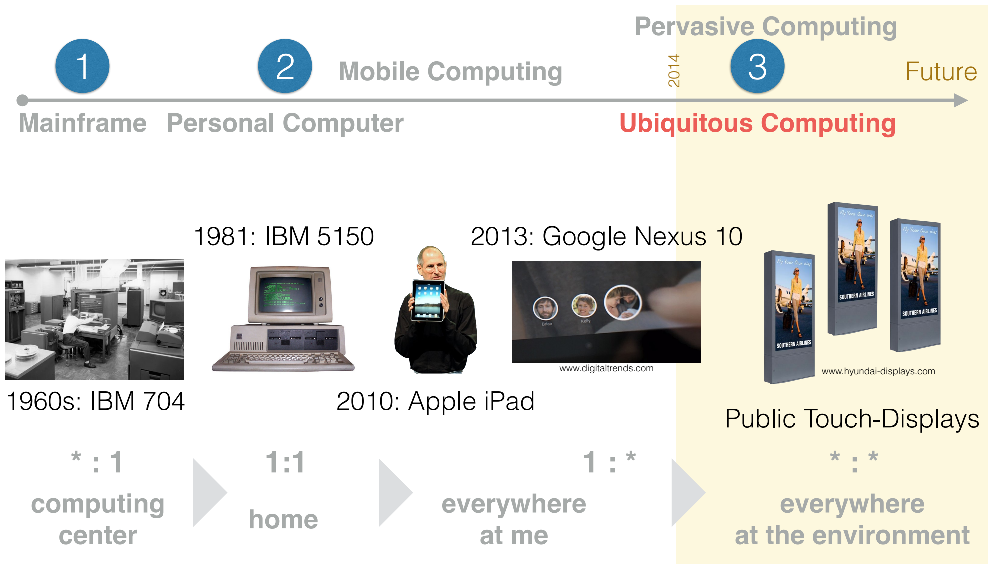


Figure 2.5: Ubiquitous computing as third generation of computing. The middle shows representative technology of each generation. The bottom shows the cardinality of the relation human:device, and the location where computing happens.

CD07]. The analysis focuses on the devices that are the user interfaces of each generation, the typical usage pattern of the technology in the cardinality of the relation human:device, and the location where the computing happens.

Fig. 2.5 shows the evolution of computing from the 1960s to 2014 and beyond. The top part of the figure shows the timeline. The circles mark the advent of a technology. The first generation of computing are *mainframes*. The second generation of computing are *personal computers*. The third generation is *ubiquitous computing* according to Abowd [Abo12].

Sec. 2.4 and Sec. 2.3 show the divisibility of *ubiquitous computing* into *mobile computing* and *pervasive computing*. The sub areas are shown at the top. As discussed in Sec. 2.3, *mobile computing* is reality in 2014. The implementation of *pervasive computing* is still an open challenge (Sec. 2.4).

The beginning of computing is in the 1960s. The first computing generation is characterized by mainframe computers that are located in computing centers and operated by many people that do time sharing (*:1).

The second generation is personal computing. It starts in the 1980s and is characterized by personal computers that are operated in offices or homes. Operators typically have a dedicated computer or share it with few others (e.g. family, 1:1). Personal computers are taken for granted by a broad audience and used regularly for creating diverse applications.

The first part of the third generation of computing starts around 2000 with different mobile communication technologies being deployed [ITU11]. An important technological advance towards mobile computing is the introduction of the iPhone and the App economy in 2008 (Sec. 3.10). It brings diverse applications for mobile computing to a broad audience.

Characteristic for mobile computing are portable computing devices with ubiquitous connection to the Internet. The relation between human operators and devices is typically one-to-many (1:*). Many people possess multiple devices [ITU11, Cis12]. Mobile computing happens everywhere. Mobile devices are typical personal devices. Smartphones are taken for granted by a broad audience and used regularly for creating diverse applications in 2014 [vE13, ITU13].

The introduction of multi-user operating system functionality on mobile devices starts a shift from the one-to-many interaction paradigm towards shared use (many-to-many, *:*). Multiple operators share multiple devices. In 2014, multi-user supporting devices are typically personal devices such as tablet computers that are shared, e.g. by all family members. With the impelementation of pervasive computing, it can be expected that User Interfaces (UIs) become available in the ambiance (ambient computing). As an example, public interactive displays [JPSM13] are shown on the right of Fig. 2.5. Pervasive computing (or ambient intelligence, Sec. 2.6.3) is not reality in 2014 yet (Sec. 2.4). The current state of the implementation is analyzed in Ch. 3 and Ch. 4.

Fig. 2.5 shows a trend of disappearance concerning computing hardware – from filling entire rooms in 1960s to ambient intelligence in the future (Sec. 2.6.3).

The ratio between users and devices evolves from many-to-one (*:1) over one-to-one (1:1), and one-to-many (1:*) towards many users sharing many devices (many-to-many, *:*) in the ambiance.

The location of computing changes from device-and-location-bound, over device-bound-and-location-unbound (mobile computing, Sec. 2.3) in 2014, towards device-and-location-unbound in the future with public personalized computing interfaces in the ambiance.

The described trend towards many-to-many usage, and computing devices in the ambiance introduce new requirements that a suitable programming abstraction for future software orchestrated Smart Spaces should support:

<R.2> **Multi-user support** should be provided by a programming abstraction as shared use of hardware is expected to become a regular usage pattern.

<R.3> **Self-management** of ambient components must be supported by an abstraction. The ubiquity of ambient devices in the future requires self-management to handle complexity and to make the technology usable by non experts.

2.5.2 UbiComp's Colonial Impulse

Paul Dourish is a computer scientist and anthropologist at the University of California, Irvine. Scott Mainwaring is an ethnographer at Intel. While Abowd gives a technology-centric review, Dourish and Mainwaring review ubiquitous computing from a user perspective. They criticize "*UbiComp's Colonial Impulse*" in their paper [DM12].

Dourish and Mainwaring claim that ubiquitous computing research colonializes the world. They compare the research with the behavior of the British empire that imposed its habits to its colonies. Ubiquitous computing research acts similar: few researchers define possible applications of the pervasive computing technology for many future users.

The authors postulate the liberalization of ubiquitous computing research and development by introducing participatory elements. They claim that the users know best what they want and they should not be influenced by tools or paradigms that limit the creativity. This adds the following requirements to the design of a programming abstraction for pervasive computing:

<R.4> Support of **user participation** in the development of pervasive computing.

In addition to supporting user-based development, the authors see the necessity to openly consider perspectives from many more domains than it currently happens in pervasive computing research. The requirements analysis in Ch. 3 of this thesis considers a broader perspective than it is usually found in literature (Sec. 3.1.2).

2.5.3 Conclusion

Weiser's article starts, "*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life.*" [Wei91]. Abowd uses this assessment criterion as an identification mark of a computing generation [Abo12]. That a technology disappeared can be noticed when it fades into the background as people are simply using it without explicitly noticing the existence of the technology. This happened for mainframe computing, personal computing, and more recently for mobile computing.

Experts from different fields share the assessment that from a technical point of view, ubiquitous computing is reality in 2014 (Sec. 2.4). The fundamental problem for the implementation of pervasive computing in 2014 is the missing of a suitable programming abstraction for enabling users to simply use pervasive computing technology [Abo12, Kru09, BD07, CD07].

The creation of such a programming abstraction is the topic of this thesis (<O.0>).

2.6 Related Terminology

The terms *Internet of Things (IoT)*, *Cyber-Physical System (CPS)*, and *ambient intelligence* are used for describing research in pervasive computing. Though all terms describe pervasive computing, their focus diverges. The terms are briefly introduced for orientation.

Context is fundamental for pervasive computing. Context-aware computing is the computation paradigm behind pervasive computing that is applied in this thesis. Context-aware computing is therefore briefly introduced at the end of this section.

2.6.1 Internet of Things (IoT)

The term Internet of Things (IoT) describes the implementation of pervasive computing using a device-centric perspective. The goal of Internet of Things (IoT) is connecting every object in the world over a network. Objects should form spontaneous networks with their neighbors and exchange information [ITU05]. Internet technology is used for communication in the IoT [YZYN08]. Relevant technologies are Web Services (WSs) (Sec. 3.8), or the Representational State Transfer (REST) paradigm for stateless resource access [Fie02].

Central challenges in the IoT are the identification and tracking of objects and the collection and processing of data from the connected objects. Typical objects are people, consumables, or medication. A key technology that is used in IoT scenarios are RFID tags. RFID tags are passive elements that can be used to exchange data with objects [ITU05, YZYN08]. They are primarily used to identify and track objects. RFID tags are the modern implementation of active badges (Sec. 2.4).

For obtaining information about the environment of objects, sensor networks and robots are used in the IoT vision. IoT aims to embed its technology into every object [ITU05].

The IoT vision follows a bottom-up strategy from devices to applications (Sec. 3.1.2). This work provides a middleware (paradigm) that is complemented by IoT research (Sec. 2.6.5). IoT devices typically use *standardized* Internet technology for communication. This simplifies interfacing them (Sec. 3.3.2, Sec. 8.3.1).

2.6.2 Cyber-Physical Systems (CPS)

Cyber-Physical System (CPS) research focuses on the integration of physical processes with (remote) computation. Smart Devices are coupled with (remote) services that manage them in software [BCG12, DDMS12, RI10, Chu10, Lee06].

While IoT research focuses on devices, Cyber-Physical System (CPS) research focuses on software-based management of devices. Using software for management allows to change the control logic of hardware devices dynamically. In addition, it allows to shift control logic from Smart Devices into a remote backbone that is connected via a network [Lee06, DDMS12]. This allows the execution of computationally intense control logic that could not run on a resource limited Smart Device. In addition, remote software control allows to couple multiple Smart Devices [DDMS12, BCG12].

The middleware (paradigm) that is introduced in this thesis (Sec. 5) can be used as platform to implement CPSs.

2.6.3 Ambient Intelligence

IoT and CPS have their focus on device hardware. Ambient Intelligence (AmI) focuses on the applications that emerge in Smart Spaces. AmI research addresses users and the user experience with pervasive computing environments (Smart Spaces) [AHS01, HA02].

2.6.4 Context-Aware Computing

Services that implement pervasive computing scenarios typically acquire context to fulfill their goals (Sec. 3.6). As described in Sec. 2.4.1, context is information that is relevant for services to fulfill their goals. Information about a physical environment can be sensed via Smart Devices and then be used as context for a service.

A computing system is context-aware when it is aware of the context in which it runs [SAW94]. Pervasive computing is inherently context-aware computing. To create context-aware computing applications, context has to be retrieved and processed. Different tasks of context-aware computing are identified in the different layers of Fig. 2.6 [KKR+13, HIM05a].

Smart Devices are on the lowest layer. They interact with their environment over sensors and actuators. Data is retrieved from the Smart Devices (data retrieval), and transformed into context information (context processing). The resulting context is stored and exchanged (context management). As support for services (applications), a decision support layer can provide functionality to infer information from the available context, e.g. by applying first order logic (Sec. 4.5.2). Compare to Sec. 8.3.2.

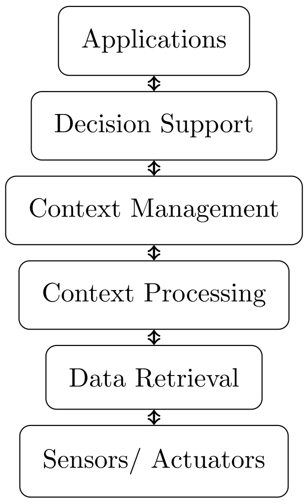


Figure 2.6: Layered design of context-aware systems according to [KKR+13, HIM05a].

Many middleware for pervasive computing covers all or most of the layers between the applications and the sensors and actuators in Fig. 2.6 (Sec. 4.5).

The context-awareness of a computer system can be classified in three stages [KKR+13, BCFF12]:

- **Context-based adaption** is supported, when services can query context synchronously on-demand.

- **Context-aware adaption** is supported, when services can react to asynchronously diffused context events.
- **Situation-aware adaption** is supported, when primitive contexts can be aggregated, high-level context is created via reasoning, and can be used as decision basis for services.

The VSL programming abstraction that is introduced in Sec. 5 supports all three levels of context-awareness (Sec. 5.3.2, Sec. 8.3.2).

Major tasks of context-aware computing are context modeling, context management, and context reasoning [KKR+13, YZYN08]. This thesis focuses on the first two aspects, and allows to add context reasoning via regular services (Sec. 8.3.2, Sec. 5.7.3, Sec. 6.2).

The following requirement for reaching the overall objective <O.0> of this thesis emerges:

<R.5> **Context-awareness support** must be provided by a suitable programming abstraction for pervasive computing since pervasive computing is inherently context-aware computing.

The requirement is further specified in Sec. 3.6.4.

2.6.5 Related Terminology Conclusion

Different terms describe different aspects of pervasive computing. All terms typically cover the entire pervasive computing domain but they have different focus.

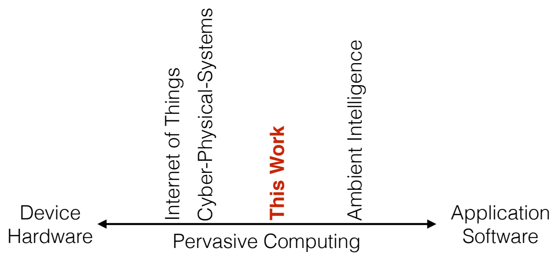


Figure 2.7: Positioning of this work in the context of the presented different aspects of pervasive computing research.

Fig. 2.7 shows the positioning of this work in the context of the described research areas. The middleware that is introduced in this work can use the devices that are in the focus of IoT. It provides a basis for CPS with their focus on orchestrating devices via (exchangeable) software. For ambient intelligence applications this work provides a basis that can be used for a rapid implementation and implementation of pervasive computing workflows.

Pervasive computing is inherently context-aware computing which makes context-awareness support a fundamental requirement for a suitable programming abstraction for pervasive computing (<R.5>).

2.7 Chapter Conclusion

Ubiquitous computing comprises distributed computing, mobile computing, and pervasive computing. Distributed computing and mobile computing are reality for a broad audience in 2014.

The evolution of Distributed computing and mobile computing is ongoing as massively scaling distributed systems (Sec. 4.3), or growing bandwidth in network transmissions over wireless[3] or wired links show. But solutions that answer Weiser's challenges for devices, software, and network are part of people's daily lives for these two subareas of ubiquitous computing.

This is not the case for pervasive computing. Smart Devices, and networking technology exist and are available for many purposes in regular shops off-the-shelf in 2014. The highest deployment rates of BAS is in professional buildings [CJ08, BLM$^+$11]. However, most spaces are not smart in 2014 as they are not orchestrated by software as a whole. The potential of the existing Building Automation System (BAS) installations is typically not exhausted. Pervasive computing scenarios that integrate different application domains are seldom.

The heterogeneity of the communication interfaces offered by the available Smart Devices results in complexity of remotely controlling diverse devices. This complexity is a key problem for the implementation of Smart Spaces in 2014 (Sec. 2.5, Sec. 2.4).

It is possible to develop software that implements pervasive computing scenarios in 2014. Implementations of limited scenarios in professional environments (BAS), in prototypical Smart Spaces of research projects, and in the homes of enthusiastic DIY users exist.

A suitable programming abstraction for Smart Space Orchestration is missing to implement pervasive computing as third generation of computing (Sec. 2.5). The availability of such an abstraction would allow pushing the technical challenges of implementing pervasive computing into the background. The technical hurdles are currently preventing a real world implementation of pervasive computing.

User participation in the development of pervasive computing scenarios and technology is required to foster innovation and to enable pervasive computing in the real world.

[3]http://grouper.ieee.org/groups/802/11/Reports/802.11_Timelines.htm

3. Analysis

Everything should be built top-down, except the first time.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

Short Summary This chapter analyzes requirements on real world Smart Space Software Orchestration. Knowing the requirements is relevant for the design of a *suitable* programming abstraction as most requirements have to be supported by the abstraction directly (Ch. 5).
In a bottom-up, and a top-down analysis the problem space is structured. Bottom-up, Smart Devices from different domains, solutions for bridging heterogeneity, middleware, and μ-kernel operating system design are analyzed. Top-down, psychology, Artificial Intelligence (AI), autonomic management, Service Oriented Architecture (SOA), and context modeling are analyzed. The App economy is analyzed as a successful example of the recent implementation of a subdomain of ubiquitous computing in the real world.
The analyzed fields give background for the design of the Virtual State Layer (VSL) programming abstraction in Ch. 5ff.
The 50 identified requirements are summarized and discussed in Sec. 3.11ff.

Key Results Sec. 3.12 contains a list with a brief description of all identified requirements.
Fig. 3.15 graphically maps the requirements to the objectives of this thesis.
Sec. 3.13.2 discusses the requirements in the context of their fulfillment in the state of the art.
The seven layers that are typically found in literature to describe context-aware computing in Smart Spaces (Fig. 2.6) can be reduced to four domains.
Smart Devices inherently use heterogeneous communication protocols. Using one protocol for all devices is not desired.
The goals of this thesis complement Do-It-Yourself (DIY) hardware making. Together, both empower users to entirely implement their pervasive computing scenarios.

Abstract interfaces have to be provided to implement *interface portability*. Standardization is required for implementing interface portability. Established standardization processes are not suitable for software orchestrated Smart Spaces. **A suitable standardization process for Smart Spaces is missing.**

Middleware is a suitable concept for designing a programming abstraction for Smart Spaces. It helps providing *interface portability* and *execution environment portability*.

A suitable programming abstraction for Smart Spaces should support all inter-service communication modes to enable the implementation of diverse pervasive computing scenarios.

Dynamic extensibility is required. The μ-kernel approach shows how it can be implemented in operating systems.

Declarative interfaces are more intuitive than procedural interfaces. Humans are familiar to the concepts of abstraction and modularization.

Finite State Machine (FSM) that model state transitions can be used to model intelligent behavior in computer programs. Event-Condition-Action (ECA) rules are a simple form for expressing *plans*. Autonomy can be reached by applying methods of AI.

The Web Service (WS) architecture is not sufficient as programming abstraction for future Smart Spaces.

A suitable meta model that enables user-based ontologies creation for Smart Spaces is missing.

Crowdsourcing could help fostering the emergence of Smart Space Orchestration in the real world.

Security-by-design is required to enable running third party services in software orchestrated Smart Spaces.

The state of the art is unlikely to be suitable for a real world deployment as it does not take enough requirements into account.

Key Contributions A requirements analysis that exceeds the literature [HIM05a, EBM05, HR06, BDR07, Kjæ07, CD07, HSK09, GGS09, Kru09, SHB10, BBH+10, Pos11, CD12, BCFF12, RCKZ12, KKR+13] is provided.

Summary This chapter analyzes requirements on real world Software Orchestration in Smart Spaces. Knowing them is relevant for the design of a suitable programming abstraction as most requirements have to be supported by the abstraction directly.

The chapter starts with an introduction of top-down and bottom-up information processing methodologies (Sec. 3.1.1). The problem field Smart Space Orchestration is structured into seven layers (Sec. 3.1.2). The seven layers that are typically found in literature to describe context-aware computing in Smart Spaces (Fig. 2.6) can be reduced to four domains. This reduces the complexity.

The bottom-up analysis starts with Smart Devices and their inherently heterogeneous communication protocols (Sec. 3.2). It analyzes Smart Devices from the professional domain and the home domain.

DIY hardware making is introduced (Sec. 3.2.3). The goals of this thesis complement DIY hardware making. Together, both empower users to implement comprehensive pervasive computing scenarios.

The future of Smart Device hardware is discussed. It is identified as suitable execution platform for Smart Space Orchestration (Sec. 3.2.4).
Reasons for the inherency of Smart Device communication protocol heterogeneity are analyzed (Sec. 3.3). The suitability of classic standardization processes is analyzed (Sec. 3.3.2). Standardization is required for implementing interface portability.
Established standardization processes are not suitable for software orchestrated Smart Spaces. **A suitable standardization process for Smart Spaces is missing.**
Topology locations for bridging heterogeneity are discussed (Sec. 3.3.3). "*In-the-middle*" is identified as best location. Different solutions and problems for bridging heterogeneous communication protocols are discussed (Sec. 3.3.4).
Middleware is introduced as concept to bridge heterogeneity (Sec. 3.4). Middleware is a suitable concept for designing a programming abstraction for Smart Spaces. It helps providing *interface portability* and *execution environment portability*. A suitable programming abstraction for Smart Spaces should support all inter-service communication modes to enable the implementation of diverse pervasive computing scenarios. Dynamic extensibility is required. The μ-kernel approach shows how it can be implemented in operating systems (sec. 3.4.4).
The top-down analysis starts with human psychology (Sec. 3.5). It introduces the observed behavior of the human mind, and structured thinking. Declarative interfaces are identified as being more intuitive than procedural interfaces. The concepts of abstraction and modularization are familiar to humans.
Looking at AI (Sec. 3.6), FSMs that model state transitions can be used to model intelligent behavior in computer programs. ECA rules are a simple form for expressing *plans*. Autonomy can be reached by applying methods of AI.
Autonomic computing is introduced at the example of autonomic management that applies AI principles (Sec. 3.7). SOA (Sec. 3.8) and context modeling (Sec. 3.9) are introduced. **A suitable meta model that enables user-based ontologies creation for Smart Spaces is identified as missing.**
Success factors of the App economy that enabled the real world implementation of mobile computing are introduced (Sec. 3.10). **Crowdsourcing could help fostering the emergence of Smart Space Orchestration in the real world. Security-by-design is required to enable running third party services in software orchestrated Smart Spaces.**
The 50 identified requirements are summarized and discussed in Sec. 3.11ff. Sec. 3.12 contains a list with a brief description of all identified requirements. Fig. 3.15 graphically maps the requirements to the objectives of this thesis. Sec. 3.13.2 discusses the requirements in the context of their fulfillment in the state of the art. The state of the art is unlikely to be suitable for a real world deployment as it does not take enough requirements into account.

3.1 Top-Down and Bottom-Up Analysis

Pervasive computing touches many research areas as it is defined by a vision (see Sec. 2.1) unlike other research domains that are based on technical problems, and have a body of results that is built over time [BD07]. This chapter contains the main part of the requirements analysis of this thesis. The diversity of pervasive computing makes the task of analyzing requirements for the design of a suitable programming abstraction for pervasive computing [Abo12] at large complex (<O.3>) [DM12, CD12, Abo12, Kru09].

Following the introducing quote by Alan Perlis, this entire thesis is structured *bottom-up* starting with a requirements analysis of different parts of the problem space. This chapter identifies requirements for Smart Space Software Orchestration as basis for the design of the Virtual State Layer (VSL) programming abstraction in Ch. 5.

The two viewpoints *bottom-up* and *top-down* are applied several times in this chapter as methodologies to look at a problem from two opposite sides. Sec. 3.1.1 introduces the terminology. Sec. 3.1.2 provides an analysis of the problem space that determines the space of the main analysis that is done in the remainder of this chapter (Sec. 3.2ff).

The chapter has four parts:

- Part I–A contains a *bottom-up* analysis from Smart Devices up the software layer 4 in Fig. 3.1. The part is structured *bottom-up* internally.
- Part I–B provides a *top-down* analysis from human developers to the software layer 4 in Fig. 3.1.
- Part I–C contains a case study about the App economy that is an important enabler for the real world implementation of mobile computing (Sec. 2.3.3).
- Part I–D summarizes and structures the requirements that are identified in this chapter.

The main contribution of this thesis is the VSL programming abstraction (Ch. 5) that is situated on layer (4) of Fig. 3.1. The services on the layers (3) and (3a) are intended to be created by third party developers following the request for user-empowerment in Sec. 2.5.2. Ch. 8 introduces services on the layers 3 and 3a to illustrate how the VSL programming abstraction supports developers in the creation of diverse services (<O.3>). Assuming that the humans at the top of Fig. 3.1 are developers, the approach of this chapter with its *bottom-up* and *top-down* analysis is visualized in the figure.

3.1.1 Top-Down and Bottom-Up Information Processing

As described in the introduction (Sec. 3.1), a top-down analysis and a bottom-up analysis are used as strategies for structuring the analysis in this chapter. The terms *top-down* and *bottom-up* describe two general information processing principles [EHS10, JL93, BPR99]. Both strategies are relevant for achieving the goal of this thesis: facilitating the development of services that implement pervasive computing scenarios.

Top-Down

Solving a problem top-down means applying a *deductive* process. Deduction means looking at a problem as a whole first, and partitioning it into layered sub-modules that solve parts of the problem then. Each resulting layer of abstraction regards the functionality of its underlying layers as a so-called black box. The term black box describes that only the interface not the internal details of the underlay module are relevant for using it (Sec. 3.8). Synonyms for top-down are *analysis* or *decomposition* [EHS10].

Top-down information processing emphasizes *high-level* planning and is based on analyzing a problem as a whole. Looking at problems from an abstract point of view and then going into details is similar to the information processing that happens in the human mind (Sec. 3.5) and therefore intuitive [JL93].

Bottom-Up

Bottom-up problem solving strategies follow the opposite, *inductive* approach. They start with the parts and assemble them to more complex solutions. Synonyms for a bottom-up design process are *synthesis* or *composition*.

A major advantage of a bottom-up approach is that it fosters the reuse of basic parts for implementing *different* complex workflows. New functionality can be assembled from existing parts [EHS10].

Software Design

Top-down and bottom-up strategies apply to software design as particular form of information processing. The top-down approach was successfully introduced to software design by Harlan Mills and Nikolaus Wirth at IBM in the 1970s [EHS10, Wir71].

A goal of the VSL programming abstraction is to facilitate the implementation of complex pervasive computing workflows in software (<O.0>). Enabling a top-down development process with the programming abstraction supports this goal. Ideally, a programming abstraction for Smart Spaces supports hiding details such as how Smart Devices are connected, and which device-specific communication interfaces they offer (Sec. 3.3.2, Sec. 3.4, Sec. 3.8).

A major advantage of a bottom-up software design is the reusability of components. Reusing components lowers the complexity of software development. It reduces the required lines of code of Smart Space services since functionality that was developed (and tested) before is reused. At the same time it makes software more robust. Reused components are typically better tested than single-use components as they are part of diverse other services.

A combination of the *high level of abstraction* of the top-down methodology with the *reusability* of the bottom-up approach is desirable. Both is considered for the VSL programming abstraction. The former is reflected by providing abstract interfaces (Sec. 3.3.2) and self-management (Sec. 5.6.2, Sec. 6.4). The latter is reflected by providing a Service Oriented Architecture (SOA) (Sec. 3.8).

3.1.2 Structuring Smart Space Orchestration

Design a programming abstraction that structures and facilitates the development of services for Smart Spaces requires an understanding of the problem domain. Smart Space Orchestration can be structured into different functional layers.

Fig. 3.1 shows *seven functional layers* when the *right part of the figure that connects the layers is ignored*. The layers are introduced bottom-up.

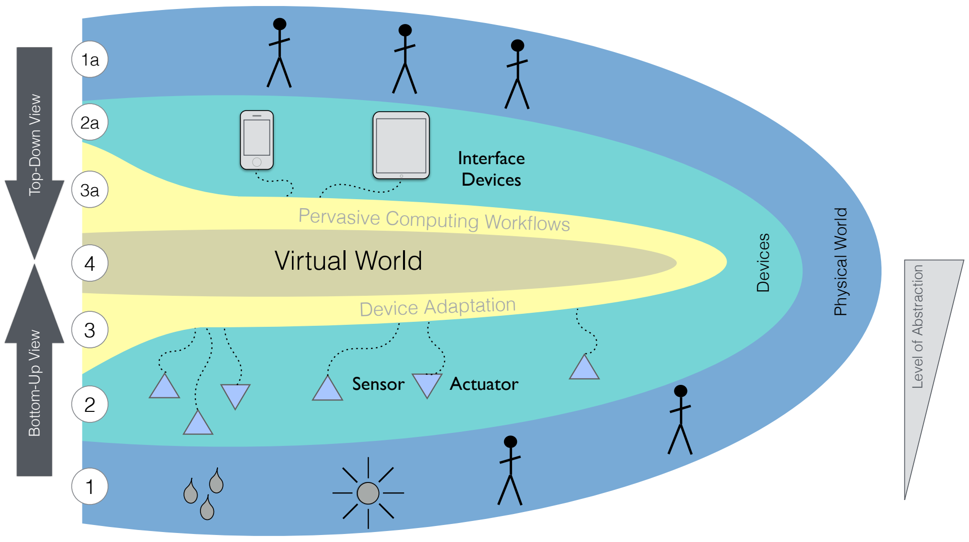


Figure 3.1: Functional view (seven layers) and domain view (four rings) on Smart Space Orchestration.

The bottom layer (1) shows physical phenomena. As described in Sec. 2.4.2, and analyzed in detail in Sec. 3.2, Smart Devices are used as interface between the physical world and the virtual world.

The virtual world contains the software for Smart Space Orchestration. Layer 3 of Fig. 3.1 shows services for device adaptation. They are called *adaptation services* in this thesis. They connect Smart Devices with their virtual interfaces in layer 4. The virtual interfaces (4) are the software representation of the Smart Devices. They are used by the services on layer 3a to implement pervasive computing scenarios.

To take the humans in a future Smart Space into the control loop, services on layer 3a offer User Interfaces (UIs). Such UIs can be accessed by devices such as smartphones (2a) as described in Ch. 2. The top layer (1a) are the humans that give high-level orchestration goals to the services on layer 3a over their interfaces.

Fig. 3.1 is similar to Fig. 2.6 that can be found in the literature. The analysis in Ch. 4 shows that the *decision support* (Fig. 2.6) is typically part of the middleware (layer 4 in Fig. 3.1) in state of the art pervasive computing middleware designs. The same applies to the *context processing* layer in Fig. 2.6. The purpose of layer 3 in Fig. 3.1 is *data retrieval* in the literature. The purpose of layer 3a in Fig. 3.1 is hosting *applications*.

The middleware that is presented in this thesis in Ch. 5 and Ch. 6 does not follow this approach. It provides only *context management* in layer 4 of Fig. 3.1. *Context processing* is provided in layer 3 (Sec. 8.3.1), and *decision support* is provided in layer 3a (Sec. 8.3.2). See Fig. 5.1.

Connecting the Layers

The objective of this thesis is designing a programming abstraction that structures and facilitates the development of pervasive computing applications (<O.0>). The focus of this work is consequently on the layers 3, 4, and 3a of Fig. 3.1.

Looking at Smart Space Orchestration from a top-down and a bottom-up view, pairwise similarities between the seven functional layers become visible. See Fig. 3.1 on the left. The layers 1 and 1a contain entities of the physical world. The layers 2 and 2a contain computing devices that act as interfaces between the physical environment on the outside and its virtual representation in the software domain in the innermost area.

The layers 3 and 3a contain software services that adapt between the specific interfaces that are used on layer 2 and layer 2a and the abstract interfaces that are used on layer 4. Layer 3 contains adaptation service towards the sensor and actuator hardware. Layer 3a contains orchestration services, user interface services, and other software services that help implementing pervasive computing scenarios (Ch. 8). The core layer (4) manages the virtual world as abstract representation of the physical world. It provides the context for services (Sec. 2.6.4).

The described understanding results in a domain view on Fig. 3.1. The four domains {(1,1a) physical world, (2,2a) computing hardware, (3,3a) services, (4) common abstraction} are indicated by the rings, the colors, and the pairwise naming of the layers. This domain view reduces the complexity of Smart Space Orchestration. The seven layers from the *functional view* that can be found in literature (Fig. 2.6) collapse into four rings in the *domain view*.

The outer two domains (1, 2) are reality in 2014. Domain (1, 1a) is the reality. Devices in the domain (2, 2a) are available off-the-shelf for consumers (Ch. 2). A suitable abstraction on layer 4 that structures and facilitates the creation of services in the surrounding domain is missing (Sec. 2.7, <O.0>). This thesis focuses on the software side of Smart Space orchestration. The domains (3,3a) and (4) are assessed in this chapter.

This thesis shows advantages of collapsing the layers 3 and 3a. This explicit step is not done by the assessed middleware designs (Sec. 4.5). Instead, *device adaptation* functionality (3) and *applications* (3a) are treated separately. The combined reference architecture in Fig. 4.5.16 that combines the aspects of all assessed middleware designs from Ch. 4 reflects this separation.

Treating services for *device adaptation* (3) and services that implement *pervasive computing workflows* in the same way reduces the complexity for service developers. The same mechanisms of layer 4 can be used to implement both service categories. Treating all services equally in terms of the used programming abstractions, developers can implement drivers for Smart Devices using the same programming interface they use to implement orchestration workflows.

The advantages of treating the layers 3 and 3a identically from a programming point of view lead to the requirement:

<R.6> Device-adaptation services and other services should not be treated separately but have a **role-independent unified interface**.

Level of Abstraction

The level of abstraction in Fig. 3.1 raises from the bottom layer to the top layer in the *functional* perspective with the seven layers. Sensors and actuators collect and change properties of the physical world on a concrete level (bottom layer). Humans give high-level input (top layer).

In the *domain* view, the level of abstraction raises from the outer ring to the inside as the entities on the outer rings are concrete while the software domain in the middle is virtual and contains abstract interfaces.

Subpart I–A

Bottom-Up – Bridging Device Heterogeneity

3.2 Smart Devices

Smart Devices are the interface between the physical world and the virtual world (Fig. 3.1). The hardware that is available in a Smart Space defines which information can be sensed from a physical environment, and which properties of the space can be altered by software via actuation.

Smart Devices are used in diverse application domains in 2014 including smart grid, industrial automation, smart cities and urban networks, home automation, building automation, structural health monitoring, or container tracking [VD10, ITU05].

As described in Sec. 2.4, various devices for Smart Spaces are available off-the-shelf in 2014 [BD07, Abo12, Kru09, BLM$^+$11]. For reasons analyzed in Sec. 3.3.1 Smart Devices use heterogeneous communication protocols. Heterogeneity in the communication protocols is a major reason for the complexity of Smart Space Orchestration today [BLM$^+$11, MH12, DLG07, KFKC12].

To give a better understanding of device heterogeneity and existing use cases for Smart Spaces, the professional building automation domain (Sec. 3.2.1) and the home automation domain (Sec. 3.2.2) are analyzed in this section. The analysis is structured in

- *installation environment*, showing where Smart Spaces exist already,
- *typical scenarios*, giving ideas for pervasive computing scenarios,
- *organization of the systems* giving ideas for structuring Smart Space Orchestration in general,
- *device control interfaces*, showing existing heterogeneous interfaces on layer 2 of Fig. 3.1,
- a discussion about the *interoperability* that is needed for Smart Space Orchestration,
- *standardization efforts* in the domain, that foster interoperability, and
- available *interfaces to the domain* that help integrating it into future software orchestrated Smart Spaces.

Technology for Do-It-Yourself (DIY) development of Smart Devices is introduced. Finally, possible directions for future Smart Devices are described. They show that Smart Devices could become the backbone for future Smart Space Software Orchestration.

The technical terms and protocols that are introduced in this section are *not* relevant in the remainder of this document. They are introduced to give an overview of the terminology that is used in the assessed domains. They should enable the reader to study literature from the domain more easily.

3.2.1 Building Automation

The term building automation comprises the orchestration, mechanization, and data aggregation of Smart Devices in buildings. The systems used for automation of professional buildings are called Building Automation Systems (BASs). Other names for BAS with partially different focus are Facility Management System (FMS), Building Management System (BMS), Energy Management System (EMS), or Intelligent Building System (IBS) [VD10, p.361].

BAS are relevant for the orchestration of Smart Spaces as they contain Smart Devices that are deployed in existing buildings. Smart Devices for professional use have typically standardized interfaces. This facilitates their integration in software that implements pervasive computing scenarios (see Sec. 8.3.1).

Installation Environment

Professional BASs are deployed in commercial buildings such as public buildings including universities, and hospitals, or private buildings including the hospitality sector or manufacturing. The targeted market for BAS systems are buildings with sizes above 100K square feet [VD10, p.361].

Typical Scenarios

The installation of Smart Devices in workplaces is typically driven by productivity concerns [Hin99]. Scenarios and use cases for building automation include [VD10, WL05, KNSN05, SIE13, RLT11]

- **occupancy and shutdown** to configure Smart Devices of rooms according to their occupancy,
- **energy management** to optimize the energy consumption of building systems during occupancy,
- **demand response** for interactive optimization of the energy consumption of multiple buildings in cooperation with an energy provider,
- **fire and smoke abatement** with support of BASs (especially the ventilation systems), and
- **evacuation** with support of BASs for informing occupants of a building via an intercom system to evacuate for instance.

The scenarios comprise diverse application domains and building subsystems of a building [HLM+97, DHKT+13, CJ08]: **energy saving** (e.g. load balancing, natural ventilation, daylighting), **comfort** (e.g. Heating, Ventilation, Air-Conditioning (HVAC), lighting, workstation and workgroup design for improved spatial, thermal, acoustic, visual, and air quality), **safety** (e.g. fire alarms), **security** (e.g. access control), and **accounting** (e.g. presence detection).

Organizational View

BASs can be structured on three layers [WS97, SIE13]:

- **Management level**: operation and monitoring, evaluation, management.
- **Automation level**: control and room automation.
- **Field level**: sensors and actuators.

This section focuses on solutions for the lower two layers, *automation level* and *field level*. The market dominant solution for the management level is introduced in Sec. 4.4.2.

Device Control Interfaces

After pneumatics (1950s), and electric circuits (1960s), microprocessors were included into the control loops of Smart Devices for building automation in the 1970s. The microcontrollers are called *Programmable Logic Controllers (PLCs)*. The concept of using them for automation is called *Direct Digital Control (DDC)*. With so-called *Supervisory Control And Data Acquisition (SCADA)*, supervisory systems for the distributed DDCs were introduced as *Central Control and Monitoring Systems (CCMS)* [WS97, KNSN05]. They implement the layered structure that was introduced before.

Different communication protocols are used on the automation level over different network topologies such as bus, point-to-point, star, or ring. BAS communication protocols are often domain-specific protocols. They are used for specific purposes such as lighting (Digital Addressable Lighting Interface (DALI)) or meter reading (M-Bus). Following, some protocols are named. The identifiers in brackets denote standards that define the technology. Technologies without identifiers in brackets are de-facto standards (Sec. 3.3.2). Frequently used protocols are [KNSN05, KFKC12, BLM$^+$11, WS97, VD10, MHH09, TK10]:
EIA-485 (American National Standards Institute (ANSI)/TIA/EIA-485), *1-wire*, *M-Bus*, *Johnson Controls Metasys N2*, *DALI* (International Electrotechnical Commission (IEC)60929, IEC62386) over twisted pair, *Ethernet* (IEEE802.3) over different physical media, and *IEEE802.11* and *IEEE802.15.4* [CGH$^+$02, Ega05a] for wireless communication, *Modbus* (e.g. over EIA-485 or Ethernet), *KNX* (EN 50090, ISO/IEC 14543) over different media, *LON* often run over twisted pair (ISO/IEC 14908) *Building Automation Control network (BACnet)* (American Society of Heating, Refrigerating and Air Conditioning Engineers (ASHRAE)/ANSI Standard 135, International Organization for Standardization (ISO)16484-5) over different media (Sec. 3.3.4). The enumeration of the media gives an impression of the heterogeneity of the different protocols.

Interoperability

Smart Devices of different application domains named before such as lighting control and metering are typically not connected. Historically motivated, different BASs still use dedicated control networks [CJ08]. Smart Devices of different application domains form **silos** [KFKC12, DHKT$^+$13, BPR99, CJ08, SJR11, TKDHC13, KNSN05, RLT11, PNKC13].

Standardization

Professional BAS are typically standardized [SIE13, CJ08, KNSN05]. Few companies dominate the market (market shares given in parenthesis) [CJ08]: Johnson Controls (15%), Siemens (14%), Honeywell (12%), Trane (7%), Tyco Fire and Security (6%), United Technologies (4%), Yamatake (4%), Novar (3%), General Electrics Industrial (3%), Bosch Security Systems (2%). The limited amount of relevant companies enables a standardization process as described in Sec. 3.3.2 to be successful.

The use of standardized technology in professional buildings facilitates the integration of different protocols (see above) for comprehensive management (Sec. 4.4.2). Because of standardized communication protocols, software that manages Smart Devices is automatically portable as vendor diversity is hidden behind the standardized interfaces of the Smart Devices [SIE13, CJ08, BPR99]. For connecting different BAS domains on the *management layer* (see before), still more standardization is needed as the introduction of BACnet in Sec. 4.4.2 shows. It connects so far separate functional domains.

Interfacing the Building Automation Domain

To allow the remote control via applications outside the BAS domain, some BAS controllers expose additional management interfaces. An example for such an interface is Open Building Information Exchange (oBIX) [OAS13]. It is getting developed and standardized within Organization for the Advancement of Structured Information Standards (OASIS).

Open Building Information Exchange (oBIX) is a Representational State Transfer (REST) [Fie02] interface to BASs. It is implemented as Web Service (WS) using the communication protocol Service Oriented Architecture Protocol (SOAP) over Hyper Text Transfer Protocol (HTTP) over standard Internet Protocol (IP). The REST operations are `read` (object), `write` (object), `invoke` (function), `delete` (object). So-called *watches* implement event-based communication. A client can add subscriptions to watches. Clients have to poll their watches periodically for event "notification".

The typical approach for creating interfaces between a specialized domain (BAS) and other domains is providing gateways. Such protocol gateways interface domain-specific communication protocols and offer a subset of the available functionality over domain-external interfaces (Fig. 8.2). Gateways often contain hardware and software for the entire domain-specific protocol stack (Fig. 3.3), starting from the hardware interface to the syntactic and semantic interpretation of domain-specific protocol data [MTMT06, MRA10, PRL10, TMKP02, OAS13].

Sec. 4.4.2 introduces BACnet which is the most widely used standard on the management layer. BACnet also supports WS interfaces [NSY08, Fel00].

Existing gateways facilitate the integration of heterogeneous BAS domains into software orchestrated Smart Spaces. BAS gateways can be integrated identical to Smart Devices in Fig. 3.1 via adaptation services. A difference is that not a single device but an entire BAS domain is connected over an adaptation service (Sec. 8.3.1) in this case.

Conclusion

BASs are relevant for the implementation of pervasive computing scenarios as they provide interfaces to properties of physical environments that are already deployed in buildings [CJ08, KNSN05, KFKC12, HLM^{+}97]. Use cases of the BAS domain can benefit from an integration with functionality outside the BAS domain and vice versa. Few major vendors and the resulting high degree of standardization (Sec. 3.3.2) facilitate the integration of BASs into software orchestrated Smart Spaces.

To allow remote control and interaction with services outside the BAS domain, existing gateways that offer access to a subset of the BAS functionality via Internet protocols can be used. Such gateways facilitate the remote access to the BAS domain as they typically use standardized Internet protocols as second (non BAS domain-specific) interface (Sec. 8.3.1).

3.2.2 Home Automation

BASs for the professional market (Sec. 3.2.1) are typically too expensive for home users. The implementations of the professional scenarios often target large buildings [VD10, CJ08, TKDHC13]. Those use cases that are not specific for professional buildings such as energy saving or security are interesting for the home domain as well.

Home automation systems can be grouped into three categories:

- high-prized systems that are installed by professionals,
- medium- and low-prized systems that are intalled by home owners themselves, and
- DIY systems that are built by skilled home owners themselves.

High-end home control solutions cost between \$20 000 and \$100 000. Installations of such systems can be found in expensive homes in the USA for instance [VD10]. The products cover not only a subset of the scenarios that are covered by professional BASs but also new domains such as entertainment. High-end solutions are typically installed and maintained by experts similar to the professional BASs.

Few vendors share the market of high-priced home automation. Though protocols different from professional BAS are used (see below), the situation can be compared to that of professional BASs. Standardization mechanisms are applied (Sec. 3.3.2).

In the medium-, and low-prized segment, vendor proprietary solutions for domains such as heating or lighting are available [EG01]. Such solutions often allow self-installation by home owners. They typically do not follow standards (Sec. 3.3.1).

Solutions for retrofitting existing buildings are available too. They typically consist of Smart Devices that only need to be plugged into a power socket and communicate over wireless or Power Line Communication (PLC). For making entities without a power socket such as windows smart, battery powered sensors that communicate wirelessly with other components of the *same vendor* can often be bought [MTMT06, BLM$^+$11, MH12].

As third possibility for implementing smart homes, there is a DIY scene of enthusiastic hobbyists that build their own systems [BLM$^+$11, MH12]. Tutorials on the web and the availability of technical building blocks facilitate the creation of DIY Smart Devices in 2014. DIY hardware making is introduced in detail in Sec. 3.2.3.

Installation Environment

Typical installation environments for home automation technology are spaces that are not primarily used for work but for the spare time. As described before, Smart Devices at different price ranges exist for home automation [BLM$^+$11, MH12, RLT11, AKOSS$^+$11]. Single components such as power plugs or heating thermostats that allow remote control via Internet cost about \$50 in 2014[4]. With different products for different purposes, home automation deployments are not bound to a certain space size [VD10].

[4]http://www.conrad.de/

Typical Scenarios

Security and comfort are major driving forces for consumers to buy home automation hardware. Typical use cases for home automation are [VD10, SRT00, Har03, KOA+99, BLM+11, MH12, RLT11]:

- **lighting control**, to control the lights in a space,
- **safety and security**, including intrusion detections and access control for providing actual and perceived security,
- **comfort and convenience**, including entertainment device control, lighting control, energy conservation, access control, and safety and security,
- **energy management**, to optimize the energy consumption of a space (e.g. house, flat) via controlling the lighting, or Heating, Ventilation, Air-Conditioning (HVAC) systems,
- **remote management**, to control systems like the heating from remote for enhancing comfort and security, and
- **Ambient Assisted Living (AAL)**, for supporting independent aging and remote care.

Homes are buildings. Therefore all use cases described for professional building automation apply to homes too. Some use cases are not so important for homes today such as accounting or evacuation. As described, homes automation comprises additional functional domains compared to professional building automation such as entertainment, or medical support.

Organizational View

The organization of BASs in homes is typically hierarchical, similar to the professions BASs (Sec. 3.2.1). In the entertainment domain, and for Smart Devices that are sold independently from combined solutions, flat topologies are common [GBRB12].

Often home automation devices are isolated and not part of larger systems. Examples are power plugs that can be controlled via a web interface that they offer (see below).

Device Control Interfaces

Smart Devices that are not integrated with other Smart Devices into larger automation systems often use standard Internet protocols such as Ethernet, IP, Transmission Control Protocol (TCP), and HTTP. This protocol stack (Sec. 3.3.4) is used by many of the devices that are available for homes [KBY+12, RLT11, Spi03, MTMT06, MRA10, TMK+03].

For Smart Devices that are coupled for fulfilling a combined goal such as the lighting management of a building, there are several standardized communication protocols for home automation. They are used for similar scenarios to professional BASs but they use different protocols. Examples are *X10* over Power Line Communication (PLC), *CEBus* over different media, *LonWorks* (ANSI/EIA 709.1, ENV 13154-2) over different media, *KNX* (ISO/ IEC 14543-3, ANSI/ ASHRAE 135) over different media, *Ethernet* (IEEE802.3) over different physical media, and *IEEE802.11* and *IEEE802.15.4* [CGH+02] for wireless communication [ND00, SRT00, KNSN05, RLT11, MHH09]. The enumeration of the media gives an impression of the heterogeneity of the different protocols.

Home automation comprises domains that are different from the domains addressed by professional BASs. An example is the integration of multimedia devices into home automation. For such integration protocols such as *Universal Plug aNd Play (UPNP)*, and *Jini Is Not Initials (JINI)* are used that provide features for auto-configuration [MRA10, KY12, GBRB12].

Professional buildings are typically installed and managed by experts [KNSN05]. In the home, especially when retrofitting a home space, experts are not always available [EG01]. Self-management plays an important role for the emergence of Smart Spaces in the home domain (Sec. 2.5.1, <R.3>). It often enables self-installation by home owners.

Interoperability

Home automation comprises more functional domains than professional building automation (Sec. 3.2.1). In addition to building-related functionality, domains such as entertainment are relevant. Different domains typically form **silos** [KBY+12, RLT11, Spi03, MTMT06, MRA10, TMK+03].

Functional domain silos exist similar to the professional domain with different protocols (see above). In addition, *vendor silos* exist in home automation. The diversity of functional domains (e.g. entertainment, surveillance, comfort), and the diversity of vendors make standardization difficult (Sec. 3.3.2). See Sec. 3.3.1 for a discussion about heterogeneity.

Standardization

When few vendors dominate a market, it is more likely that standardized (Sec. 3.3.2) communication protocols emerge. In the home automation sector, BAS and entertainment are example domains where such a situation can be observed.

Smart Devices of smaller vendors, and for more specialized purposes than lighting or switching electronic consumers typically use proprietary communication protocols for reasons analyzed in Sec. 3.3.1 [KBY+12]. This leads to **vendor silos**.

Like in the professional building automation, **functional silos** exist in the home automation domain. The integration of more functional domains in home automation than in professional building automation results in increased heterogeneity.

The overall degree of standardization is lower in the home automation domain than in the professional domain for reasons analyzed in Sec. 3.3.2.

Interfacing the Building Automation Domain

In home automation often gateways exist that offer web interfaces for remote control. Such interfaces bridge between domains that run automation-domain-specific communication protocols and the Personal Computer (PC) world [ND00, SRT00].

The success of smartphones (Sec. 2.3) fostered the deployment of web gateways. The success of the App economy encourages hardware vendors to offer smartphone Apps as added value to their products. To interface a vendor's Smart Devices from a smartphone, web gateways are required. Even though such gateways may not have been intended to be used as gateways into closed proprietary silos, they can fulfill this purpose for Smart Space Orchestration.

As they are often designed for interaction with smartphone Apps, such vendor gateways typically use standard Internet technology such as HTTP REST interfaces. Using such standardized technology facilitates the integration of home automation Smart Devices into software orchestrated Smart Spaces (Sec. 8.3.1).

Before vendors deployed web gateways to the *vendor silos* and *functional silos*, such functionality had to be built by researchers (Sec. 4.5) or skilled home enthusiasts for implementing Smart Space Orchestration [Spi03, MTMT06, MRA10, TMK+03, Mat10].

Conclusion

Home automation extends the professional building automation domain by integrating new functional domains such as entertainment. From a Smart Space perspective, this allows software to interact in more ways with the physical environment.

In areas where few vendors dominate the market, standardized technology is used. More functional domains lead to more *functional silos*. More vendors for diverse available automation solutions lead to additional *vendor silos*.

For many standardized communication protocols, gateways towards Internet protocols exist. Through the success of smartphones, vendors offer web gateways to their formerly closed silos. Such gateways can be used for remote controlling devices to implement Smart Space Orchestration.

3.2.3 Open-Source Hardware

The implementation of pervasive computing scenarios in ubiquitous computing research often comprises the creation of customized hardware [AKOSS+11, SBK+11, TPR+12, AGG10, SRN+08]. Real world deployments also often require such customized hardware [BLM+11, MH12, TPR+12].

A Do-It-Yourself (DIY) or maker culture emerged over the past years [Abo12, TWDT13]. With its tutorials and hardware building blocks (see below), it enables not only experts but also skilled technical enthusiasts to build their own hardware. As an example, in the user study that is presented in Sec. 9.6, all 14 participating students managed to build their own complex Smart Device (Fig. 9.22).

Arduino

An important hardware development platform for Smart Devices in 2014 is the Arduino project[5]. The Arduino project started in 2005 at the Interaction Design Institute Ivrea in Italy where one of the founders of Arduino wanted to create a cheap platform for teaching his students about embedded systems [Ban08].

[5]http://www.arduino.cc

The Arduino platform is an embedded system with different digital and analogue inputs and outputs. It can be used to connect non-smart (Sec. 2.4.1) sensors and actuators to build Smart Devices (Fig. 9.22). Different so-called shields are available that can be used to add different communication interfaces to an Arduino board. Examples are an Ethernet shield, or an IEEE802.11 shield. The system is based on an Atmel Reduced Instruction Set Computer (RISC) Central Processing Unit (CPU).

The Arduino is an open-source hardware. The hardware layout is available and can freely be reused. The sharing of the hardware layout led to the availability of very cheap (9$) hardware clones[6]. The Arduino platform is well suitable for the creation of DIY Smart Devices with its low price in combination with the relatively low (<4W) power consumption, and the ability to run self-compiled programs.

Conclusion

Fig. 9.2 and Fig. 9.22 show Smart Devices based on the Arduino platform that were built for the evaluation of this thesis' results (Ch. 9). They show the practical reality of the DIY maker culture in 2014.

The ability to build customized Smart Devices is important for fostering the emergence of future Smart Spaces. It enables technically skilled users to create the hardware they want to have in their Smart Spaces (Sec. 2.5.2). DIY hardware fosters the implementation of diverse pervasive computing scenarios (<O.3>).

The objective of this thesis, to create the basis for DIY software development for Smart Space Orchestration complements the DIY maker culture. Together they empower users to freely implement the pervasive computing scenarios they want (Sec. 2.5.2).

3.2.4 Future Smart Devices

The evolution of devices for BASs comprises pneumatic control in the 1950s, electromechanical control in the 1960s, PLCs in the 1970s, and since the 1980s distributed DDC [WL05] (Sec. 3.2.1). The mass production of microprocessors lowered their price. This made it attractive to use Smart Devices for building automation.

Reasons for using specialized microprocessors instead of standard instruction set microprocessors are lower prices, and lower energy consumption. It can be expected that microprocessors get more energy efficient and cheaper in the future. The developments on the smartphones market show this tendency already. This makes it likely that future controllers for BASs contain standard RISC or Complex Instruction Set Computer (CISC) processors [Wan09].

DIY hardware (Sec. 3.2.3) shows that it is possible to create affordable Smart Devices with embedded microprocessors that support a standard instruction set in 2014. The Intel Galileo project is such an architecture that uses a 32-bit CISC processor[7]. The use of standard instruction set microprocessors allows to run standard software programs on the used processors. The standardization of the instruction set results in portability of services (Sec. 5.6.3).

[6]http://www.indiegogo.com/projects/9-arduino-compatible-starter-kit-anyone-can-learn-electronics
[7]http://www.intel.com/content/www/us/en/do-it-yourself/galileo-maker-quark-board.html

Smart Space Orchestration is implemented by coupling many services (Sec. 2.4.1, Ch. 8). The use of standard instruction set microprocessors allows services for future Smart Space Orchestration to run directly on Smart Devices.

Smart Devices have to be deployed in future Smart Spaces to interface the physical environment. The possibility of running additional services on those then existing Smart Devices makes adding Smart Space Orchestration to such spaces cost neutral and energy neutral.

The software that is developed in this thesis (Ch. 6, Ch. 7, Ch. 8) is designed to support distributed execution on resource-limited devices.

3.2.5 Conclusion

Networked Smart Devices for diverse purposes are available on the market in 2014. Smart Devices typically form *functional silos* and *vendor silos* [Lee06, BLM$^+$11, MH12, DDMS12].

Bridging the Smart Device diversity is a fundamental challenge for Smart Space Orchestration (<O.1>, Sec. 4.5). Available gateways that bridge from domain-specific communication protocols to Internet protocols can facilitate this integration (Sec. 8.3.1).

The possibility to create DIY Smart Devices allows users to develop their own Smart Device hardware. The ability to create DIY hardware complements the goals of this thesis to enable users to develop their own software services, and to implement and share pervasive computing scenarios for their Smart Spaces (<O.3>, <O.4>).

Numerous pervasive computing scenarios for professional and private use of buildings exist (<O.3>).

The use of standard instruction set CPUs in Smart Devices makes it possible using them as run time environment for Smart Space Orchestration. This could foster the emergence of Smart Spaces in the real world. It lowers the necessary investment in hardware for running orchestration services, and it lowers the running costs in form of energy used by the necessary systems for Smart Space Orchestration.

To support running on diverse hardware of future Smart Spaces:

<R.7> **Portability of the implementation of the** programming abstraction itself must be supported.

3.3 Overcoming Heterogeneity

Smart Devices are available for a broad range of sensing and actuation functionalities [KNSN05, BLM$^+$11, MH12]. Sec. 3.2 shows that the diversity of Smart Devices does not only cover their functionality but also their interfaces. Interface heterogeneity covers the physical interfaces of Smart Devices and their software interfaces.

Pervasive computing scenarios typically integrate formerly separate functional domains. Smart Space Orchestration requires interfacing diverse Smart Devices to implement the necessary workflows to reach an orchestration goal. An example is coupling schedules for a meeting room with the Smart Device in the room to configure the HVAC systems, the lighting, or the network automatically based on the information from the calendar.

Heterogeneity in the interfaces of Smart Devices makes Smart Space Orchestration complex [SHB10, CD07, MRA10, EG01]. It is the main reason for introducing pervasive computing middleware (Sec. 4.5). This section analyzes reasons for heterogeneity, classifies heterogeneity, and presents existing solutions to overcome heterogeneity.

3.3.1 Inherency of Heterogeneity

Heterogeneity makes Smart Space Orchestration complex [SHB10, CD07, MRA10, EG01]. It is inherent to Smart Devices for different reasons including *market tactics*, *differences between functional domains*, and *technical reasons.*

Market Tactical Heterogeneity

Vendors typically aim preserving their market shares. Incompatibility with competitors forces customers to stick with one vendor if they want their products to interoperate [Pos11, WS97] (*walled garden*).

A typical way to overcome market tactical heterogeneity is standardization. Sec. 3.3.2 discusses difficulties of harmonizing the communication interfaces of Smart Devices via standardization.

Domain-Specific Heterogeneity

Smart Devices cover diverse functional domains. Sec. 3.2 introduced some examples. Different functional domains require different functions to be offered over the interfaces of Smart Devices.

So-called Domain-Specific Languages (DSLs) aim providing an optimal representation for information that is relevant for a specific domain [MH05, CFK$^+$13, LST13, BCG12]. For different DSLs, different communication protocols are optimal.

Therefore it makes sense for vendors to use different communication protocols depending on the domain a Smart Device covers [WS97].

Technology-Specific Heterogeneity

According to the diverse functionalities they offer, Smart Devices have to fulfill different technical requirements. They are installed in heterogeneous environments that make different hardware designs an optimal choice.

For retrofitting existing spaces into Smart Spaces, technology that does not require changes at the cabling of a space is attractive [BLM$^+$11, HWN10, CSG07]. For some environments, energy harvesting Smart Devices that communicate over a wireless link are optimal. They do not require cabling and typically have a long life-time. In other cases, Smart Device plugs that communicate over PLC are the optimal devices to be installed [KNSN05].

Different properties of a communication protocol are optimal for different installation conditions including wireless/ wired, high/ low bandwidth, high/ low latency, or high/ low reliability [PD11, SWW$^+$02, Nyq24, Sha01]. The differing requirements of the used hardware and their installation make it technically necessary to use different communication protocols for different Smart Device hardware.

The mass production of hardware communication chips that support standardized protocols such as IEEE802.15.4 for wireless communication, or Ethernet for wired communication help channeling low-level heterogeneity [KNSN05, Ega05b]. Such communication chips are used by many vendors. The standardization of the physical communication interfaces allows bridging communication protocol heterogeneities in software [GJ86]. Sec. 3.3.4 introduces how messages from distributed Smart Devices can be routed to hosts running adaptation services. Standardization on the protocol layers below the application layer (Fig. 3.3) helps implementing Smart Device adaptation in software (Sec. 8.3.1).

Conclusion

The communication interfaces of Smart Devices are heterogeneous for market tactic, domain-specific, and technology-specific reasons. While the first reason could be overcome with a suitable standardization process (Sec. 7.4), the latter reasons for heterogeneity are inherent to the diversity of functional domains and physical spaces that are covered by pervasive computing scenarios.

For interfacing the full functionality that is available in a functional domain, and to allow the use of the technology that fits best for the installation environment, *the use of a single communication protocol for all Smart Devices is not optimal* (Sec. 3.3, Sec. 4.4.2).

3.3.2 Standardization

Standardization is a process to overcome heterogeneity. It is mainly relevant for this work concerning the diversity of communication protocols that are used by Smart Devices (Sec. 3.2).

The term *standard* has different connotations depending on its use. Technology standards are "*specifications that developers and manufacturers of the technology follow*" [RC96]. This definition targets interoperability [Com96]. To enable interoperability, standards typically *specify interfaces* not implementations [RC96].

Standards "*determine the technology that will implement the Information Society, and consequently the way in which industry, users, consumers and administrations will benefit from it. They play an important role in cooperation and competition between companies, are a key element for the effectiveness of the Single Market and are essential for [...] competitiveness*" [Com96]. This second definition targets the economic power that standards have on markets.

Standards channel research and development in their domain. If patented technology is involved in a standard, the necessary license fee can prevent companies from producing standard conform products. This may keep them out of a market [Grø09, Hil13, YLY05]. Via the described market channeling, standardization has impact on innovations [Grø09]. It fosters the development of interoperable solutions following the standardized directions.

Standards can be classified into [RC96, ITU05]:

- **Formal standards** that are defined by standardization organizations. An example is the Institute of Electrical and Electronics Engineers (IEEE).
- **Public specifications** that are published by consortia. An example is the Internet Engineering Task Force (IETF).
- **De-facto standards** that are typically developed by few (companies) and adapted by many. Operating systems are examples for de-facto standardization.

Standardization is typically driven by companies. Standardization processes are typically not suitable for involving regular users into the standardization activity [JPW98, ITU05].

The creation of a *formal standard* consists of different phases, shown in the standard life cycle in Fig. 3.2 [JPW98, Car95].

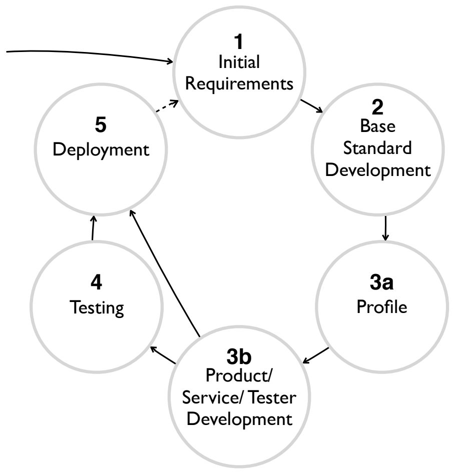

1
Initial Requirements
2
Base Standard Development
3a
Profile
3b
Product/ Service/ Tester Development
4
Testing
5
Deployment

Figure 3.2: A standard life cycle adapted from [JPW98, Car95].

The standardization process can be slow as it has different steps, and as participating bodies –which are typically companies– can have diverging needs, requirements, and experience [JPW98, JPW01]. The standardization process of the ISO / IEC shows the duration of a formal standardization process. It consists of the following five stages. The time of each stage is given in parenthesis [II12]:

- **Proposal Stage**: acceptance of a proposal (3 month).
- **Preparatory Stage**: preparation of a working draft.
- **Committee Stage**: development and acceptance of a committee draft (4-12 month).
- **Enquiry Stage**: development and acceptance of a enquiry draft (3-9 month).
- **Approval Stage**: approval of the final draft (5 month).
- **Publication Stage**: publication of international standard (2 month).

As indicated in parenthesis, each stage takes between two and twelve month. This results in a minimum standardization time of eleven month and a maximum time of 31 month (>2,5y) plus the preparation time for the working draft [II12].

Conclusion

Standardization of technology used in Smart Devices helps reducing the heterogeneity (Sec. 3.2). One standard communication protocol for all Smart Devices is not desired as technical reasons such as protocol expressiveness for a domain (DSL), and adaption to the requirements of an environment require different protocols (Sec. 3.3.1).

Limiting the protocol diversity per $\langle \text{domain}, \text{environment condition} \rangle$ pairs would be desired and useful. The application domains, and the environmental conditions that are relevant for pervasive computing are diverse. Standardization of all combinations would require many standardization processes.

The amount of standardization processes and the amount of participating parties (e.g. vendors) can be expected to make formal standardization a long and difficult process. The low speed of *formal standardization* is likely to prevent innovation in future Smart Spaces (see Sec. 3.2.3).

Formal standardization is unsuitable for future Smart Devices *to the necessary extent.* The standardization of communication technology that is used as building blocks in Smart Devices is important to make the diversity of Smart Devices manageable.

Public specifications are relevant for Smart Devices for defining standard representations for information for instance. Similar to the formal standardization, the diversity of Smart Devices makes it difficult to implement a homogenization at large via *public specifications.*

De-facto standards are unlikely to be successful for Smart Devices as too many vendors produce devices in too many functional domains.

The time and the money that standardization costs can have negative impact on the price of Smart Devices. Higher prices that are caused by standardization can be expected to slow the implementation of pervasive computing in the real world significantly down [RS00]. The lack of user involvement in the standardization processes is unlikely to reflect the diversity of pervasive computing (Sec. 2.5.2) [DM12, Hin99].

Though the existing processes are expected to be usable only to a limited extent, the necessary portability requires a standardization of abstract interfaces on layer 4 of Fig. 3.1 (Sec. 5.6.3):

<R.8> **Logical portability of services** must be supported by offering abstract interfaces to Smart Devices.

3.3.3 Bridging Heterogeneity

Pervasive computing combines functionality from different domains to enable domain-comprehensive orchestration workflows (Sec. 2.4). For technical and functional diversity, and market tactical reasons Smart Device communication protocols are heterogeneous (Sec. 3.3.1). Implementing pervasive computing scenarios requires bridging the communication protocol heterogeneity.

Fig. 3.1 shows different layers that are relevant for applying Smart Space Orchestration to Smart Spaces. There are three possible functional layers in the architecture where heterogeneity could be handled:

- In the orchestration services (layer 3a in Fig. 3.1).
- "In-the-middle" (layer 3 and 4 in Fig. 3.1). The quotation marks denote that this is not necessarily a middleware but neither the orchestration services nor the Smart Devices.
- Within the Smart Devices (layer 2 in Fig. 3.1)

This section discusses advantages and disadvantages of each position. Table 3.1 summarizes advantages on the left and drawbacks on the right.

In the Orchestration Services

Without middleware support, the heterogeneity in Smart Device communication protocols has to be bridged in the orchestration services. This approach is followed by research projects that focus on the implementation of specific use cases [CPW99, BPR99, DAS01] and in DIY Smart Space installations [BLM$^+$11, MH12, TPR$^+$12, RS00].

An *advantage* of this approach is that it prevents unintended information loss. Such loss can happen when using multiple protocols with different expressiveness for transporting information between Smart Devices and adaptation services (Sec. 3.3.4). As the communication protocol of the Smart Device is used in this approach, the described loss does not happen. The Smart Devices remain unchanged in this approach, allowing to use unmodified (off-the-shelf) devices.

The major *disadvantage* of bridging heterogeneity in orchestration services is the lack of portability and extensibility. Only the devices that are preconfigured to an orchestration service can be used[8]. The resulting orchestration service is not portable as it is coupled with specific devices. Supporting fixed Smart Device communication protocol functionality limits the extensibility as Smart Devices that use other communication protocols cannot be interfaced.

Implementing Smart Device adaptation functionality in orchestration services introduces complexity and leads to code redundancy. Complexity is unwanted as it makes the service development difficult. Especially the processing of different protocol semantics results in complex code. It increases the risk of introducing bugs in the

[8]Even if an implementation supports device discovery, which is typically not the case, only devices that were known at compile time of a service can be used.

In the orchestration services (layer 3a n Fig. 3.1)	
+ Controlled information loss (device format used). + Smart Devices remain unchanged (one protocol).	- No portability. - No extensibility. - Complex service code (multiple protocols). - High code redundancy. - No Smart Device access arbitration.
"In-the-middle" (layer 3 and 4 in Fig. 3.1)	
+ Simple services (one protocol). + Simple devices (one protocol). + Low code redundancy in the system (one gateway per protocol can be enough). + Centralized Smart Device access arbitration. (+) Portability (when supported). (+) Extensibility (when supported).	- Possibly transparent information loss.
Within the Smart Devices (layer 2 in Fig. 3.1)	
+ Simple services (one protocol). + Controlled information loss (service format used). + Centralized device access arbitration.	- Complex device software (multiple protocols). - Often infeasible with available resources. - Portability. - Extensibility.

Table 3.1: Advantages and drawbacks of bridging heterogeneity on different functional layers in Fig. 3.1.

program code since more Lines Of Codes (LOCs) are needed to implement a functionality. Code redundancy is introduced as every service that wants to access a Smart Device has to implement corresponding driver functionality[9] [DAS01].

Finally the Smart Device access is not arbitrated. When multiple orchestration services access a Smart Device simultaneously this is likely to cause problems as Smart Devices are typically not designed for concurrent access by multiple services.

"In-the-Middle"

Bridging Smart Device communication protocol heterogeneity can happen "in-the-middle" (layer 3 and 4 in Fig. 3.1) by introducing abstract interfaces to Smart Devices [Sat01, KKR+13, HMEZ+05, MRA10, MTMT06]. This approach allows using unmodified off-the-shelf Smart Devices.

An *advantage* of this approach is that Smart Devices and orchestration services remain simple as they have to use a single interface only. The adaptation is handled in the middle. The same adaptation code can be reused for multiple services. The code can arbiter the access to a Smart Device. Depending on the implementation, this approach can introduce portability and extensibility [KFKC12]. For *portability* it is necessary to provide an instance independent discovery mechanism for Smart Devices (Sec. 5.2.13, <R.8>). For *extensibility* it is necessary to enable adding support for new Smart Devices dynamically (Sec. 5.2.8, <R.43>).

A *drawback* of bridging "in-the-middle" is that a possible information loss (Sec. 3.3.4) is transparent for orchestration services and Smart Devices since it happens transparent to both "in-the-middle" (<R.9>). If the Smart Device's communication protocol is more expressive than that used by a service, the software "in-th-middle" must strip the semantics that cannot be expressed in the other protocol and transparently add it again on communication the other way round. If the stripped semantics are important, the entity in the domain with less expressiveness cannot act in a suitable way.

Within the Smart Devices

The properties of supporting multiple communication protocols within a Smart Device are symmetric to those of supporting heterogeneity in the orchestration services.

It has the *advantage* that orchestration services remain simple. A possible information loss can be handled on the Smart Devices. As the protocol adaptation happens on the Smart Devices, they have all context about the used communication protocols and their restrictions. This enables them to communicate their data in an optimal way. Access to Smart Devices can be arbitrated on the device.

The major *drawback* of bridging protocol heterogeneity on the Smart Devices is that it makes the software on the devices complex. This is not desired as it raises the error potential. It is also often impossible as Smart Devices are typically the most resource limited entities in a Smart Space (Sec. 3.2). Running different protocol stacks in parallel often exceeds a Smart Device's resources.

[9]The use of code libraries simplifies this task but the heterogeneity of such libraries still introduces complexity.

The *portability* (<R.8>) of this approach is limited as Smart Devices can typically only support a limited amount of communication protocols. Supporting new protocols requires updating the code on the Smart Device, which is typically not possible (Sec. 2.6.2). Because of the described limitation of resources and the problems with distributing new code on deployed devices, bridging heterogeneity within Smart Devices is not promising.

Conclusion

Bridging heterogeneity on the edges of the communication process –in the orchestration services or the Smart Devices– allows to control possible information loss that is introduced when translating between information representations with different expressiveness (Sec. 3.3.4). But the additional complexity, the missing portability, and the missing extensibility are major drawbacks.

Bridging heterogeneity "in the middle" only has the major drawback that the information loss happens where least knowledge about the requirements of the application domain and the context of a Smart Device is available.

Because of the discussed advantages, this thesis proposes to bridge communication protocol heterogeneity in layer 3 of Fig. 3.1, and to provide abstract interfaces to Smart Devices on layer 4. To overcome the drawback of information loss that can occur by bridging heterogeneity "in-the-middle", the following requirement is introduced:

<R.9> **Information loss** should be prevented or handled in an explicit way so that the orchestration services are aware of the loss.

Portability of services (<R.8>) and extensibility with adaptation services (<R.43>) for supporting new Smart Devices (<R.23>) –ideally at run time– are desired for Smart Space Orchestration. Both require support by the system architecture [KFKC12, DAS01, DMA$^+$12].

3.3.4 Protocol Conversion

Different Smart Device communication protocols are optimal for different purposes (Sec. 3.3.1). This thesis proposes to connect unmodified Smart Devices via services as shown in Fig. 3.1 on layer 3. This approach has the advantage that existing Smart Devices can be used "out-of-the-box" [Lee06] (Sec. 3.3.3).

This section discusses how Smart Devices can be connected to the adaptation services, and how a conversion between two communication protocols can be implemented.

The ISO/OSI and TCP/IP Protocol Stacks

The problem of connecting computer systems over heterogeneous communication protocols exists since the early days of the Internet [Zim80]. The name Open Systems Interconnection (OSI) for the ISO/ Open Systems Interconnection (OSI) protocol stack refers to openness of the standard. The openness allows all communication systems that follow the standard to interconnect.

The ISO/ OSI protocol stack is the conceptual basis for diverse communication protocols. Its general mechanism for providing interoperability between different communication protocols is introduced. It is relevant for connecting Smart Devices with their adaptation services (Sec. 8.3.1), and to connect the distributed components of the VSL middleware solution in a future Smart Space.

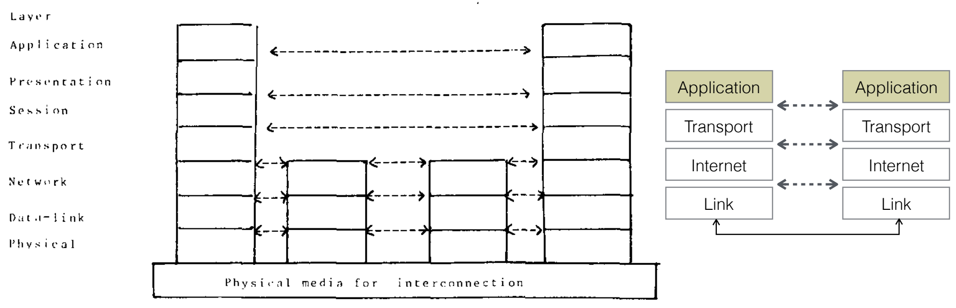


Figure 3.3: Left: the OSI architecture (original figure from [Zim80]); right: the TCP/ IP stack.

Fig. 3.3 shows the ISO/ OSI protocol stack. It introduces the principles of a protocol stack. Each of the shown layers implements different functionality. The interfaces between the layers are standardized. The dashed arrows show logical communication between the corresponding protocol implementations at the sender and the receiver site. The protocol message is identical on both ends of each arrow. The physical communication only happens in the lowest layer [PD11].

The OSI protocol stack [Zim80] is designed in a way that allows simple replacement of protocols in each layer [GJ86]. See Fig. 3.3. This is possible as each layer of the protocol provides full encapsulation which means that it can be regarded as black box by all other layers. This includes that the payload that enters the protocol instance on the sender side will leave the protocol instance on the receiver side of the protocol stack identically [GJ86].

The right side of Fig. 3.3 shows the TCP/ IP stack. This stack is used in the Internet in 2014. It works as described for the left stack only with fewer layers.

The described layered communication is relevant for this thesis as it allows to transparently connect Smart Devices using different communication technologies (Fig. 8.2). The protocol message that is sent by a Smart Device is on the highest layer.

The center of the ISO/ OSI figure (Fig. 3.3 left) shows the processing of the encapsulation of the message on two additional nodes (routing). In this process so-called protocol gateways can be used to transparently exchange the used protocols –e.g. between the two inner nodes. The use of Ethernet on the bottom layer could be exchanged with Token Ring for instance. This makes the connection of Smart Devices that use Internet communication protocols flexible.

Technology for bridging between different protocol standards on the layers below the application layer (top) exist in 2014. Various research exists that covers this problem in detail and provides solutions [Oku86, Lam88, BGRB11, GJ86, SL89]. They can be used for this task.

The VSL middleware that is introduced in this thesis uses IP for communication (Ch. 5). The described existence of technology for transparent bridging between implementations of the different layers in the protocol stack allows a flexible distribution of the VSL components in a Smart Space.

The transparent bridging of heterogeneity in the lower layers of the protocol stack removes the complexity of connecting Smart Devices to their adaptation services (Sec. 8.3.1). The same advantage applies to connecting the distributed components of the VSL (Sec. 6). The property of the bridging that it is transparent for the application layer preserves encryption on the application layer that is introduced in Sec. 6.5.

Syntax and Semantics

The previously described mechanisms can be used to provide connectivity between the application layers (top layer in Fig. 3.3) of two communicating hosts (e.g. Smart Device–adaptation service). Such connectivity is assumed as given in the remainder of this section.

For providing portability (Sec. 5.6.3) it is necessary to provide a common interface on layer 4 in Fig. 3.1 (Sec. 5.3.1). The adaptation services on layer 3 in the figure typically adapt the heterogeneous communication protocols that are used by Smart Devices. From the perspective of a developer of an adaptation service that provides an abstract interface to a Smart Device, the protocol heterogeneity *on the application layer* is relevant. It can be structured into *syntax heterogeneity* and *semantic heterogeneity* (Sec. 8.3.1).

The *syntax* of a protocol message defines how the information it transports is structured [RN95, Hoa78, GJ86, Pos11]. The *semantic* of a protocol message defines the meaning of the transported information representation (Sec. 3.9) [RN95, Hoa78, GJ86, Pos11].

Syntax Conversion

Vendor silo lead to the situation that different vendors transport information from similar sensors in different protocols (Sec. 3.3.1). Such heterogeneity is often syntactic [KNSN05, MH12, BLM^{+}11]). Translating between two communication protocols with different syntax and a common semantic can typically be done without information loss [GJ86, BRa09, BGRB11, GBRB12].

Heterogeneity in communication protocols that is imposed by technical requirements is often structural, concerning the syntax [SWW^{+}02]. In the broader sense, syntax heterogeneity also covers using compression in protocols as this operation can be reversed without loss. Compression can be attractive for saving bandwidth in reliable communication channels. Redundancy can be attractive for decreasing loss in unreliable communication channels.

Semantic Conversion

Different functional domains typically use different semantics (Sec. 3.3.1). Translating between two communication protocols that provide differing expressiveness leads to loss [RN95, Hoa78, GJ86, Pos11, BGRB11, GBRB12]. Two Smart Devices with the same temperature sensor could provide temperature information with different precision over their different protocols for instance. Transforming messages from the protocol that allows to express more precision (high expressiveness) into the protocol that is less expressive introduces loss. Transforming between a protocol that is built for exchanging video streams and a protocol that is built for reading sensor values is typically accompanied by loss, or it is impossible.

A more abstract example are two different DSLs that use different concepts. One DSL could know the concepts {*mother, father, daughter, son*} while the other only knows the concept {*human, animal*}. A translation from the former DSL to the second could be done via using ontologies (Sec. 3.9). It would introduce loss of precision [Mar10, SLP04, BBH+10]. The backwards transformation is impossible without additional information.

Multi-Protocol Conversion

As described before, each protocol conversion possibly introduces loss. This happens when conversion happens between two protocols with varying expressiveness. The approach that is followed in this thesis is to introduce a single common layer of abstraction (layer 4 in Fig. 3.1). This allows to eliminate the information loss (Sec. 5.2.15).

In literature a different approach can be found. It provides interoperability between multiple protocols by implementing multi-protocol conversions. Similar to a "babelfish" [Ada79], each communication partner can communicate in its own communication protocol. Gateways translate pairwise between protocols in a transparent way. Typically multi-protocol gateways act as proxy for all available functionality of one domains in another domain to make the conversion transparent [Che04, LM05, HPJP06, NPD+07].

The major problem with this approach is the transparent introduction of loss (Sec. 3.3.3). Either the functionality that is offered in a foreign domain is limited to the common semantic subset of both domains, or the behavior of control actions that are sent to the proxy are not clearly defined. Such a solution is not suitable for implementing dependable (<R.30>), self-managing (<R.3>) Smart Space Orchestration in future Smart Spaces.

In Sec. 8.3.4, the described proxy concept is used to provide a lossless integration of functionality from one Smart Space into another Smart Space.

Conclusion

Mechanisms to transport protocol messages from Smart Devices to adaptation services over different communication protocols exist. The presented protocol stack (Fig. 3.3) that is used in this work provides IP connectivity between distributed hosts over different underlying protocols and media.

The inherency of Smart Device heterogeneity (Sec. 3.3.1) leads to heterogeneity in the syntax and the semantic of messages that are exchanged between adaptation services and different Smart Devices. Syntactic heterogeneity can typically be bridged in a lossless way while bridging semantic heterogeneity typically introduces loss.

Providing multi-protocol gateways is no solution for implementing Smart Space Orchestration as it either limits the usable functionality, or it introduces transparent loss. The latter violates the requirements of designing a programming abstraction that allows to implementing a dependable (<R.30>) and self-managing (<R.3>) system.

3.3.5 Conclusion

Future Smart Devices use heterogeneous communication protocols for technical and market-tactical reasons (Sec. 3.3.1). The heterogeneity has to be handled by the programming abstraction to provide portability of services (<R.8>).

Standardization of abstract interfaces is required to provide portability:

<R.10> A programming abstraction for Smart Spaces must **support standardization** of abstract interfaces to Smart Space functionality in a suitable way.

Existing standardization methods are not suitable for the Smart Space domain with its high diversity. It is not designed for the high amount of stakeholders that needs to participate and the amount of necessary standardization processes (Sec. 3.3.2).

Heterogeneity is best bridged in the middle between services that implement pervasive computing scenarios, and Smart Devices. The problem that emerges from this approach is transparent information loss.

Mechanisms to connect entities over heterogeneous physical links and communication protocols exist. On the application layer syntax heterogeneity can typically be bridged in a lossless way. Different semantic expressiveness introduces loss when converting between two protocols. Therefore multi-protocol gateways are discarded as solution for future Smart Space Orchestration. Instead a single conversion to a common abstraction (layer 4 in Fig. 3.1) is implemented in this thesis (Ch. 5).

3.4 Middleware as Concept to Overcome Heterogeneity and Distribution

The goal of middleware is providing a unified system view on a distributed system [TVS02, CD05, ACKM04]. A unified system view hides the complexity that comes with the distribution and the heterogeneity of computing nodes that should be connected to a system. To implement a unified system view, middleware provides different functionality including [TVS02]:

- **Access transparency** to hide differences in data representations and the way resources are accessed.
- **Location transparency** to hide the location of a resource.
- **Migration transparency** to hide that resources may move to different locations.
- **Relocation transparency** to hide the relocation of resources that are in use.
- **Replication transparency** to hide if resources are replicated.
- **Concurrency transparency** to hide the shared use of resources.
- **Failure transparency** to hide failure and recovery of resources.
- **Persistence transparency** to hide if resources are in memory or on disk.

Widely used examples for middleware are Common Object Request Broker (CORBA) and Distributed Component Object Model (DCOM) [TVS02, CD05].

Smart Spaces are distributed systems with possibly high heterogeneity (Sec. 3.2). A unified system view facilitates the creation of services for Smart Spaces (<O.0>). Therefore the VSL programming abstraction that is introduced in this thesis provides the described functionality of a middleware (Ch. 5).

3.4.1 Comunication Modes

Communication between distributed nodes is fundamental in distributed systems for providing a unified system view. Services that communicate over a middleware can be coupled in different modes depending on the support of the middleware.

Communication between distributed entities can be structured using the dimensions *temporal* and referential coupling [CLZ00, Tan04]. Table 3.2 gives an overview of different communication modes.

Referential Coupling

Referential coupling describes if the communication partners know each other's identity and location [FHA99]. Entities have a referential coupling when communication addresses the communication partners directly. It requires sharing a name space of identifiers between the communication partners. When the communication is anonymous it is named referentially uncoupled or loosely coupled.

Loose coupling in general means that the identity and location of a communication partner must not be known and that it is not necessary that the communication partners exist at the same time [FHA99].

		Temporal	
		coupled	uncoupled
Referential	coupled	direct e.g. RPC, RMI	mailbox e.g. message queues
	uncoupled	group based e.g. publish-subscribe	generative e.g. tuple space

Table 3.2: Communication modes structured by referential and temporal coupling combining data from [Tan04, CLZ00].

Temporal Coupling

The term temporal coupling describes that communication partners must exist at the same time to communicate. Entities are temporally coupled when their communication is synchronous. They are temporally uncoupled when their communication is asynchronous. From the point of view of a programmer using synchronous coupling results in blocking calls. Using asynchronous coupling of distributed methods results in non-blocking calls.

3.4.2 Interaction Paradigms

The different communication modes are implemented by different interaction methods.

Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a mechanism to invoke procedures on a remote host [BN84]. A RPC typically requires referential synchronous coupling. Procedures get locally invoked on a so-called *client stub*, the parameters get marshalled, sent to the remote host, the procedure gets invoked on the remote host via the *server stub*. The answer gets *marshalled*, sent back, and locally delivered. The remote interaction is transparent for the calling service.

The interfaces of the client and server stubs are typically defined using an Interface Definition Language (IDL) to describe the abstract interfaces [Tan04].

RPCs communication is referentially and temporally coupled (synchronous).

Remote Method Invocation (RMI)

Remote Method Invocation (RMI) is the equivalent to RPC in the Object Oriented Programming (OOP) world. The *client stub* offers the methods of a remote object. The remote object is accessed via the *server stub*. Similar to the RPC case, remotely accessed objects reside on one machine only [Tan04].

RMI communication is referentially and temporally coupled (synchronous).

Message-Passing

In a *message passing* architecture distributed processes communicate by exchanging messages. Instead of RPC or RMI message passing does not directly invoke predefined methods remotely but pass messages between distributed entities. Middleware that uses message passing as communication principle is called Message Oriented Middleware (MOM) [ACKM04].

RPC and RMI are synchronous communication techniques. Remote methods are invoked on remote processes/ objects. The result of an operation is immediately returned to the remote caller. The interfaces that are offered by the stubs are defined on the client and the server. Messages abstract from functional interfaces. They can transport generic content. The use of *message passing* does imply a certain processing on the machine receiving the message.

Message passing implements referential and temporal (synchronous) coupling.

Message-Queuing

The message-passing paradigm can be used for asynchronous interaction by introducing *message queues*. In a MOM with message queues, messages can be queued at the sender and/ or the receiver for asynchronous processing. [ACKM04]

Message queuing implements referentially coupled and temporally uncoupled (asynchronous) communication.

Publish-Subscribe

In a *publish-subscribe* interaction model the communication partners have the roles publisher and subscriber. A publisher publishes information over a bus or a middleware.

Subscriber receive published information based on subscriptions. A subscription is characterized via criteria a published information has to meet. For example in a topic-based publish subscribe system the topic is a criterion subscribers can identify their interest with [ACKM04, TVS06].

A publish-subscribe system implements referentially decoupled synchronous communication. A publisher is unaware of potential receivers. Information is received based on subscriptions by receivers if they are online at the time of its dissemination [EFGK03].

Publish-subscribe implements referentially uncoupled and temporally coupled communication.

Blackboard Systems

A *blackboard* implements a shared memory on a higher level of abstraction. The term was coined in the context of Artificial Intelligence (AI) systems (Sec. 3.6). Participating processes (e.g. software agents) read and write information on a shared data structure that is called blackboard [Nil98, HR88, CLZ00]. A blackboard is used for communication between processes similar to a shared memory in multi-processor systems [TVS02].

Blackboards can be implemented in different ways. Tuple spaces are an example for an implementation of the blackboard communication pattern (see “generative communication” below).

A blackboard implements referentially and temporally uncoupled communication (loose coupling) [Bro91a].

Generative Communication

Generative communication is implemented by so-called *tuple spaces*. By implementing generative communication, processes create communication records, so-called tuples, and store them on a shared space [BFK$^+$11]. A *tuple space* is an implementation of the blackboard communication pattern.

The motivation for tuple spaces was creating a mechanism for *loosely coupled* processes [FHA99]. The loose coupling includes time and space. Tuples are stored on the shared tuple space that exists independently of services. Therefore it is not necessary that producer and consumers of a tuple are running at the same time.

A tuple space offers the functions `out` to put new tuples into a space, `in` to take a tuple from a space, and `read` to obtain a copy of a tuple leaving the tuple in the space [EFGK03]. Services can subscribe to objects by indicating which type of tuple they are waiting for, identified by the types of data it contains. Subscribing services get notified when a tuple matching the previously registered criteria becomes available.

Tuples are stored in a (persistent) tuple space making them available until they get consumed via a service calling the `take` operation. A tuple space is a shared persistent storage between all participating processes.

Tuple spaces are associative storages as tuples are identified by their types and not by their addresses. The tuple space routes tuples between participating entities. Tuple space communication partners do not have to know about each other. Addressing of tuples is based on content. The structure of tuples is stored in so-called templates. Producers produce tuples and put them into the tuple space. Consumers can search tuples based on the templates [FHA99, Gel85].

The tuple space concept was introduced with the Linda system in 1985 [Gel85]. Since then the concept was adapted for different application domains [BFK$^+$11]. Examples are JavaSpaces and Linda In Mobile Environments (LIME). JavaSpaces is a Java implementation of the tuple space concept [FHA99]. LIME [MPR01] takes into account the mobility of nodes. Mobile tuple spaces are merged whenever mobile nodes come into contact.

Generative communication implements referentially and temporally uncoupled communication (loose coupling).

Information Bus

The authors of [OPSS94] describe an *information bus*. It extends the tuple space communication concept by using objects instead of pure data records and type identifiers (subjects) for identifying relevant entries instead of full type signatures. This results in higher performance as only specific object fields have to be considered to handle subscribtions and tuple access.

Subjects describe functional nodes. Information bus objects can inherit from each other. The operations of the information bus are similar to the ones of the tuple space: `get`/ `set`, `publish`/ `subscribe`.

An information bus implements referentially and temporally uncoupled communication (loose coupling).

3.4.3 Two Dimensions of Portability

Middleware is typically running on top of one or many distributed Operating Systems (OSs) providing access to distributed heterogeneous computing resources [TVS02, CD05, ACKM04]. This aspect of middleware is relevant for future Smart Spaces to implement a distributed run time environment for services (Sec. 7.2).

As second use, the middleware concept can be applied on a higher level of abstraction in software orchestrated Smart Spaces. A suitable middleware for Smart Spaces supports providing a unified view on the (distributed) heterogeneous Smart Devices that are used by services to implement pervasive computing scenarios. A unified system view is required to facilitate the software development, and for enabling the independence of service implementations from concrete Smart Device instances (Sec. 4.5).

Both aspects are called **portability**. The former aspect of portability is addressed by the requirements <R.7> and <R.45> (Sec. 6.3.1).

The latter aspect is addressed by requirement <R.8> (Sec. 5.6.3). It is typically implemented by providing abstract interfaces that are used by services to access functionality.

3.4.4 μ-Kernels

The Operating Systems (OSs) that are widely used in 2014 (Windows, Linux, Mac OS X) are typically not regarded as middleware in the classical sense as they do not provide a unified system view on distributed resources [TVS02, CD05, ACKM04]. Instead, middleware is running on top of one or many OSs to provide access to distributed computing resources. But OSs serve the second purpose described in Sec. 3.4.3 by providing unified interfaces to hardware via device drivers. They provide interface portability (<R.8>).

Pervasive computing middleware takes the role of an OS for Smart Space orchestration services. Therefore the acronym of the framework that is presented in this thesis in Ch. 7, Distributed Smart Space Orchestration System (DS2OS), also stands for Distributed Smart Space *Operating System.*

Early operating systems were monolithic, running all OS components like device drivers, and file system in the kernel. An advantage of this approach is that it is efficient by not requiring context switches between kernel mode and user mode for accessing peripheral hardware. A disadvantage of the solution is that it does not scale for adding heterogeneous peripherals. Drivers for all devices have to be included in the fixed monolithic kernel [GG92].

As a solution to the problem, μ-kernels emerged in the early 1980s [RR81, ABB$^+$86]. A μ-kernel only contains the fundamental elements of an OS like Inter Process Communication (IPC), memory management, and scheduling. Device drivers are dynamically added to a μ-kernel at run time.

The advantage of flexible extensibility comes at the drawback of μ-kernels possibly being less resource efficient than monolithic kernels since context switches are necessary between drivers in user space and the kernel that runs in kernel space [Tan01, RR81, ABB$^+$86, GG92].

In modern operating systems, the core functionality of the OS can be extended via libraries [BGJR92, EK95]. DirectX is an example for such a library that extends Microsoft Windows with multimedia functionality.

The state of the art analysis in Sec. 4.5 shows that pervasive computing middleware that is available today follows a monolithic design as it is not extensible (Fig. 4.3). Middleware that is not especially designed for general use –which includes most middleware that was designed for research projects– contains the Smart Device drivers in the (monolithic) middleware kernel.

The core functionality of existing pervasive computing middleware cannot be extended by user-provided services in any of the assessed middleware designs (Sec. 4.5). The VSL programming abstraction that is introduced in this thesis (Sec. 5) enables extensibility of the middleware kernel (Sec. 6.2).

3.4.5 Conclusion

Middleware hides the complexity of a distributed system from the services using it. This is required for a programming abstraction for future Smart Spaces:

<R.11> A suitable programming abstraction for future Smart Spaces must **support distribution** of data sources and of run time environments.

From this section the following transparency requirements emerge:

<R.12> **Access transparency** to hide what is accessed.

<R.13> **Location transparency** to hide the location of an accessed service.

<R.14> **Migration transparency** to hide the migration of an accessed service.

<R.15> **Relocation transparency** to hide that a service is currently getting moved.

<R.16> **Concurrency transparency** to hide the shared use of a service.

<R.17> **Failure transparency** to hide the failure and the recovery of services.

<R.18> **Persistence transparency** to hide if a service operates on memory or on disk.

Concerning the supported communication modes, the diversity of pervasive computing requires the support of different modes (Sec. 5.4.3):

<R.19> **Direct communication support** for time and security critical applications.

<R.20> **Mailbox communication support** for security critical applications.

<R.21> **Publish-subscribe communication support** for time but not security critical applications.

<R.22> **Generative communication support** for time and security uncritical applications.

As monolithic system architectures fail providing enough flexibility to add Smart Device support and to extend the support functionality that is available for Smart Space service programmers:

<R.23> **Dynamic extensibility with device drivers <u>and</u> support functionality** must be supported by the programming abstraction.

Subpart I–B

Top-Down – Facilitating Service Development

3.5 Human Psychology

Chew, if only you could see what I've seen with your eyes!

Blade Runner by Ridley Scott (1982), based on "*Do Androids Dream of Electric Sheep?*" by Philip K. Dick (1968).

The goal of this thesis is providing an intuitive Application Programming Interface (API) for Smart Space Orchestration (<O.0>). To understand what might be intuitive to humans, background in cognitive science is given in this section. It motivates why the chosen design of the programming abstraction for Smart Space Orchestration does not introduce a new layer of complexity, but helps reducing the complexity by offering abstractions that are familiar to humans. Psychology helps identifying a suitable abstraction to represent the physical world in software (layer 4 in Fig. 3.1).

3.5.1 Understanding the Human Mind

Psychology has different theories to explain intelligent behavior. Until 1960 *behaviorism* was the dominant approach to explain intelligent behavior [RN95]. Behaviorists explain the brain with processes that are determined only by their external inputs, the stimulus, and a response.

Since 1960 an *information-processing view* dominates psychology [RN95, JL93]. AI research (Sec. 3.6) helps understanding the functionality of the human mind. Computer programs are written that reason according to hypothesis' of psychologists. The outcome of such emulated brain functionality is compared with the observable behavior of humans. This method helps understanding how the human brain may work [JL93].

Stimulus-response behavior can be emulated with computer programs. So-called *reactive behavior* can be expressed with Event-Condition-Action (ECA) rules. A typical example of *reactive behavior* is a thermostat that reacts to temperature out of a defined range with simple actions (Fig. 3.10).

A concurrent approach to behaviorism is *cognitive psychology.* Cognitive psychology introduces internal reasoning. It can be characterized with the following three steps [RN95]:

1. a stimulus gets translated into an internal representation,
2. internal representations get manipulated by cognitive processes to derive new internal representations,
3. the new internal representations are translated back to actions.

3.5.2 Memory

Basic knowledge about the work of the *human memory* is important for the choice of an intuitive representation of the reality when designing a programming abstraction for Smart Spaces. A representation that is close to the reality humans perceive facilitates the understanding-of, and the interaction-with a programming abstraction [Dét02].

Abstraction seems to be an important tool of the human brain to store and retrieve information [JL93]. Gestalt psychology illustrates how natural it is for humans to match new information to known abstract patterns. As a result of this processing, humans are likely to perceive a (known) *whole* thing instead of its (possibly unknown) parts. The *whole* is an abstract concept compared to the concrete *parts*. An example is shown in Fig. 3.4. The observed phenomenon motivates the *top-down* approach that was introduced in Sec. 3.1.1.

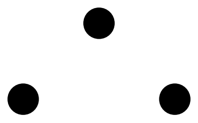


Figure 3.4: Humans are likely perceiving a *whole* (triangle) instead of its *parts* (only three circles are shown).

Frames

Marvin Lee Minsky, Turing award winner 1969, and a co-founder of the Massachusetts Institute of Technology (MIT) AI laboratory, describes human intelligence in his 1974 paper [Min74] with the *frames* concept. *Frames* are the abstract essence of the environment humans perceive. A frame for a room could be that it has four walls, a floor, and a ceiling for instance. Minsky writes, "*A Frame is a collection of questions to be asked about a hypothetical situation: it specifies issues to be raised and methods to be used in dealing with them.*" [Min74].

Human reasoning is described as identifying transitions between *frames*. Actions are identified by associating perceived and abstracted environmental states with the *frames* that are stored in the human brain. Minsky describes in his article that the same process also happens when humans perceive environmental state only *theoretically*, e.g. by reading about it.

3.5.3 Structured Thinking

Structuring perceptions by applying abstraction is natural for humans when thinking. The human language in its spoken and written form structures human thinking. With their work on formal languages, Noam Chomsky and Marcel-Paul Schürzenberger provided an explanation how humans can learn and use languages [Cho56]. Formulating, and thereby structuring thoughts (including tasks), is daily routine for humans. Structuring information is a technique to learn and memorize things [JL93, RN95].

Adapting to new things that are similar to known things is easier than learning entirely new things for humans. Working on abstractions and manipulating symbols are common tasks for humans. Computers process abstract symbols. Such information processing is familiar to humans as the processing in the human brain seems to follow similar patterns [JL93]. Thinking about tasks in an abstract way instead of defining all the details is convenient for humans (Sec. 3.1.1) [Woo09, p.38].

3.5.4 Planning: The Human Decision Process

Newell and Simons founded the theory that people solve problems via *heuristic search* through a problem space [Ohl12, NSS58]. In their theory, so-called *planning* is the identification of paths that lead from an initial state to a goal as shown in Fig. 3.5 [JL93].

The process of *planning* starts with a set of desired properties. It consists of trying to devise a *plan* that leads from an initial state to a state with the desired properties. The ability to *plan* ahead is a fundamental property of human beings [GN87].

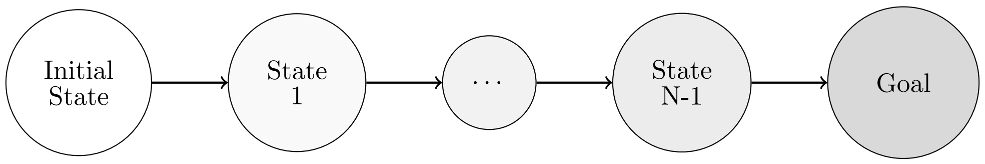


Figure 3.5: A plan to reach a goal according to Newell and Simons, adapted from [JL93].

Possible actions to reach a goal are stored as workflows in the human mind, similar to computer programs. In this analogy, *planning* is the process of selecting the computer program to reach the goal. Applying a concrete initial state to think possible workflows through is similar to configuring the variables of a computer program.

The process of identifying ways to reach the goal (Fig. 3.5) can be deductive or inductive. It is based on known things (Sec. 3.1.1). Humans solve problems by correlating new problems to known mental images and remembered actions [Pri97]. Frames (Sec. 3.5.2) are a concept for such mental images.

The decision process of Newell and Simons has three components, the *mental representation* of the problem, the *goal*, and a *set of actions*. When deciding about which actions to take, humans evaluate different solution paths withdrawing paths that do not lead to the goal and weighting the successful alternatives. Humans are able to think through different alternatives without actively doing them and can decide about their strategy by evaluating the projected outcomes (Sec 3.5.2) [Ohl12].

If a chain of planned actions (Fig. 3.5) cannot be carried out, e.g. as things such as physical tools are missing, sub-plans can be introduced. A sub-plan could be to acquire a missing thing, e.g. getting a physical tool like a hammer. After finishing a sub-plan, the original plan is continued [JL93]. Complex human behavior can be described as goal-driven and hierarchically organized [Ohl12, Woo09].

3.5.5 Conclusion

Computer programs and the way the human brain works are closely related. Many of the observed behaviors of humans can be modeled with computer programs. Unconsciously, humans automatically seem to do something similar to creating and running programs on their minds. As a result, Smart Space Orchestration as process of implementing software to orchestrate Smart Spaces can be intuitive to humans when the used abstraction is close to manual orchestration of a space.

Humans plan actions by setting a goal and identifying a path to reach that goal from a set of actions they know or can deduce. Humans are able to plan things in thoughts without physical input. Having an abstraction that is close to human perceivable reality helps humans to act on the abstraction instead of acting in the real world which is done when writing software.

As shown in Fig. 3.5 the steps towards a goal are states. Abstraction is a basic tool humans use to memorize. *Frames* are a concept for explaining how humans may remember states. A usual way for humans to access knowledge is association. Humans associate things they perceive with things they know already to activate their knowledge about them [JL93]. See Fig. 3.4.

For designing an *intuitive* programming abstraction it should

<R.24> offer a **descriptive knowledge representation** that represents states (e.g. *frames*) not procedural knowledge (actions).

The human mind automatically creates abstractions of different levels. To correspond with this property, a programming abstraction for Smart Spaces should:

<R.25> Foster **interaction** between services on **different levels of abstraction**.

Humans solve problems by combining known activities to reach a superior goal. Humans are used to build complex things out of less complex parts [JL93]. This leads to the requirement:

<R.26> To reflect human problem solving, a suitable programming abstraction should support **modularization** to allow partitioning complex problems into smaller building blocks [Par72].

Writing programs is conceptually close to the human thinking process [JL93]. With the right tools it is easy for humans to learn programming [Ahl76, Can76, Gol76, KG77, Cho56, Kay96, Pap05]. Providing interfaces to various programming languages and programming models (e.g. visual programming [Kay90, Kay96, WHI97]) facilitates the creation of Smart Space Orchestration services. Therefore the developed programming abstraction should:

<R.27> **Support multiple programming languages** for the implementation of services.

This requirement is extended in Sec. 3.10.5.

3.6 Artificial Intelligence

You wouldn't know that because I didn't, when I created you.

Tron Legacy, written by Adam Horowitz, Edward Kitsis (2010).

Artificial Intelligence (AI) tries to understand intelligent behavior, and to make artifacts behave as-intelligent-as, or more intelligent than humans [Nil98, Bro91b]. AI has touching points with different scientific fields. It combines understandings from philosophy, mathematics, psychology, and computer engineering. It overlaps with linguistics in computational linguistics and natural language processing [RN95].

Sec. 3.5 introduced how AI helps understanding intelligent human behavior. This section focuses on the creation of programs that implement intelligent behavior. The given background in AI is intended to provide basic understanding about the feasibility-of, and the requirements-for an implementation of autonomous functionality (Sec. 3.7). Autonomous functionality is important for Smart Space Orchestration as it supports developing manageable and dependable software systems (<R.3>, <R.30>).

A suitable programming abstraction for Smart Space Orchestration targets providing an abstraction on top of the Smart Device hardware that is used as interface between software and the physical world (layer 4 in Fig. 3.1). A background in AI helps designing the programming abstraction in a way that facilitates the creation of intelligent services without introducing unnecessary limitations (<O.3>).

Following, models for implementing intelligent functionality are presented.

3.6.1 Intelligent Behavior

Intelligent behavior includes perception, reasoning, learning, communicating, and acting in complex environments [Nil98, Bro91b]. A **top-down** approach to make artifacts intelligent is based on Newell and Simons theory [NSS58] that intelligent behavior can be modeled by writing computer programs. Knowledge about a domain is represented as *declarative sentences* (compare to *frames* in Sec. 3.5.2). *Declarative sentences* represent knowledge explicitly in contrast to *procedural knowledge* that represents knowledge implicitly stored in programs [Nil90] (<R.24>).

Intelligent behavior emerges via reasoning over knowledge [McC58]. Intelligence is decomposed into functional information processing modules that are combined to implement complex behavior [Bro90] (<R.26>). Situated automata are a method for creating low-level circuitry from high-level specifications [Nil98, KR90]. See the top and center part of Fig. 3.6.

A **bottom-up** approach tries building elementary functions of intelligent life forms into machines. A basic building block is sensing of signals [Nil98, Bro91b]. In his robots, Brooks builds functional units for basic behaviors like `forward`, or `turn` and composes them to create intelligent behavior [Bro86]. Intelligence emerges by connecting different behaviors and self-evolution of systems [Bro90]. Via subsumption of different Finite State Machine (FSM) complex behavior can be implemented [Bro90, Bro91b]. This leads to the requirement that composition should be supported to allow the creation of complex functionality from less complex components (<R.26>).

Neuronal networks are a method to implement a (possibly self-learning) evolutionary bottom-up approach [Nil98]. See top and center of Fig. 3.6.

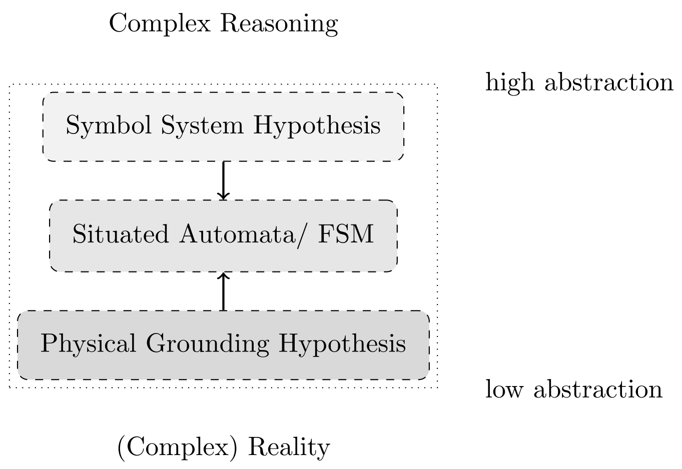


Figure 3.6: Different approaches to implement AI in order of their degree of abstraction from the physical world [Nil90, Bro90].

Fig. 3.6 shows the top-down and the bottom-up approach. Their level of abstraction over a sensed information is indicated on the right. Both approaches use situated automata [RK90]/ FSMs as tool to implement intelligent functionality in programs. Support for FSM in a programming abstraction facilitates the implementation of intelligent behavior in future Smart Spaces. Such support is provided by offering state-based context management (<R.5>).

3.6.2 Reasoning

Acquiring context about an environment representing its current state is an important task for AI. The process of giving meaning to acquired information is called *reasoning* [Kom00]. Methodology for reasoning includes the application of neuronal networks, heuristics, or logic on beliefs.

Reasoning can be used to provide context on different levels of abstraction (<R.25>). Intelligent agents in AI or Smart Space services can reason about facts that are available in the knowledge base (Sec. 3.9) to create new knowledge (Sec. 8.3.2). This fosters a modular information processing (Sec. 3.6.5, <R.26>).

3.6.3 Believe-Desire-Intention (BDI)

The *Believe-Desire-Intention (BDI)* software model is a model for reasoning agents that was developed in the 1980s [GPP+98]. It corresponds to the human *planning* process (Sec. 3.5.4). The model is relevant for the design of a suitable programming abstraction for Smart Spaces as it describes basic processing steps for implementing intelligent behavior in software. It can be applied for developing services that implement ambient intelligence in future software orchestrated Smart Spaces (Sec. 2.6.3, Sec. 3.7).

Fig. 3.7 shows different processing steps that structure the implementation of intelligent behavior. An *intelligent agent* senses events from the environment and generates *beliefs* that are stored in the *beliefset*. *Beliefs* represent knowledge of the world

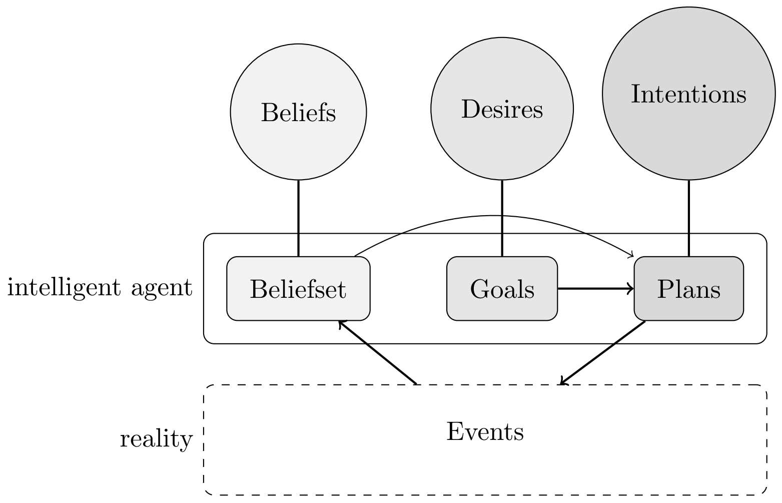


Figure 3.7: Believe-Desire-Intention model.

[GPP$^+$98]. The purpose of an agent is achieving its *desires* that are stored as *goals*. To change the reality so that the *beliefset* and the *goals* match, an agent has *intentions* that result in *plans* to be executed. Intelligent agents *deliberate* by deciding which state they want to achieve and store this *intention* [Woo09, p.66]. The execution of the *plan* that helps reaching the intention generates events that can be sensed and the control loop begins again (Sec. 3.7).

The BDI model is criticized that it does not fit well for systems that must learn and adapt their behavior [GPP$^+$98] dynamically. However many tasks in future Smart Spaces can be expected to be static enough for applying the BDI model (Sec. 3.7).

The BDI model can be used to identify, which information is necessary for implementing autonomous behavior in software. Following the analysis of the functionality of the human memory (Sec. 3.5.2), it shows that necessary context can be stored as state (*declarative knowledge*) [GPP$^+$98]. Supporting the storing and exchange of state facilitates the creation of intelligent behavior as desired in this thesis (<O.0>). This is a relevant insight for the design of the abstraction on layer 4 in Fig. 3.1 (<R.24>).

3.6.4 Planning

Plans are the manifestation of workflows to reach certain goals. Services that implement planning tasks are called *orchestration services* in this thesis. In AI the process of planning and executing a plan can be described as shown in Fig. 3.8.

The input to a planning process is an expected current initial state σ, a goal designator ρ, a set of possible actions Γ, and a decision function Ω that correlates the former three and decides which action γ to take. The action γ is sent to an executor that tries to transform the currently observed initial state of the real world i into the desired state g [GN87].

<R.5> To support the implementation of artificial intelligence, the VSL programming abstraction should support **virtual state management**, also-called **context management** for facilitating context reasoning (Sec. 2.6.4).

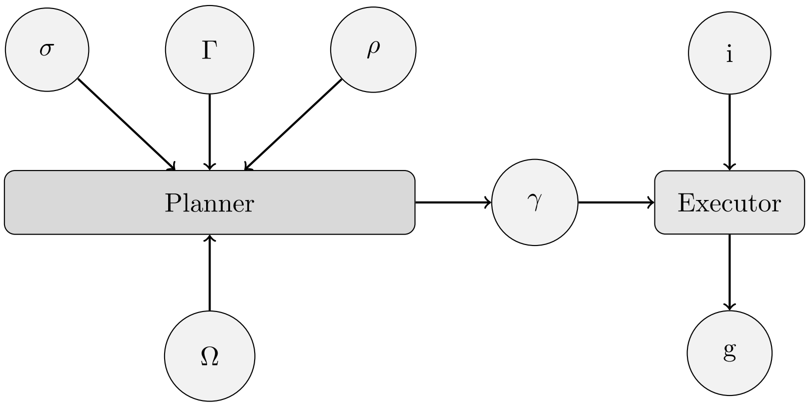


Figure 3.8: Planning and Execution in AI [GN87].

The representation of state is always an abstraction. It is a fundamental challenge finding a context representation that is expressive enough not to limit the possibilities of software to implement plans [Bro91a].

<R.28> The context representation of the VSL programming abstraction must have **high expressiveness**.

At the same time the expressive context abstraction should be simple-to-use (<O.4>) [Bro91a]:

<R.29> The VSL programming abstraction must have a **simple-to-use context abstraction**.

Reacting appropriately to dynamic changes with plans is difficult [AC90]. The challenges come with the complexity of entities and the amount of possible operations [KR90]. Sec. 8.3.1 shows how the presented architecture can be used to connect Smart Devices autonomously with their representation in a knowledge base (Sec. 3.9).

Reactive Behavior

Reactive behavior is a simple form of intelligent behavior. It implements the behavioristic viewpoint of actions that depends on available input at a certain time (Sec. 3.5.1). Policies are a representation of reactive behavior. Fig. 3.9 shows the components of an ECA policy. The *event set* defines the necessary input of a *policy rule*. The *condition set* defines, which criteria the *event set* must match so that the *policy* applies. The *action set* defines the *plans* to be executed as *reactive behavior* [DBB$^+$88, GJ91].

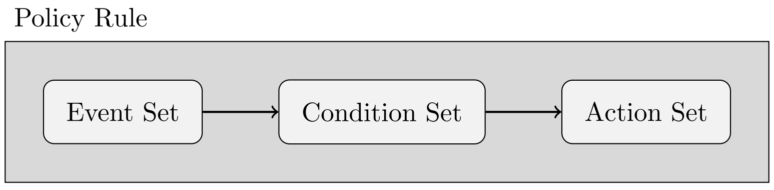


Figure 3.9: Components of a policy.

Advantages of *reactive behavior* are simplicity, low computational complexity, robustness against failure, and elegance [Woo09]. This makes a reactive design attractive not only for beginning programmers but also for experts as it is simple to implement, which lowers the risk of programming errors.

Purely reactive designs have several limitations [Woo09]:

- Being stateless, purely reactive services *lack context* (<R.5>).
- Reactive services typically take only local information into account as it is difficult to obtain non-local information only by applying rules (<R.13>).
- A methodology to describe and define interactions between services is missing (<R.24>, <R.32>).
- The dynamics of multiple layers of reactive services that interact with each other becomes quickly complex and hard to understand (<R.26>, <R.36>, <R.32>, <R.34>, <R.24>).

If the identified limitations could be overcome, reactive services would fulfill the properties of being *simple-to-use*, and at the same time powerful. This would make them a suitable candidate architecture for Smart Space services (<O.0>). The mapping to challenges in the parentheses shows that a programming abstraction that fulfills the requirements, identified in this thesis, would allow to model complex behavior with simple reactive services.

The most challenging points are the methodology to describe and define service interactions, and the complexity of multi-layer service composition. Both are solved by the use of context models as abstract service interfaces in this work (Sec. 5.4.1). Sec. 8.4 presents an example that shows how the introduced VSL programming abstraction overcomes the limitations.

3.6.5 Modular Design

A typical method when designing AI is decomposing problems into sub problems to reduce the complexity of planning tasks [KR90] (<R.26>). The resulting modules that implement intelligent behavior can be distributed (Sec. 3.4) [Gas88, Woo09].

A separation of *perception* from *action* is a standard use case for such a decomposition [KR90]. Sec. 3.7 introduces such a separation more in detail.

3.6.6 Conclusion

The methodology and many results of the AI research can directly be applied for creating of services for Smart Spaces. A suitable tool for implementing intelligent behavior are automata. They are the common denominator of the top-down approach and the bottom-up approach in AI. Reasoning via automata is used to generate higher level knowledge (Sec. 8.3.2, Sec. 3.9).

The BDI model provides an architecture to structure autonomous behavior. It is an artificial implementation of the human planing process (Sec. 3.5.4). As shown in Fig. 3.7, the different stages of the processing need context. A suitable programming abstraction for implementing intelligent behavior should support developers in acquiring, storing, and processing context (<R.5>).

Reactive behavior can be implemented via ECA rules following the BDI model. It implements simple autonomous behavior. Modeling functionality for Smart Space Orchestration via reactive behavior, expressed in rules, is desirable as it is simple. The identified limitations of ECA rules make reactive behavior often unsuitable for implementing complex tasks. This thesis shows how the limitations can be overcome by providing an API that enables expressing complex tasks in simple rules (Sec. 8.4).

Modularization of functionality is a method that is used in AI to reduce complexity (<R.26>).

3.7 Autonomic Management

The term *autonomic computing* describes computation algorithms that use autonomy as central paradigm for implementing complex functionality [LTW01]. A combination of autonomous components that follow a common goal is called *Complex Adaptive System (CAS)*. A real-world example for a CAS is a sports team that plays together without permanent advise by a coach. CAS have different properties [LTW01]. They are

- **autonomous** as their individual components act independently without a central controller,
- **emergent** as the behavior that emerges from the interplay of the entities is not predefined in the entities,
- **adaptive**, or context-aware as the individual components can change their behavior based on context changes,
- **self-organizing** as the other behaviors are achieved from within the system and not via external control.

A real world example for a CAS is the Internet.

3.7.1 Self-*

Autonomic management of computer systems was introduced by IBM in 2001 [KC03, GC03, HM08]. *Autonomic management* focuses on applying autonomic computing methods for facilitating the management of complex systems [KC03]. Concerning management, the self-organization (above) is called self-management.

Self-management comprises self-* properties such as [KC03]:

1. **Self-configuration**: Components of a system configure themselves automatically based on high-level goals.
2. **Self-optimization**: A system monitors its own performance and efficiency and permanently tries optimizing them.
3. **Self-healing**: A system detects, diagnoses, and repairs software and hardware problems.
4. **Self-protection**: A system detects attacks and cascading failures, counteracts and tries to predict and *prevent* such situations.

3.7.2 Monitor-Analyze-Plan-Execute (MAPE)

The methods of AI are fundamental for implementing autonomic computing. The autonomic computing implementation of the Believe-Desire-Intention (BDI) model (Sec. 3.6.3, Fig. 3.7) is called Monitor-Analyze-Plan-Execute (MAPE) cycle. Context (Knowledge) is relevant for processing current events. To express this importance, the

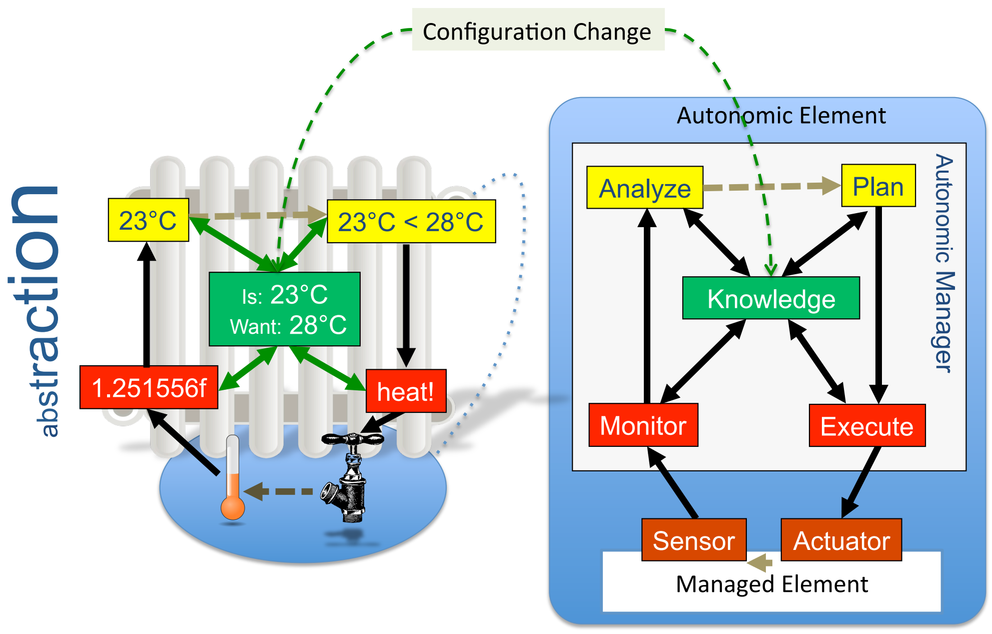


Figure 3.10: Components of a Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) cycle.

acronym is sometimes extended to Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) e [KC03, HM08, MALP12, HMEZ^{+}05]. A Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) cycle is shown in Fig. 3.10.

The figure shows an *autonomic element* that consists of an *autonomic manager* and a *managed element*. A *managed element* typically offers information (sensor) and allows changes (actuator). In case of a Smart Space, a *managed entity* can be a Smart Device that monitors parts of the physical environment via sensors and that can change the physical environment via actuators. The example for a *managed entity* on the left of Fig. 3.10 is a heater. It contains a temperature sensor, and a valve as actuator.

The *autonomic manager* interfaces the *managed entity* via the *Monitor* (M) component for obtaining information and the *Execute* (E) component for changing the state of the *managed entity*. When information is obtained, it is further processed in the *Analyze* (A) module. In the example the raw sensor value is converted into a meaningful format (Sec. 3.6.2).

Based on the *Knowledge* the system *Plans* (P) its actions. Changes are implemented by *Executing* (E) a change request on the *managed element*.

The level of abstraction is typically higher in the upper (yellow) components of Fig. 3.10 as the *Analyze* and the *Plan* modules process, and implement high-level goals that are given to the *autonomic element* from outside (*configuration change* at the top of Fig. 3.10).

The gray sliced arrows indicate that the left and the right side of an autonomic manager are not necessarily coupled. New sensor data can be reflected into the *knowledge base* of an element without the *Planner*. A *Planner* can be triggered by events independent of the *Analyze* module (e.g. external trigger).

External information is relevant for an *autonomic element* to receive high-level goals, and to obtain knowledge from other elements that is relevant for any of the components of the *autonomic manager* (Sec. 8.3.1). The *configuration change* at the top of

Fig. 3.10 shows an interaction with the *autonomic element* from outside by accessing the *knowledge* of the *autonomic manager*. Typical operations for access from remote are obtaining data about the *autonomic element* or changing its high-level goals.

3.7.3 Degrees of Autonomy

The degree of autonomy can be described with the following categories from manual to autonomic [GC03]:

1. **Manual**: Manual administration.
2. **Managed**: Management tools aggregate and pre-evaluate management data. People act.
3. **Predictive**: The management systems monitor and correlate data from distributed system components and gives recommendations. People act.
4. **Adaptive**: The management system monitors and correlates data and acts. People manage and control the compliance with high-level goals.
5. **Autonomic**: All system components are dynamically managed by business rules or policies. People focus on high-level goals.

The last two levels (4, 5) provide flexible adaptivity at low costs, and resilience [GC03]. Flexible adaptivity is relevant for Smart Spaces as Smart Devices can be added, removed, and reconfigured dynamically (Software Orchestration). Resilience is important as Smart Spaces need to be dependable since they provide fundamental interaction with humans in their physical environments (Sec. 2.6.2).

3.7.4 Dependability

Autonomic management was introduced to make complex system dependable [KC03, HM08, MALP12, HMEZ+05]. A so-called dependable system provides different properties, including [AL86, TVS02]

- high **availability**, which describes that a system is available at any point in time with high probability,
- **reliability**, which describes that a system operates without failure, and
- **maintainability**, which describes how simple it is to recover a system after errors; autonomic recovery provides best maintainability (see 5 in Sec. 3.7.3).

Dependability is important for software orchestrated Smart Spaces as Smart Spaces directly or indirectly interfere with humans and other creatures (Sec. 2.6.2) [BCG12, DDMS12, RI10, Chu10, Lee06].

3.7.5 Conclusion

Autonomic management is relevant for the implementation of pervasive computing in the real world. It helps reducing the complexity that emerges from the diversity of hardware and software in Smart Spaces.

Fostering service autonomy (<R.33>) helps facilitating the development of services for Smart Spaces (Sec. 3.8). The MAPE-K loop provides a general architecture for implementing reactive and proactive behavior in autonomous components. Reactive behavior attempts to implement counter measurements when an undesired state is observed. Proactive behavior attempts to prevent undesired states by applying counter measurements before such a state occurs [Str04].

The managed entity and the autonomic manager can be distributed. This is shown in the layers 2 and 3 of Fig. 3.1. Adaptation services that connect Smart Devices with the software domain can be implemented using the MAPE-K structure that is introduced in this section (Sec. 8.3.1).

Autonomic management is relevant for designing a middleware for pervasive computing to implement dependability. Dependability is a fundamental requirement for software orchestrated Smart Spaces. A suitable programming abstraction for Smart Spaces must be dependable, and foster the dependability of Smart Space services. The following requirement emerges:

<R.30> A middleware for pervasive computing must be implemented as **dependable system**.

3.8 Service Oriented Architectures (SOA)

The goal of this thesis is structuring and facilitating the development of software for software-orchestrated Smart Spaces. Sec. 3.5 identified modularization as requirement to reach the goal (<R.26>). A software architecture that implements modularization is a Service Oriented Architecture (SOA) [ACKM04, VVE09, Erl05].

A SOA divides complex functionality into smaller modules that each solve parts of a bigger problem [Erl05]. It provides mechanisms to *automatically* combine independently implemented logical units. Such modules are called *services* in a SOA.

Properties of a SOA are dynamic service discovery, composition, and interoperability [SWS07]. Services are offered and can be used remotely to mash up more complex services by reusing existing functionality.

The following key aspects of service-orientation [Erl05] map directly to requirements for reaching the objective <O.0> of this work:

<R.31> **Loose coupling of services**: loose coupling describes that services are independent of each other, allowing to bind to different services at run time.

<R.32> **Service contracts**: a service contract is an abstract description of the functionality of a service.

<R.33> **Service autonomy**: a service is autonomous when it implements functionality independent of other services.

<R.34> **Service abstraction**: abstraction describes that services hide everything but the *service contracts* from the outside world.

<R.35> **Service reusability**: service reusability describes the support of using one service for implementing multiple more complex services. A SOA fosters reusability by modularizing functionality into services and enforcing abstract interfaces (service contracts).

<R.36> **Service composability**: composability describes the ability to mash complex functionality by coordinating and assembling multiple services (Sec. 3.8.1).

<R.37> **Service statelessness**: statelessness describes that services should ideally not store any activity related state by themselves.

<R.38> **Service discoverability**: discoverability describes that services must be retrievable over discovery mechanisms.

3.8.1 Service Interaction

The terms *orchestration* and *choreography* are used to describe the hierarchical coupling of services in a SOA [PTDL07].

Orchestration

The term *orchestration* describes composing different services into a coherent whole [ACKM04, PTDL07].It describes composing existing services and their workflows to more complex services / workflows in a SOA. The composed orchestration workflows that emerge from the composition can cover multiple domains [Erl05].

A language that is used to represent orchestration workflows is WS-Business Process Execution Language (BPEL).

Choreography

Choreography describes the process of coordinating different orchestration workflows to enable collaboration. It is one level of abstraction above orchestration. A task of choreography is preventing conflicts between orchestration services, e.g. when two services follow opposing goals [Erl05, PTDL07].

Smart Space Orchestration

In a SOA, *orchestration* combines different services. *Choreography* combines different orchestration services.

In Sec. 2.4.1 the term *Smart Space Orchestration* was introduced to describe the orchestration of services in software orchestrated Smart Spaces. The use of the term *orchestration* implies that two management layers are introduced. This thesis shows that Smart Space Orchestration is not limited to two layers. Orchestration services can orchestrate orchestration services that orchestrate orchestration services (e.g. Sec. 8.3.2). Therefore only the term *orchestration* is used for Smart Spaces in this thesis. It covers the aspects of both, orchestration and choreography, from the SOA vocabulary.

3.8.2 Web Services (WS)

Web Services (WSs) implement the SOA principle for Machine-to-Machine (M2M) communication. The WS architecture is standardized by the World Wide Web Consortium (W3C) [BHM$^+$04, ACKM04, VVE09].

Organization Model

The term *Web Service (WS)* is used for a service. Abstract *service interfaces* are specified in the *Web Service Description Language (WSDL)*. For implementing portability abstract service interfaces are published to a *service registry*. An implementation of a *service registry* is *Universal Description Discovery and Integration (UDDI)*. The communication between services happens using the *Service Oriented Architecture Protocol (SOAP)*. Fig. 3.11 shows major architectural components of the WS architecture.

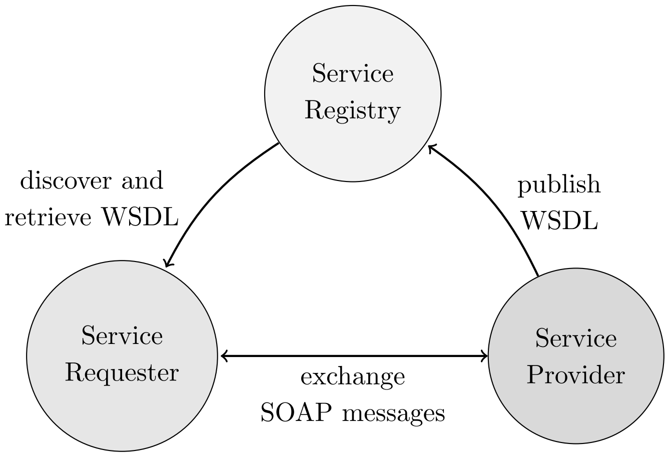


Figure 3.11: Organizational Model of WS; adapted from [Erl05].

Information Model

The interfaces of web services are described in the WSDL using Extensible Markup Language (XML). A WSDL service specification consists of an abstract and a concrete part. The abstract part defines types, messages, operations, and port types. The concrete part defines bindings to concrete protocols and data types, and services and their ports [BHM+04, ACKM04].

XML schema defines the types used in the WSDL and the types and payload for the SOAP messages. It allows to *validate* the content of the abstract interface specifications and the messages which makes the technology more robust [Erl05].

Validation of abstract service interfaces is required in a suitable programming abstraction for Smart Spaces:

<R.39> **Abstract interface validation support** is needed to enhance the robustness of software using a programming abstraction.

Communication Model

SOAP is used for communication between different WSs. SOAP transports XML content. In general the acronym has the meaning Service Oriented Architecture Protocol. In the case of using RPC it also stands for Simple Object Access Protocol (SOAP) [BHM+04].

The structure of SOAP messages is similar to that of Hyper Text Markup Language (HTML) messages. It consists of a header for meta information, and a body for the message payload.

SOAP is specified over multiple protocols including HTTP, Simple Mail Transfer Protocol (SMTP), and File Transfer Protocol (FTP) [BHM+04, ACKM04].

3.8.3 Conclusion

The SOA paradigm helps structuring and facilitating the implementation of complex functionality via modularization. This makes SOA a suitable architecture for a programming abstraction for Smart Spaces (<O.0>).

WSs are a frequently used implementation of the SOA paradigm. WS use function signatures as abstract interfaces. Services expose their individual method signatures in their abstract WSDL interface. The resulting diversity of interfaces introduces complexity for developers. They have to call diverse functions over varying interfaces to reuse and compose existing services. Increased complexity violates requirement <R.1>.

Sec. 3.5 and Sec. 3.6 identified descriptive interfaces to Smart Space functionality as most suitable for Smart Space Orchestration of Smart Spaces (<R.24>). The function-based WS service interfaces do not fulfill this requirement.

No support for context-aware computing (Sec. 2.6.4) is provided by WS.

As described above, the WS architecture provides support for limited protocols for inter-service communication. This limits the possible communication modes that are supported by the WS programming abstraction (<R.19>, <R.21>, <R.20>, <R.22>).

The WS architecture depends on central components such as the UDDI or mail servers for communication over Simple Network Management Protocol (SNMP). This decreases the dependability (<R.30>) and the scalability (<R.50>) of the system.

The identified shortcomings make the WS architecture unsuitable as programming abstraction for future Smart Space Orchestration.

3.9 Context Modeling

As described in Sec. 2.6.4, and further motivated in Sec. 3.5 and Sec. 3.6, context processing is fundamental for implementing Smart Space Orchestration. This section introduces methods to represent context.

A context representation puts real world context into a structured form. This is shown between "*real world object*" and "*VSL context node*" in Fig. 3.12. Providing a structured representation raises the level of abstraction which facilitates the creation of orchestration services.

Context representations can be used to exchange information between services in software orchestrated Smart Spaces. Inter-service communication is implemented as exchange of (structured) context in this thesis. Using standardized context representations as coupling element between services results in service portability (<R.8>). The heterogeneity of instances gets hidden by the standardized context representation they share. Smart Devices that share a common context representation, such as one context interface for different types of lamps, are an example (Fig. 5.5).

Context modeling is the process of designing abstract models for context. So-called *context models* provide an abstract structure for the context a Smart Devices provides. Different kinds of context models exist, including key-value models, markup scheme models, object oriented models, and logic based models such as ontologies [KKR+13, SLP04, BBH+10, BCFF12].

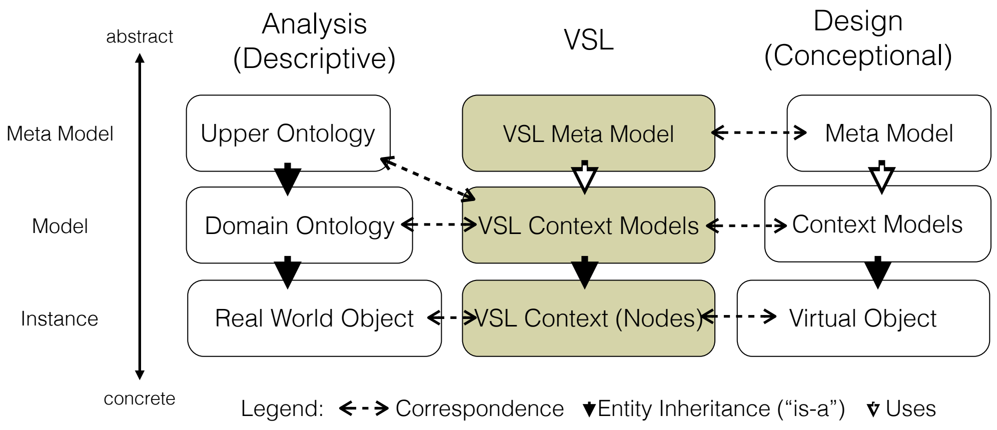


Figure 3.12: Context related terminology; extended graphic based on [AZW06].

Fig. 3.12 puts context-related terminology in relation. The level of abstraction raises from the bottom to the top of the figure as indicated on the left. The low layer contains *instances*, the middle layer *models*, and the upper layer *meta models* that are used for creating models.

The left stack comprises terminology to describe the real world. On the lowest layer are *real world objects*. Their properties can be structured using so-called *domain ontologies* (Sec. 3.9.1). An ontology formally describes the entities that can be present in the real world. *Upper ontologies* contain domain comprehensive concepts. An *upper ontology* may define a "car" as entity with four wheels for instance. An *ontology* created by *car enthusiasts* may add *diverse* specific attributes to the concept "car". An ontology created by *socker fans* may not add such rich details to the upper ontology concept "car" in their domain-specific ontology.

The right stack shows the concepts that are used to represent the descriptions of the left stack. In object oriented terminology the right stack shows the abstract classes and the left stack the instances. The concepts on the right can be used to represent the entities on the left in software.

The conceptual representation of a *real world object* is a *virtual object*. *Domain ontologies* can be described with *context models*. The structure of *context models* is defined by *meta models*.

As shown in Fig. 3.12 upper ontologies and meta models are abstractions of domain models and context models but they do not correspond to each other. *Context models* are used to describe *upper ontologies*.

The middle stack shows the terminology that is used in this thesis. As this thesis introduces a programming abstraction, it provides concepts. Sec. 7.4 shows how the provided concepts can be used to implement a dynamically-extensible crowdsourced ontology for Smart Spaces.

Following, the concept *ontology* is introduced in detail. Then, different meta models for creating context models are assessed in terms of their suitability for modeling context in future Smart Spaces.

3.9.1 Ontologies

The term *ontology* originates in philosophy. It typically describes an explicit specification of a conceptualization [GN87, Gru93, AZW06]. The VSL programming abstraction that is introduced in this thesis (Ch. 5) is based on an ontology that is instantiated as virtual representation of properties of physical Smart Spaces (layer 4 in Fig. 3.1).

Sec. 3.3.3 identified the importance of the *expressiveness* of an abstract interface for Smart Space functionality (<R.28>). Context representations can be categorized into different classes. To understand the meaning of *expressiveness*, different classes and their expressiveness are briefly introduced from the least expressive to the most expressive [RN95, Pos11].

- **Controlled Vocabulary**: A *controlled vocabulary* consists of a collection of terms that have a defined meaning. To describe context with a controlled vocabulary only the terms of the controlled vocabulary are used.
- **Taxonomy**: A *taxonomy* adds the concept of *parent-child relationship* to controlled vocabularies.
- **Thesaurus**: A *thesaurus* extends the concept of the taxonomy with *associations*. Terms cannot only be subordered (taxonomy) but also related in a *thesaurus*.
- **Formal Ontology** *Formal ontologies* allow to represent different relationships between entities. The formal representation of relationships allows to apply formal logic to reason about context (Sec. 3.6.2). The *domain ontology* in Fig. 3.12 is a *formal ontology*.
- **Upper Ontology** An upper ontology is an ontology that describes –typically domain comprehensive– concepts that are used in domain ontologies [AZW06].

3.9.2 Meta Models

A *meta model* defines the structure that is used to create context models. Meta models are *used* by context model designers to create context models. The context models *represent* entities of an ontology. See Fig. 3.12.

Meta models define the expressiveness of context models. Each context model in an ontology uses the same meta model. Only what can be expressed with the meta model can be expressed in an context model.

This section analyzes different meta models according to different criteria based on typical representatives of each class. The selection of the meta models and the selection of the assessment criteria is based on different surveys [KKR$^+$13, SLP04, BBH$^+$10, BCQS07].

The first set of requirements for a suitable meta model is **human related**. It describes the *usability* of a meta model for a human designer of context models:

- *Comprehensibility* describes how simple a meta model makes it for human context model designers to transfer a subset of the real world into its formal representation (context model). The term comprehensibility is used to describe the *complexity of the concepts* offered by a meta model. An example is that a meta model only supports representing hierarchical relationships between modeled entities.
  This is relevant for the *simplicity-to-use* (<R.1>, <R.29>).

- *Complexity* describes how simple it is *technically* for human designers to transfer a subset of the real world into the context model. The complexity is defined by the formal structure of a meta model. As an example, the use of a complex syntax to express a relationship introduces complexity.
  A simple-to-understand concept can be implemented in a technically complex way (Sec. 3.9.6).
  This is relevant for the *simplicity-to-use* (<R.1>, <R.29>).

- *Collaborative design* describes if the context models that form the ontology can be created collaboratively, and shared between context model designers for reuse.
  This is relevant for the *user participation* (<R.4>, <R.48>).

The next set of properties is related to the *expressiveness* of a meta model and the dynamic extensibility (**diversity support**):

- *Supported relations* describe the diversity of relations between real world objects that can be expressed using the meta model (Sec. 3.9.1).
  This is relevant for the *expressiveness* of the abstraction (<R.28>).

- *Timeliness* expresses, if a meta model supports versioning of context, e.g. by offering properties that support versioning of context. Timeliness support is relevant for providing historic context in Smart Space instances.
  It supports *context-aware computing* (<R.5>) by simplifying the provisioning of context versioning with a programming abstraction (<R.1>). It facilitates the context access by providing a unified mechanism to access historic context (<R.29>).

- *Dynamic extensibility* describes the level of support for dynamic extension of context models at run time. This is relevant for software orchestrated Smart Spaces as new Smart Devices can be expected to be added dynamically. Smart Devices that provide context that was formerly not known in a Smart Space requires adding new context models or extending existing context models (at runtime).
  This is relevant for the *dynamic extensibility* (<R.23>).

The last set of assessment criteria is related to *efficient processing* by machines (**machine related**):

- *Reasoning support* describes the level of support, a meta model provides for automated reasoning (Sec. 3.6).
  This is relevant for enabling *autonomous functionality* (<R.3>).
- *Distributed processing* describes if context can be processed by distributed entities simultaneously.
  This is relevant for implementing *distribution support* (<R.11>).
- *Syntax validation* describes if the syntax of a meta model supports automated validation of context models.
  This is relevant for enhancing the *dependability* (<R.30>), the *security* (<R.49>), and the *simplicity-to-use* (<R.1>, <R.29>).
- *Semantic validation* describes if the semantics of a context model can be formally validated.
  This is relevant for enhancing the *dependability* (<R.30>), the *security* (<R.49>), and the *simplicity-to-use* (<R.1>, <R.29>).
- *Efficient parsing* describes how efficient context models that use the meta model can be parsed.
  This is relevant for the *scalability* of the entire system (<R.50>).

Table 3.3 summarizes the following assessment.

3.9.3 Key-Value Models

Key-value models use tuples of keys and values. Programs without explicit data structures typically use key-value pairs in form of regular variables to represent their data. The key is the name of the variable. So-called cookies of a web browser that can be used to store the state of a world wide web connection also use key-value pairs as data structure. Key-value models are use case, and implementation specific. They have no abstract structure that could be used to create context models.

Human Related

Without explicit structure the comprehensibility of key-value models is low. No conceptual representation of the real world exists (-). The complexity of key-value models is low as they consist of simple tuples that are accessed via their key (++). Without explicit structure, key-value models do not support collaborative design (-).

	Key-value	Markup scheme	Object oriented	Logic based	VSL meta model
Human Related:					
– Comprehensibility	-	+	+	++	++
– Low Complexity	++	+	++	-	++
– Collaborative Design	-	(+)	-	-	++
Diversity Support:					
– Supported Relations	-	+	+	++	++
– Timeliness	-	(+)	(+)	(+)	++
– Dynamic Extensibility	-	(+)	-	-	++
Machine Related:					
– Reasoning Support	-	+	+	++	++
– Distributed Processing		(+)	-	(+)	++
– Syntax Validation		++	++	++	++
– Semantic Validation		+	+	++	++
– Efficient Parsing	++	+	+	-	++

Table 3.3: Properties of different meta models [KKR$^+$13, SLP04, BBH$^+$10, BCQS07] compared with the VSL meta model. + = high, - = low, ␣ = significantly depending on implementation, (+) = basic support provided

Diversity Support

Without context models there is no diversity support (-).

Machine Related

As key-value models have no abstract structure, generic reasoning approaches [BBH+10] cannot be applied (-). Distributed processing support depends on the implementation. Typically it is not provided, e.g. local program data. The validation support depends on the specific use case and implementation. Therefore no assessment is done. The parsing of key-value models is highly efficient as they are very simple (++). [KKR+13, SLP04, BBH+10, BCQS07]

3.9.4 Markup Scheme Models

Markup scheme models use hierarchical data structures of markup tags with attributes and values. Markup tags are typically recursively defined by other markup tags. Often subclasses of the Standard Generic Markup Language (SGML) such as XML are used as markup language (listing 3.1). Another example is the Resource Description Framework (RDF) [BG04].

Human Related

SGML based markup schemes are human readable. Their hierarchical structure is well understandable by humans (Sec. 3.5). The support for modeling real world objects is limited as markup schemes typically do not provide real-world related semantics (+). Markup schemes use a –typically simple-to-understand– syntax that can become complex via nesting and support for numerous attributes (+). The loose coupling of models in a markup scheme allows collaborative creation of context models if mechanisms like a Context Model Repository (CMR) are provided. Therefore the plus is only in parenthesis.

Diversity Support

The expressiveness of a concrete markup is implementation specific. In general a markup scheme is flexible concerning the content that gets modeled. This includes the general ability to model any dependencies. As described above, markup schemes support the concept of type relationships (+).

Timeliness support is not provided by the meta model but it is supported via extending the attributes of a markup. If programs implement corresponding semantics, a markup scheme can express timeliness. Therefore the plus is in parenthesis.

For implementing dynamic extensibility, software that uses markup scheme models has to provide support accordingly (+).

Machine Related

Having an explicit representation is the basis for reasoning. Markup schemes support reasoning about type relationships via the meta model (+).

The self-containment of instances of markup scheme based models enables distributed processing, if the software that implements markup-based context models supports it (+).

Markup schemes are designed for syntax (++) validation. Markup Schemes typically support data types and inheritance which allows basic semantic validation (+). With growing size of a scheme the processing efficiency decreases. As only inheritance has to be resolved which happens typically at instantiation time, the overall performance of markup scheme-based context models is high (+) [KKR$^+$13].

3.9.5 Object Oriented Models

Object oriented models represent context as objects. They support object oriented design functionality such as inheritance, encapsulation, and abstract data types. An example is the object representation of context in Java Agent Development Environment (JADE) [BCG07].

Human Related

Object orientation follows principles that are intuitive for humans (Sec. 3.5). The concept does not provide domain-specific support for the creation of context models (+). The intuitive concept facilitates the creation of context models using it. Object orientation has low complexity as concept for context modeling (++). An object instance typically "lives" on one machine. This makes collaborative design of context models difficult (-). The dynamic extensibility via sharing of abstract classes may enable it if support functionality in form of a CMR would be provided.

Diversity Support

Object oriented models support inheritance allowing to express type relationships and composition by including objects into other objects (+).

By creating new objects for new versions of context, timeliness could be implemented. Object orientation enables timeliness but it does not provide specific support functionality (-)

Dynamic extensibility has to be supported by the software that uses object oriented models. Typically the structure of an object is fixed (-).

Machine Related

Like markup-based models, instances of object oriented models provide a defined structure. It enables reasoning. In addition, inheritance is part of object oriented design which allows dependencies to be taken into account when reasoning (+).

Distributed processing of context model instances is typically not possible with object oriented models as instances run on one machine at the same time only (-).

Syntax validity is typically inherently checked when instantiating an object to use it (++). As the inheritance mechanism is part of object oriented design, dependency semantics can be validated (+). As objects are usually handled by object oriented programming languages their processing is efficient in this context (+). [KKR$^+$13, SLP04, BBH$^+$10, BCQS07]

3.9.6 Logic Based Models

Logic based models express relations between entities in a form that allows automated reasoning [ZM03, HIR02, AY03]. An example for logic based models are formal ontologies (see below).

Human Related

Logic is a familiar concept for humans (Sec. 3.5). The human brain builds structures similar to ontologies to model the reality and its relationships. Therefore they are well usable to represent entities of the physical world (++).

To design logic based models, designers have to follow formalisms that define a specific ontology. Those formalisms are typically complex (listing 3.1). Ontologies represent different relations between entities. Those relationships are also called predicates. Via its meta model, an ontology introduces predefined categories for relationships that are difficult to extend. The complexity of ontologies increases quickly when many aspects are to be modeled. Typically experts are required to create ontologies (-) [KKR$^+$13].

Collaboration support for the design of logic based abstractions is an open research topic [HJ02, KA06, LST13]. A major problem is the convergence to a common model (-) (Sec. 3.3.2, Sec. 5.6.4).

Diversity Support

Logic based models typically support different concepts for expressing dependencies such as inheritance, association, or location (++).

Logic based modelings enables timeliness as it typically uses markup for representing context but it does not provide specific support functionality for context versioning (+).

The dynamic extensibility of logic based models typically requires exchanging the entire model, e.g. the entire ontology (e.g. Sec. 4.5.9). Adding new context models at run time is typically not possible. The context models are typically defined before instances are created, and their structure cannot be changed at run time. Logic based models do typically not support dynamic changes at run time (-).

Machine Related

The foundation in formal logic provides the best reasoning support of all assessed meta models. Reasoning can be implemented by applying formal logic such as the λ-calculus [Lan64] (++).

The low dynamics, and the being well-defined of logical expressions support distributed processing (+). The parenthesis are added as an implementation must support such distributed access.

The syntactic and semantic validity of logic formulas can be proved (++). The complexity of relationships that can be mapped into ontologies makes the parsing of ontologies complex. Dependencies typically have to be resolved at run time (-). The complexity of real world ontologies makes their processing by machines inefficient [KKR+13, SLP04, BBH+10, BCQS07].

Web Ontology Language

An example for ontology based models is the Web Ontology Language (OWL) [HPSvH03, ZM03, ZM03, HIR02, AY03]. OWL is standardized by the W3C. OWL can be represented using RDF or XML [ACKM04].

An example[10] for the XML representation of an ontology in OWL is shown in listing 3.1.

```
<owlx:Class owlx:name="WineDescriptor" owlx:complete="false" />

<owlx:Class owlx:name="WineColor" owlx:complete="false">
  <owlx:Class owlx:name="#WineDescriptor" />
</owlx:Class>

<owlx:ObjectProperty owlx:name="hasWineDescriptor">
  <owlx:domain owlx:class="Wine" />
  <owlx:range owlx:class="WineDescriptor" />
</owlx:ObjectProperty>

<owlx:ObjectProperty owlx:name="hasColor">
  <owlx:range owlx:class="WineColor" />
</owlx:ObjectProperty>

<owlx:SubPropertyOf owlx:sub="hasColor">
  <owlx:ObjectProperty owlx:name="hasWineDescriptor" />
</owlx:SubPropertyOf>
```

Listing 3.1: Example representation of an entity in OWL XML.

Typically, tools such as graphical editors are used to interact with ontologies. Listing 3.1 shows that the representation of an ontology is complex. This makes the use of OWL by non-experts difficult which conflicts with the real world usability that objective <O.4> demands.

VSL

The VSL meta model is introduced in this thesis. Its properties are discussed in detail in Sec. 5.2.19.

3.9.7 Conclusion

Context modeling is a fundamental task for context-aware computing. A meta model defines the expressiveness of context models. Context models are used to represent domain ontologies.

[10]http://www.w3.org/TR/owl-xmlsyntax/apd-example.html

Key-value models are simple-to-understand. *Markup models* represent hierarchies and type-relationships. They introduce syntax validation. *Object oriented models* support inheritance. Before storing a new value into a field of an object, a programming language typically validates given values according to the defined types of the corresponding fields in the object.

Ontologies provide most support for implementing intelligent behavior. The complexity, the missing support for collaborative design, the lack of dynamic extensibility, and the parsing inefficiency for large ontologies make them unsuitable as meta model for future Smart Space Orchestration.

The analysis of requirements for a suitable meta model shows that none of the assessed approaches is suitable as meta model for future Smart Space Software Orchestration:

<R.40> A **meta model with suitable properties** as shown in table 3.3 is required to support scalable, and intuitively understandable and usable context-aware computing.

The major points are:

<R.41> **Low complexity** in the meta model is required for enabling user-based development.

<R.42> **Collaboration support** is required to support user-based development, for fostering user involvement, and to distribute the workload that emerges when all Smart Devices of future Smart Spaces need context models.

<R.43> **Dynamic extensibility** of an ontology with context models at run time must be supported to enable adding new Smart Devices.

<R.44> **Context validation** is required to minimize the error potential, enhancing the dependability of Smart Space Orchestration.

This thesis introduces a novel meta model that implements a hybrid approach of the presented meta models (Sec. 5.2).

Subpart I–C

Case Study – App Economy

3.10 Use Case: App Economy

Smartphones are an important factor for making mobile computing ubiquitous [ITU13, ITU11, Cis12, MTC$^+$13, CD07]. As described in Sec. 2.4.4, smartphones fostered the commercial success of wireless broadband, and the sales of mobile devices. In both areas the commercial success fostered innovation.

While at the beginning, mobile Internet was the major success factor for smartphones [WM10], the availability of *Apps* is becoming an important selling point for smartphones [vE13, VSCK11]. Apps are the software services of smartphones. They can be deployed dynamically to smartphones and add functionality to existing devices.

The term *App economy* is used to describe the technical mechanisms and the commercial ecosystem that emerged around Apps for smartphones since 2008 [VSCK11, GWB10, LY10, Kim10]. Fig. 3.13 shows the building blocks of the *App economy* [TTP11, Kim10, GWB10, TH10, MG10].

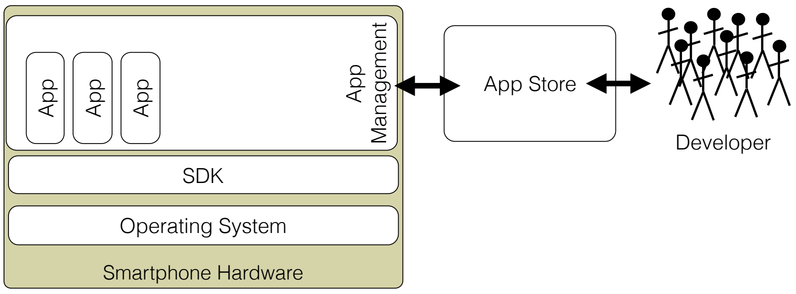


Figure 3.13: Building blocks of the App economy.

The left side of Fig. 3.13 shows a smartphone. The smartphone runs an Operating System (OS) that offers abstract interfaces to its peripherals (Sec. 3.10.1), and to common software components such as the address book. Using fixed interfaces provides portability (Sec. 3.10.3). Software developers are supported by so-called Software Development Kits (SDKs) that facilitate the access to OS functionality (Sec. 3.10.5). The SDK is used to develop Apps.

A central component of the *App economy* is the *App store* (Sec. 3.10.4). It provides diverse functionality including a software repository that is used to distribute Apps. Each smartphone has a locally running instance of the App store. It manages the life-cycle of an App in cooperation with the smartphone OS.

As shown on the right, Apps are developed by distributed developers in the App economy [TTP11, Kim10, GWB10].

The presentation in this section follows a bottom-up approach starting with the hardware of, and the software on the device. Then the App store and different development strategies are presented.

3.10.1 Smartphone Hardware

Smartphones are Smart Devices. Their sensors include an accelerometer, a compass, a microphone, and a camera. Their actuators include a flash, a loudspeaker, and a monitor. Additional peripherals can be connected over wired or wireless connectors. Smartphones have a network interface and can be controlled remotely with the corresponding Apps.

Like Smart Devices, Smartphones hardware changes from specialized hardware (e.g. main processors) towards standard PC hardware [Wan09, VSCK11, SWG13, PK10]. This enables running standard PC programs on smartphones as the same CISC is used (Sec. 3.2.4).

3.10.2 Analogy Smartphone – Smart Space

Smartphones are chosen for this case study for two reasons. First, they are Smart Devices that can be reconfigured via software. Second, they are spatially limited Smart Spaces themselves. The first aspect is described in Sec. 3.10.1.

Considering each peripheral (Sec. 3.10.1) as heterogeneous Smart Device, a smartphone is a *physical space that contains Smart Devices* – a Smart Space (Sec. 2.4.1). The Smart Devices are connected to the smartphone OS that orchestrates them via Apps. A smartphone implements Smart Space Orchestration.

The analogy of smartphones being Smart Spaces explains why it is relevant looking at the successful development of mobile computing from a smartphone perspective to identify possible strategies for Smart Spaces. Smart Spaces are at a similar evolutionary step smartphones were at the introduction of the Apple iPhone in 2007 [WM10]: smartphone hardware was available before, only the enabler for software development and the software were missing. Sec. 2.4 describes the same situation for Smart Spaces in 2014.

3.10.3 Smartphone Operating Systems

Like an operating system of a PC, the OS of a smartphone provides *execution-platform portability* between different hardware platforms running the operating system (<R.45>). All Apps that are programmed for one OS can typically run on all systems that run the OS [Tan01].

Smartphone OSs provide standardized interfaces to the peripherals that are described in Sec. 3.10.1 [Goo]. This results in *interface portability* of heterogeneous peripheral hardware such as camera modules from different vendors (<R.8>).

There are three major operating systems for smartphones in 2014, Apple iOS, Google Android, and Microsoft Windows Phone 7 in order of their popularity for developers [VSCK11]. Table. 3.4 shows properties of the three systems that are most relevant for this work. The format of the *execution binary* indicates if an App can run on heterogeneous hardware. The *programming language*, the availability of an *Integrated Development Environment (IDE)*, and the availability of an *SDK* indicate the developer support that the platform offers.

Apple iOS runs native binaries. It uses Objective-C as programming language. A SDK and the IDE Xcode are provided.

Google Android runs Java bytecode. It uses Java as programming language. A SDK and the IDE Eclipse are provided.

Microsoft Windows Phone 7 runs .net bytecode. It can be created using any .net programming language. A SDK and the IDE Visual Studio are provided.

	Execution Binary	Programming Lang.	IDE	SDK
Apple iOS	native	Objective-C	XCode	✓
Google Android	Java	Java	Eclipse	✓
Windows Phone 7	.net	.net-*	Visual Studio	✓
DS2OS	Java	Java	Eclipse	✓

Table 3.4: Properties of different smartphone operating systems.

For comparison, the last row shows the properties of the DS2OS framework that implements the technical components of an App economy for Smart Spaces (Ch. 7).

Looking at table 3.4 shows that all operating systems that run on heterogeneous hardware platforms use interpreted bytecode for execution binaries [CGR11, VSCK11]. The advantage of interpreted code is that the execution binaries can run on heterogeneous hardware platforms without change, only by adapting the interpreter. This advantage comes at the price of reduced performance [GM95]. Apple's hardware platform is homogeneous as Apple controls it. This allows using native binaries that do not have to be interpreted for execution.

The hardware in future Smart Spaces is likely to be heterogeneous (Sec. 3.2). To enable the sharing of Smart Space services,

<R.45> **Portable service executables** are required to enable running the same binary on the heterogeneous hardware platforms of future Smart Spaces.

The choice of the programming language determines the comfort that developers have when developing Apps for a platform. All programming languages in table 3.4 are suitable for all levels of programmer from beginners to experts [GM95, GR11, MG10]. The third column shows that all solutions facilitate programming by offering IDEs. The last column shows that all solutions support programmers with a SDK (Sec. 3.10.5) that typically consists of support material such as development templates (Sec. 8.2), and documentation [MG10, PMC$^+$12, VSCK11].

3.10.4 App Store

An *App store* acts as *software repository* and *market platform* for Apps. In the described platforms (Sec. 3.10.3), the App store has an interface that is running locally on a smartphone. This interface is used to download Apps from the repository (Fig. 7.5) [VSCK11, GWB10, LY10, Kim10].

The most successful App store in 2014 is the Apple App store that was opened in 2008, followed by the Google Play Store that opened in 2010 [SZWN11, TTP11]. Within the first half year after opening, 300 million Apps had been downloaded [LY10]. Until

October 2013, 60 billion downloads from the Apple App store happened and more than 1 million Apps are available [11]

This shows that adding features via software is attractive for users. In 2014, the availability of Apps is a decisive factor for a smartphone [Kim10, vEF12, LY10, Ant11, TTP11, Eri12]. For making pervasive computing ubiquitous, a distribution mechanism for Smart Space services is required:

<R.46> An **App store with Simple access** is required to enable real world pervasive computing.

3.10.5 Software Development Kit (SDK)

So-called *Software Development Kit (SDK)* of smartphone OSs provide developers with tools that enable and facilitate the development of software for the supported platform. An SDK typically includes [MG10, GR11, TTP11, GWB10, LY10]:

- API *libraries* that allow accessing the hardware and software that is available through the OS.
- An *emulator* that allows testing software without having to install it on the hardware (e.g. smartphone).
- *Example programs*, and supportive material for learning the development for the platform.

The API libraries are required to implement *portable* access to the smartphone peripherals (<R.8>). The emulator facilitates the development. It allows testing and debugging software before deploying it to the target platform. An emulator typically offers better debugging support than the target device [TH10, MG10, GR11, App13, VSCK11, GWB10].

The diversity of Smart Devices in real world Smart Spaces makes offering *emulators* important for implementing Smart Space Orchestration. *Emulators* allow developing software without owning a device, facilitate testing, and enable testing of diverse scenarios before deploying a Smart Space service. Emulator support can be expected to enhance the dependability and security of software orchestrated Smart Spaces (Sec. 8.3.3, Sec. 7.4).

<R.47> A suitable programming abstraction for future Smart Spaces should support the testing of services with **emulators**.

As SDKs facilitate the development of software for a platform, requirement <R.27> can be extended:

<R.27> The availability of a **SDK** for user friendly programming languages with an IDE support developers.

[11]http://www.theverge.com/2013/10/22/4866302/apple-announces-1-million-apps-in-the-app-store.

3.10.6 Crowdsourcing

The term crowdsourcing describes a competitive process of cooperative work. The competitive aspects comes via incentives such as direct (sales) and indirect (advertisement) monetary payment, honor, or other forms of perceived rewards. The competition leads to innovation and quality assurance [BCPR09].

App stores allow crowdsourced development. They attract many developers [PMC$^+$12, KK10]. In 2012 more than one million people were developing Apps. The top 1500 developers had an average revenue per App between \$2000 and \$4000 per month [PMC$^+$12].

Different revenue models exist including paid Apps where the store owner typically keeps 30% of the revenue, and free Apps that show advertisements. The advertisements are added via the API of the smartphone OS SDK. The advertisements to be shown are typically provided by the App store operator. App developers are often paid per showed advertisement [PMC$^+$12, Kim10, GWB10, YYC10, LY10].

<R.48> **Crowdsourced development** fosters innovation and can enhance the quality of software via its competitive mechanisms.

3.10.7 Quality Assurance

A major difference between the Apple App store and the Google Play Store is that the Apple store is gated. This means that Apple reviews Apps before publishing them while Google is publishing all Apps and removing malicious Apps when malicious behavior is noticed or reported [But11, App10, SFK$^+$10].

The former results in a higher overall quality while the latter approach scales better. In 2011 no successful mechanisms to provide scalable quality assurance were known [But11, SZWN11]. In Sec. 7.4 a candidate for such a mechanism is proposed that is based on crowdsourcing [BCPR09] and automated feedback collection.

Bug tracking is reported as the biggest challenge by App developers [PMC$^+$12]. For high real world usability bug tracking is required for implementing <R.48>. The solution that is introduced in Sec. 7.4 contains automated feedback mechanisms that foster bug detection. The methods discussed in Sec. 5.5.6 can help identifying and preventing bugs.

3.10.8 Security

The opening of a platform for third party Apps makes it necessary to provide security mechanisms to secure users, software, and devices from malicious or defective Apps.

Sandboxing

Running third party Apps requires mechanisms to secure the OS and Apps from unauthorized access at run time. The same applies between different Apps. As solution for this problem, iOS, Android, and Windows Phone 7 run applications within sandboxes that cage Apps [BBS$^+$10, SM13, SFK$^+$10, App13, App12].

The use of Java provides sandboxing functionality in the DS2OS system that is introduced in Ch. 7.

Code Signing

The installation of third-party Apps makes it necessary to identify Apps before installing them to prevent the installation of unauthorized Apps. Apps signing by the App store ensures that an App went through the security mechanisms of the App store (Sec. 3.10.8), and that it was not altered. For this purpose, Apps are signed by the App stores. The signature of an App is validated automatically by the OS before installing it on a smartphone [SFK$^+$10, App12].

The same mechanism is applied when developers upload their Apps to an App store. A store typically only accepts code that is signed with the developer's private key. The described mechanism ensures that the identity and the integrity of an App can be guaranteed [SM13, App12].

A similar solution for signing services is proposed for Smart Spaces in Sec. 7.3.

Access Control

Smartphones and their Apps collect private data including the user's location, device usage, contacts, and calendar entries. This makes it necessary to protect user privacy by protecting the user's data from unauthorized access. As solution, the smartphone operating systems uses role-based access control and encryption as mechanisms [SM13, App13, App12, NKZ10].

The data access of an App is per default restricted to the data it created. To get more access, the user is asked to give permission based on roles such as "The App wants to access your address book.". Typically if the user grants access via the simple option "yes", the App can always access the address book unless this right is revoked in the phone's settings.

The amount of smartphone OS built-in features, and supported peripherals is limited (Sec. 3.10.1). For managing the access to the known entities, access permissions lists are available in the smartphone UI (Fig. 7.5) that show Apps that are granted access to data from software or hardware components (e.g. Global Positioning System (GPS) location) [NKZ10, App12, SM13]. Sec. 7.3.3 presents how the same mechanism can be used in Smart Spaces supporting their distribution and increased complexity.

For protecting data in case of a hardware theft, data on disk can be encrypted [App12, SFK$^+$10]. Encryption can also be applied for Smart Space data (Sec. 5.5.5).

Security for Smart Spaces

The brief description of the security architecture of the App economy shows that security and privacy features can be successfully enforced for third party Apps when they are built into the OS. As the OS is the execution platform for Apps, it can impose security and privacy features to Apps in a transparent way. The term "*security-by-design*" is used to describe that security is provided from ground up, independent of the mechanisms that are implemented in the software (App) that is using a system.

Smart Spaces contain a diversity (<O.3>) of devices and use cases. They are distributed. Smart Space Orchestration typically happens invisible in the background. Users are not as aware of the ambient orchestration as they are of using Apps on the smartphone, which they hold in the hand. This makes providing security and protecting privacy more challenging in Smart Spaces than in smartphones [CD05]. Smart Spaces will inevitably collect more privacy-relevant data of user's and their environments [MHDM09]. This leads to the requirement:

<R.49> Mechanisms for providing **security-by-design** must be supported by a programming abstraction for Smart Spaces.

3.10.9 Conclusion (Difference Smartphone – Smart Space)

Smartphones and Smart Space are similar in several aspects (Sec. 3.10.2). Therefore, looking at the use case App economy helps understanding relevant success factors for a software-driven technology to become mainstream.

The diversity and the distribution of use cases and participating entities, and the orchestration of entities that are operating in the background as embedded devices (Sec. 3.3.1) make Smart Spaces a more complex environment than smartphones.

Peripheral Diversity

The peripherals of smartphones are limited in their amount, and in their diversity. This allows the integration of fixed abstract interfaces to peripherals smartphone OSs (Sec. 3.10.1). As the peripherals of a smartphone are known in advance, suitable device drivers can be tested and added to a smartphone OS before shipping a new device. Smartphone OSs follow a monolithic approach (Sec. 3.4.4) having the drivers as part of the OS [But11].

The possibility to *dynamically adding support for new Smart Devices* is a major challenge in Smart Spaces [KFKC12, DHJT+10, Kru09] (<R.23>, <R.43>, <R.8>, <R.6>). It is not necessary in the App economy.

Peripheral Distribution

A second fundamental difference is that the peripherals and the execution platform of smartphones are **not distributed**. Handling distribution is a major challenge for a programming abstraction for Smart Spaces (<R.13>, <R.14>, <R.15>, <R.7>, <R.49>).

Invisibility of Smart Devices

A third difference is that the software in Smart Spaces will often run in the background out of the users direct attention. This is challenging in different aspects including software deployment, expected dependability, and self-management (<R.3>).

Distribution of the Execution Environments

A fourth challenge that is relevant in Smart Spaces but not on smartphones is the distribution of the execution environments. Smartphones Apps typically run on a single device, or in a managed cloud back-end (Sec. 4.3). Smart Spaces instead are likely to consist of distributed heterogeneous hosts. This requires more complex mechanisms for implementing a Smart Space service management (Sec. 7.2.4).

Portability

The analysis of the App economy introduced a third dimension for *portability*:

<R.45> **Portable service executables** are required to enable running the same binary on the heterogeneous hardware platforms of future Smart Spaces.

The *portability* of the OS itself that is required for Smart Spaces was already introduced with requirement

<R.7> **Portability of the implementation of the** programming abstraction itself must be supported.

The *portability* from specific peripherals was introduced before with the requirement:

<R.8> **Logical portability of services** must be supported by offering abstract interfaces to Smart Devices.

As described in Sec. 3.10.3, smartphone OSs provide fixed support for the peripherals described in Sec. 3.10.1. Other peripherals can only be supported by providing driver code in Apps which is not desired for Smart Spaces as discussed in Sec. 3.3.3.

Scalability

Smartphones are a fixed execution environment, Smart Spaces are a dynamically changing execution environment (Sec. 3.10.2.

Smartphones cannot provide scalability as their resources are fixed. Applications can only scale by offloading computation into the cloud (Sec. 4.3).

By adding or changing hardware components of Smart Spaces, providing scalability is enabled. To provide scalability, suitable software support is needed:

<R.50> A suitable programming abstraction for Smart Spaces must support **scalability**.

The described scalability is relevant for the implementation of a suitable programming abstraction for pervasive computing (Sec. 5.9), and for service implementations (Sec. 7.6.2).

With DS2OS, this thesis shows how the mechanisms of the App economy can be transferred to Smart Space Orchestration, and how resulting additional challenges of the Smart Space domain can be solved.

Subpart I–D

Requirements

3.11 Summary of the Analysis

Pervasive computing is not reality in 2014 (Sec. 2.4). A bottom-up (part I–A) and a top-down (part I–B) analysis of different domains were presented in this chapter. They identified 50 requirements that prevent the implementation of pervasive computing in real life. The high amount of requirements illustrates the difficulty of finding a suitable programming abstraction for pervasive computing (<O.0>).

The remainder of this chapter summarizes the requirements analysis results. This is done for making the relation between the requirements better understandable, and to recapture them in the context of this thesis' objective. To make them better manageable and understandable, Sec. 3.13 clusters the requirements into seven domains (see Fig. 3.14). To provide another structure, the requirements are also clustered to the thesis objectives they are related to (see Fig. 3.15).

The chapter started with an analysis of a future Smart Space topology. It revealed that the seven layers of context-aware Smart Space Software Orchestration that can typically be found in literature (Sec. 2.6.4) can be merged to four domains. This reduces the overall *complexity.* For the software layers (3,4,3a) in Fig. 3.1 this results into two domains, the "virtual world" that provides abstract interfaces, and services. Services adapt Smart Devices to the abstract interfaces, or implement orchestration workflows by using the abstract interfaces (Ch. 8).

Bottom-up (part I–A), Smart Device are the interfaces between the physical world and the virtual world (Fig. 3.1). They define what can be sensed and actuated via software. Sec. 3.2 analyzed Smart Devices in the professional (Sec. 3.2.1), and in the home domain (Sec. 3.2.2). It was shown that diverse Smart Devices are *available* in 2014. Interface heterogeneity was identified as a major challenge for software orchestrated Smart Spaces.

The DIY maker culture for Smart Devices was introduced (Sec. 3.2.3). It shows the possibility for users to create their own Smart Devices as basis for diverse pervasive computing scenarios. DIY hardware making *complements the goal of this thesis*, to enable user-based software development for software orchestrated Smart Spaces (<O.0>). If this thesis reaches its goal, users are enabled (and empowered, Sec. 2.5.2) to create all components of their future software orchestrated Smart Spaces on their own.

Projecting the development of Smart Devices into the future (Sec. 3.2.4), it was discussed that Smart Devices are likely to become a suitable *execution platform* for software in future Smart Spaces.

The *inherency of interface heterogeneity* was discussed (Sec. 3.3.1). It is a major challenge for providing *portability* of services in future Smart Spaces. The discussion showed that *using a single communication protocol for all Smart Devices is not desired.*

As existing methodology to overcome heterogeneity, classical *standardization* was analyzed. The diversity of domains and vendors makes all assessed approaches unsuitable for standardizing the abstract interfaces to Smart Devices. A *convergence* of abstract interfaces is necessary to provide service portability (Sec. 5.6.4).

The three possibilities to bridge protocol heterogeneity, *in the service*, "*in-the-middle*", and *on the Smart Devices* were discussed (Sec. 3.3). The result is that bridging "*in-the-middle*" has most advantages. To implement it, methods to bridge heterogeneity were analyzed. Diverging *expressiveness* of communication protocols was identified as major cause for undesired transparent *information loss.* The existing principle of multi-protocol translation was analyzed and considered unsuitable for future software orchestrated Smart Spaces.

The dual applicability of the middleware paradigm for providing *interface portability* towards services, and for providing *portability from heterogeneous execution environments* was introduced (Sec. 3.4). Different modes of inter-service communication were discussed. All are relevant for the implementation of diverse pervasive computing scenarios (<O.3>).

The comparison between monolithic kernels and μ-kernel operating systems showed the advantages of dynamic extensibility. Pervasive computing middleware should be extendable with services for purposes such as adaptation of Smart Devices and orchestration, and services that add functionality to the middleware core for supporting service developers. Such extension can be additional context search functionality for instance.

The **top-down** analysis (part I–B) started with the human mind (Sec. 3.5). It was shown that humans apply *abstraction principles* for generating *descriptive knowledge.* Problem solving can be described as identifying paths between so-called *frames* that represent states in the human mind (Sec. 3.5.4). A goal-oriented planning process seems more intuitive for humans than a procedural description of problem solving.

AI was analyzed as software implementation of intelligence (Sec. 3.6). FSMs are typically used to implement intelligent behavior. *ECA rules* were identified as simple-to-use, and often suitable method to represent Smart Space Orchestration tasks. *Modularization* was identified as important tool to structure complex tasks.

With *autonomic management*, a practical application of intelligent behavior for managing distributed entities was introduced (Sec. 3.7). *Autonomy* was identified as important principle for lowering *complexity* and increasing *dependability.*

Service orientation was introduced as a relevant method to modularize functionality (Sec. 3.8). The functionality of the WS implementation of the SOA principle was identified as not sufficient for Software Orchestration in future Smart Spaces.

Context modeling was introduced (Sec. 3.9). *Ontologies* were identified as most suitable concept for representing context in future Smart Spaces. The analysis of diverse meta models showed that *no suitable meta model for Smart Spaces exists* (table 3.3).

Finally, the App economy as successful example for the real world implementation of mobile computing was analyzed in part I–C. The analysis shows that Smart Spaces in 2014 are in a comparable situation to smartphones in 2007.

3.12 Requirements List

The analysis section (Ch. 3) identified 50 requirements for implementing Smart Space Orchestration in the real world. Supporting the requirements is required to reach the objectives of this thesis (<O.0>, <O.1>, <O.2>, <O.3>, <O.4>). The requirements were identified by analyzing different research fields.

To give an overview on the requirements they are briefly summarized in the order they were identified before. Requirements that belong together are indented. They are not summarized to one requirement to maintain the diverse aspects that are relevant for designing a suitable programming abstraction. Ch. 5ff shows that the indented requirements are solved with different methods.

A one page list of the requirements can be found in the appendix in Sec. A.1.

<R.1> *Simplicity-to-use* is an overall requirement. A simple-to-use programming abstraction is needed to shift the focus from implementing pervasive computing scenarios to implementing pervasive computing scenarios [Abo12]. The goal is to provide an abstraction that allows shifting the intellectual effort from the technical burden of implementation to focusing on the logic of pervasive computing scenarios.

<R.2> *Multi-user support* is needed to reflect the expected shared use of ambient computing resources.

<R.3> *Self-management* is needed to lower the complexity, to make ambient components usable for non experts, and to enhance the dependability of software orchestrated Smart Spaces.

<R.4> *User participation* is desired to empower users to implement the pervasive computing scenarios they have in mind. In addition, it is required to scale software development with the amount of services that are required in software orchestrated Smart Spaces (Ch. 8). *User participation* is supported by different other requirements, including <R.1>, <R.3>, <R.40>, <R.39>, <R.47>, <R.46>.

<R.5> *Context-awareness support* is required as processing context is a fundamental activity that all Smart Space services have to implement. Providing support for this functionality in the programming abstraction facilitates the development of services.

<R.6> *Role-independent unified interfaces* to all services (Ch. 8) facilitate the development of services. *Unified interfaces* are simpler to learn and use than heterogeneous interfaces. *Unified interfaces* enable modularization as all services use the same (compatible) interfaces.

Portability has different dimensions that are relevant for this work:

<R.7> *Portability of the implementation of the programming abstraction* is necessary to enable its use on heterogeneous computing nodes (Sec. 3.2.4).

<R.8> *Interface portability* describes the implementation of "standardized" interfaces for common functionality. This requirement includes *unified interfaces* (<R.6>), and *standardization* (<R.10>).

<R.9> *Information loss* happens when information is converted between representations with different expressiveness. To support diverse pervasive computing scenarios, and to enable dependable Smart Space Orchestration, it must be prevented or made explicit so that it can be considered by software.

<R.10> *Standardization* is needed to overcome heterogeneity of interfaces to services that connect Smart Devices, or provide other functionality. Requirement <R.6> focuses on the mechanisms that are used to access functionality of a service. This requirement focuses on the functionality that is offered over the interface, e.g. using one interface for the functionality "*lamp*" only (Sec. 5.6.4).

<R.11> With their spatial extent, Smart Spaces are inherently distributed which leads to the requirement of *distribution support*.

- <R.12> *Access transparency* is needed to hide how a resource is accessed (<R.6>) and how its data are represented (<R.10>).
- <R.13> *Location transparency* is needed to provide portability (<R.7>) in a distributed system via hiding the location of a resource.
- <R.14> *Migration transparency* is needed to hide when software services change their execution environments within software orchestrated Smart Spaces.
- <R.15> *Relocation transparency* is needed to make the access to services that are getting moved transparent.
- <R.16> *Concurrency transparency* is needed to provide shared access to services in a transparent way.
- <R.17> *Failure transparency* is needed to hide failure and recovery of resources.
- <R.18> *Persistence transparency* hides how information is obtained (e.g. disk, memory, database, service).

Communication Modes

- <R.19> *Direct communication support* is needed for time and security critical information exchange between entities in Smart Spaces.
- <R.20> *Mailbox communication support* is needed for security critical information exchange between entities in Smart Spaces.
- <R.21> *Publish-subscribe communication support* is needed for time but not security critical information exchange between entities in Smart Spaces.
- <R.22> *Generative communication support* is needed for time and security uncritical information exchange between entities in Smart Spaces.

<R.23> *Dynamic extensibility* must be supported at runtime by a programming abstraction for dynamically adding support for new Smart Devices, and for functionality that supports the implementation of pervasive computing scenarios.

<R.24> A *descriptive knowledge representation* should be used to represent functionality as it is more intuitive for humans than a procedural knowledge representation.

<R.25> *Interaction on different levels of abstraction* helps structuring orchestration workflows. Supporting it makes developing Smart Space service more intuitive for humans.

<R.26> *Modularization support* must be provided to structure problem solving in Smart Spaces. Modularization support reflects the natural problem solving process of humans: partitioning complex problems into manageable small ones.

<R.27> The *support of multiple programming languages with SDKs and IDEs* facilitates the software development process. It enhances the usability of Smart Space Orchestration in future Smart Spaces by supporting the tools that are best suitable for solving a certain problem with the programming abstractions.

<R.28> A *high expressiveness of a context representation* helps to minimize information loss when converting information from and into other representations (<R.9>). The context representation defines which information can be used by services. High expressiveness enables the implementation of diverse scenarios.

<R.29> A *simple-to-use context abstraction* facilitates the development of Smart Space services. For programming abstractions that offer context-awareness support (<R.5>) this is a sub requirement of <R.1>.

<R.30> Systems that implement Smart Space Orchestration must be *dependable* (Sec. 2.6.2).

<R.31> *Loose coupling of services* is needed to implement dynamic binding of services at run time. This allows to implement pervasive computing scenarios via modular components (<R.26>). Loose coupling of services can be implemented by fulfilling the following requirements.

<R.32> *Service contracts* are needed to implement encapsulation. They are the abstract interfaces of services (<R.33>, <R.34>).

<R.33> *Service autonomy* is needed to encapsulate functionality.

<R.34> *Service abstraction* is the second part of encapsulation. It is required to implement information hidings.

<R.35> *Service reusability* is needed for fostering modularization (<R.26>). It is implemented by encapsulation.

<R.36> *Service composability* is needed for fostering modularization (<R.26>). Providing unified interfaces results in composability (<R.6>).

<R.37> *Service statelessness* describes the separation of programming logic and data. It helps facilitating the development of services and it facilitates the migration of services (<R.14>, <R.15>).

<R.38> *Service discoverability* is needed to implement dynamic composition of services (<R.36>) at run time (<R.23>).

<R.39> *Abstract interface validation support* helps enhancing the dependability of a solution for Smart Space Orchestration (<R.30>, <R.49>).

<R.40> A *Meta model with suitable properties* as shown in table 3.3 is required to support scalable, and intuitive context-aware computing (<R.5>, <R.9>, <R.28>, <R.29>). Suitability includes:

<R.41> *Low complexity in the meta model* is required to facilitate the development of context models. It enables user participation in the context modeling process (<R.4>).

<R.42> *Collaboration support* in the creation of context models is needed for fostering user participation (<R.4>), and to provide the necessary scalability of the process to support the high amount of Smart Devices.

<R.43> *Dynamic extensibility of the context models at run time* is needed to add new context at run time. This is required to represent new functionality that becomes available in a Smart Space via adding Smart Devices or services.

<R.44> Context validation enhances the dependability of a software orchestrated Smart Space. It helps preventing errors (<R.49>). It supports user participation by providing immediate feedback (<R.4>).

<R.45> *Portable service executables* are needed to run executables on heterogeneous hosts. This requirement is introduced by enabling to run third parts services in Smart Spaces. It completes the *portability* requirements <R.7>, and <R.8>.

<R.47> The support of *emulators* facilitates the development of services and fosters user participation (<R.4>). It can be used to enhance the dependability of Smart Space software via extensive testing of services in an emulator. It allows emulating complex Smart Space setups.

<R.46> The availability of an *App store* is needed for managing the distribution of services to Smart Spaces. It is required for supporting user-based development via exchanging services between Smart Spaces (<R.4>). It is required for crowdsourcing (<R.48>).

<R.48> *Crowdsourced development* is needed to attract users in the development of services for software orchestrated Smart Spaces. It fosters a broad developer basis for developing the high amount of services that is required for implementing future Smart Space Orchestration.

<R.49> *Security-by-design* provides security in Smart Space software without developers having to implement it. This is needed as an important part of self-management (<R.3>) to implement a dependable system (<R.30>). *Security-by-design* is a requirement for user participation in the development of Smart Space services (<R.4>).

<R.50> *Scalability* is required for real-world use. The diversity of pervasive computing scenarios and Smart Devices require managing the big amount of context and services.

3.13 Mapping of Requirements to Objectives

This chapter is divided into a bottom-up and a top-down analysis. It is structured by analyzing different scientific domains. This section provides the mapping of the identified requirements to the thesis objectives (Sec. 1.1). Those of the 50 requirements that belong together (Sec. 3.12) are combined in a single circle for better understanding, and for saving space.

The high-level assessment of the state of the art in Sec. 3.13.2 and the detailed analysis in Sec. 4.5 confirm the necessity for a detailed requirements analysis that was done in this chapter. Existing surveys do not cover such a broad range of requirements (Sec. 4.5.1). A reason for a more narrow view is that none of the assessed research middleware is designed for real world use. The following assessment of the solutions and the more detailed analysis in Sec. 4.5 show that existing middleware covers only a subset of the identified requirements.

3.13.1 Legend

Fig. 3.15 clusters the identified requirements to the four objectives *Smart Device adaptation support* (<O.1>), *orchestration workflow implementation support* (<O.2>), *diversity support* (<O.3>), and *real world usability* (<O.4>).

Additional to the clustering, the requirements are associated with the three functional domains *service interface related*, *context interface related*, and *additional support functionality*. Fig. 3.14 shows the functional domains and the symbols that are used

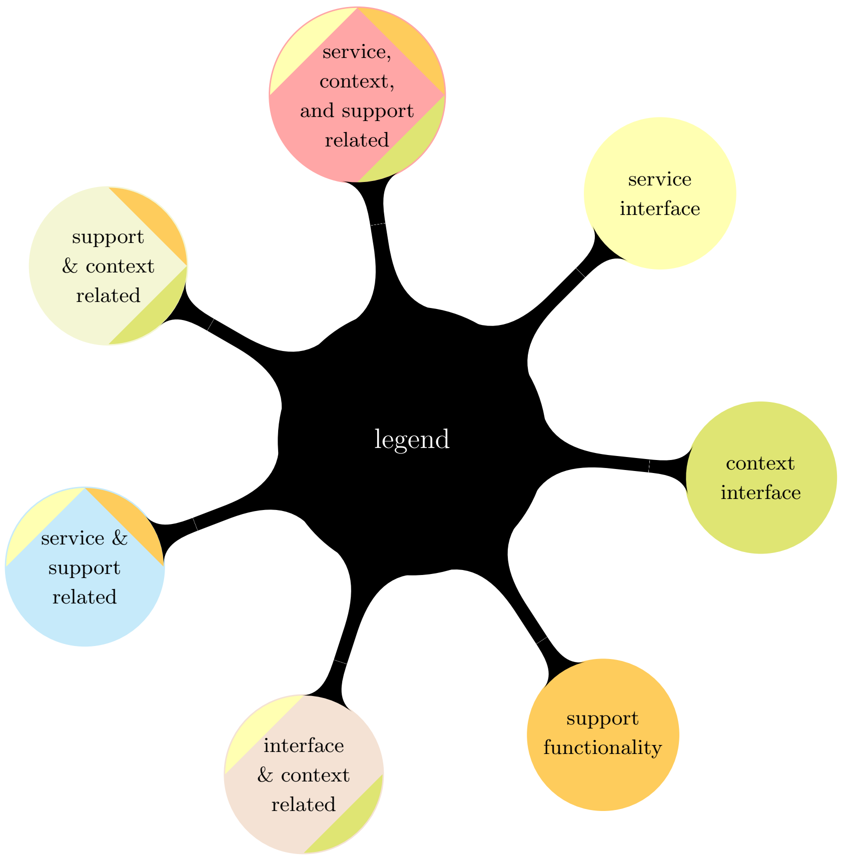

service, context, and support related
service interface
support & context related
legend
context interface
service & support related
support functionality
interface & context related

Figure 3.14: Legend to the requirements mindmap Fig. 3.15. The basic functional domains are on the right. The nodes on the left are combinations for requirements that belong to multiple domains.

to classify requirements that belong to multiple functional domains. It is the legend to Fig. 3.15.

Service interface related functionality is shown in yellow and has a symbol on the top left when combined with other functional domains. *Context interface related* has the color green and its symbol on the bottom right in case of combination with other functionality. *Additional support functionality* is shown in golden color. It has its symbol on the top right.

Fig. 3.15 shows that most of the requirements belong to more than one functional domain. That indicates that a mechanism that fulfills a requirement has impact on multiple functional domains. The design of the programming abstraction in Ch. 5 and its implementation in Ch. 6 show how the *service interface* and the context interface domain can be merged. Fig. 3.15 motivates this step. In each of the resulting requirement rings the *service interface* related requirements are shown next to the *context interface* related requirements. In between both groups are shared requirements of the two and between the two. As can be seen, many requirements are shared between the two functional domains.

The connecting lines in the flower of requirement <O.4> on the right from requirement <R.26>, *modularization support*, and requirement <R.5>, *context-awareness support*, show related requirements. Both are superior requirements that comprise the connected children.

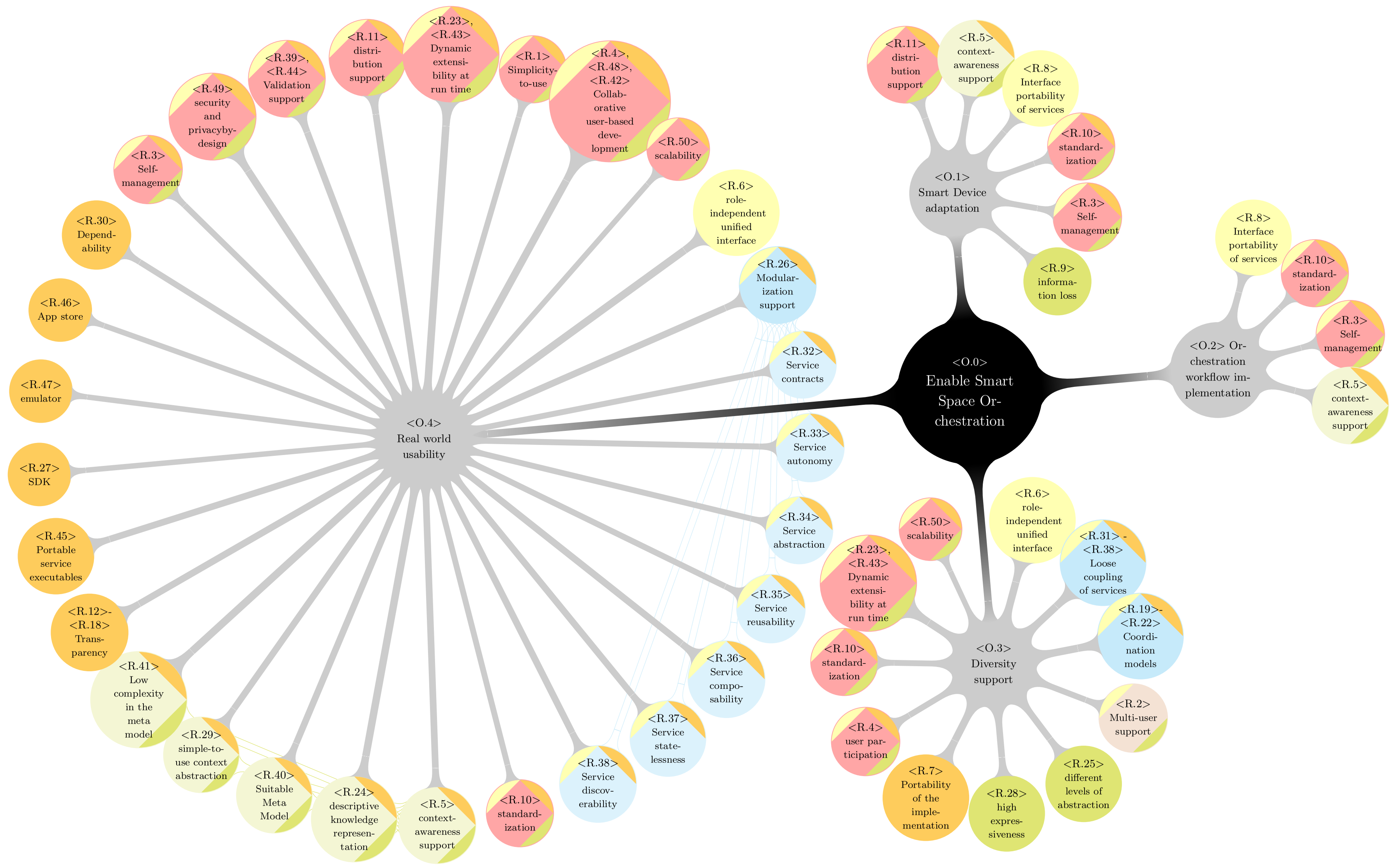

<O.0> Enable Smart Space Orchestration
<O.1> Smart Device adaptation
<R.11> distribution support
<R.5> context-awareness support
<R.8> Interface portability of services
<R.10> standardization
<R.3> Self-management
<R.9> information loss
<O.2> Orchestration workflow implementation
<R.8> Interface portability of services
<R.10> standardization
<R.3> Self-management
<R.5> context-awareness support
<O.3> Diversity support
<R.50> scalability
<R.6> role-independent unified interface
<R.31> - <R.38> Loose coupling of services
<R.19>- <R.22> Coordination models
<R.2> Multi-user support
<R.25> different levels of abstraction
<R.28> high expressiveness
<R.7> Portability of the implementation
<R.4> user participation
<R.10> standardization
<R.23>, <R.43> Dynamic extensibility at run time
<O.4> Real world usability
<R.49> security and privacyby-design
<R.39>, <R.44> Validation support
<R.11> distribution support
<R.23>, <R.43> Dynamic extensibility at run time
<R.1> Simplicity-to-use
<R.4>, <R.48>, <R.42> Collaborative user-based development
<R.50> scalability
<R.6> role-independent unified interface
<R.26> Modularization support
<R.32> Service contracts
<R.33> Service autonomy
<R.34> Service abstraction
<R.35> Service reusability
<R.36> Service composability
<R.37> Service statelessness
<R.38> Service discoverability
<R.10> standardization
<R.5> context-awareness support
<R.24> descriptive knowledge representation
<R.40> Suitable Meta Model
<R.29> simple-to-use context abstraction
<R.41> Low complexity in the meta model
<R.12>- <R.18> Transparency
<R.45> Portable service executables
<R.27> SDK
<R.47> emulator
<R.46> App store
<R.30> Dependability
<R.3> Self-management

Figure 3.15: Correlation between the identified requirements and the objectives of the thesis. The colored edges identify relationships to functional elements. A yellow color in the left top side identifies a relationship to the abstract interface of a service, a golden color on the top right identifies a relationship to platform support functionality, a green marker in the bottom right identifies a relationship to context management.

3.13.2 Assessment of the Fulfillment of the Requirements

The assessment of the state of the art in Sec. 4.5 shows that existing middleware for pervasive computing focuses on *Smart Device adaptation* (<O.1>), and *support for orchestration workflow implementation* (<O.2>).

The requirements that are associated with *diversity support* (<O.3>) are rarely supported. Most of the requirements that were identified in this chapter are associated with *real-world usability* (<O.4>). The discussion below and the state of the art analysis in Sec. 4.5 show that those requirements are typically not fulfilled by existing pervasive computing middleware.

The available support by pervasive computing middleware decreases clockwise with the increase of the number of the objective in Fig. 3.15. Most of the requirements that were identified in this chapter are associated with the objectives <O.3> and <O.4>. This could be an explanation, why pervasive computing is still missing a suitable programming abstraction for real world use (<O.0>, Sec. 2.5).

In the remainder of this section this high-level assessment is discussed more in detail. The discussion is structured by the thesis' objectives and their associated functional requirements in Fig. 3.15. Sec. 4.5 analyzes the state of the art structured per middleware.

Detailed Analysis

Fig. 3.15 shows the mapping of the requirements that were identified in this chapter (Sec. 3.12) to the thesis' objectives (Sec. 1.1). Following, all requirements from the figure are explained in the context of the objective they are assigned to.

Additionally the fulfillment of each requirement in the state of the art is described on a high level. This assessment is not done in form of a table as the requirements are too numerous. Instead a more detailed state of the art analysis with fewer criteria that are matched with existing surveys is presented in Sec. 4.5. Fig. 4.2 shows a mapping of the requirements to the functional domains that are used in the survey.

The services that implement Smart Device adaptation are called *adaptation services* in this work. Services that implement orchestration workflows are called *orchestration services*. Different service classes are introduced in Sec. 8.3.

Smart Device Adaptation

Smart device adaptation describes the ability of a middleware to connect Smart Devices.

Distribution support (<R.11>) is needed to make access to distributed Smart Devices transparent for orchestration services. Such support is provided by most related middleware designs (Sec. 4.5).

Context-awareness support (<R.5>) is required to implement pervasive computing services. Most of the related work supports context management. The degree of the support is varying. Most related work does not use explicit context models (Sec. 3.9).

Interface portability (<R.8>) expresses the ability to run a service in different Smart Space instances. *Interface portability* requires providing abstract interfaces to services that are identical in all Smart Space instances. In addition it requires a discovery mechanism for instances of the interface in a concrete Smart Space.

Abstract interfaces to functionality in Smart Spaces are only provided by few solutions. Discovery mechanisms are provided by a subset of them. None of the assessed middleware designs provides mechanisms to share abstract service interfaces. This would be necessary to allow distributed developers to program against an interface.

Standardization (<R.10>) is required to implement unified abstract interfaces to diverse (Smart Device) functionality in a Smart Space. The same functionality should be accessible via the same interface (e.g. "*lamp*") independent of the implementation (Sec. 3.3.2). None of the assessed related middleware offers standardization mechanisms.

Self-management (<R.3>) is required to manage the complexity that emerges from Smart Space Software Orchestration. All assessed middleware is self-managing.

Adaptation services convert between device-specific interfaces and abstract interface. This can lead to *information loss* (<R.9>). Only one of the assessed middleware designs (PACE, Sec. 4.5.11) considers such loss.

Orchestration Workflow Implementation

Orchestration workflow implementation support describes the support of a middleware to develop orchestration services.

Portability of services (<R.8>) is needed to enable the reuse and the sharing of orchestration services. Only one of the assessed middleware designs (One.World, Sec. 4.5.13) enables such reuse. The rest supports orchestration services only as context consumer. This is problematic as it prevents modularization.

Standardization (<R.10>) is needed for unifying the abstract interfaces to services (Sec. 3.3.2). The assessed middleware designs (Sec. 4.5) do not offer abstract interfaces to services and standardization methods.

Self-management of services (<R.3>) is provided by self-contained services. As services communicate typically not with each other they are self-contained.

Context-awareness support (<R.5>) is inherently needed to implement orchestration services (Sec. 2.6.4). Most assessed middleware (Sec. 4.5) support access to context repositories that are either stored in the implementation of a service or –in few cases– in the middleware.

Diversity Support

Diversity support describes the support of a middleware for diverse scenarios and Smart Devices.

Role-independent unified interface (<R.6>) simplify the implementation of services and foster modularization. Only one assessed related middleware implements such unification of the interface (One.World, Sec. 4.5.13).

Loose coupling of services (<R.31> - <R.38>) supports mash ups of complex functionality from simpler modules. In particular it facilitates the implementation of services that interact with diverse domains as it allows to use existing services from those domains. Service coupling and discovery are not supported by most existing solutions.

The support of different *communication modes* for inter-service communication (<R.19>-<R.22>) is required as different use cases require different coupling. Only six of the assessed middleware designs (Sec. 4.5) allow free communication between services. The rest only allows vertical communication with adaptation services. None of the assessed middleware supports all modes.

Multi-user support (<R.2>) will be required for diverse future pervasive computing scenarios (Sec. 2.5). None of the assessed middleware frameworks (Sec. 4.5) explicitly addresses multi-user support.

The possibility to interact with the environment on *different levels of abstraction* (<R.25>) facilitates the implementation of pervasive computing scenarios. Though some of the assessed middleware frameworks include hierarchical context processing, the concept of offering context on different levels of abstraction is not provided (Sec. 4.5).

High context expressiveness (<R.28>) is relevant for being able to meet the requirements of the diverse domains that are affected by pervasive computing. Expressiveness is not a topic that is addressed explicitly in the assessed related work (Sec. 4.5). Only one of the assessed middleware designs (PACE, Sec. 4.5.11) considers loss (<R.9>).

Implementation portability of a middleware (<R.7>) is important to allow using a programming abstraction and its implementation in diverse environments. Some of the related work is implemented in Java which leads to portability. None of the related work explicitly addresses portability to support diverse run time environments explicitly.

User participation (<R.4>) is desired to foster the implementation of user-driven pervasive computing scenarios (Sec. 2.5). In addition it is required to implement support for diverse Smart Devices and scenarios. Only one related work (OPEN, Sec. 4.5.12) provides support for user-based development, and this middleware does not support the implementation of user-based services with other mechanisms than logic rules. Another one discusses providing it (HomeOS, Sec. 4.5.5)

Standardization (<R.10>) is needed to enable *interface portability.* It is necessary for implementing diverse scenarios with diverse devices in a way that allows services to be used in multiple Smart Spaces with diverse hardware. Only OPEN (Sec. 4.5.12) and SOCAM (Sec. 4.5.9) support a convergence of context models. Both implement the approach of standardization by experts. As discussed in Sec. 3.3.2 this approach is not flexible enough for real world Smart Spaces.

Dynamic extensibility at run time is required for context (<R.43>), and for services (<R.23>) that are used in a Smart Space. Most of the reviewed middleware (Sec. 4.5) supports dynamic extension. Without sharing of abstract interface or context models the the possible extensions is of little use as services have to be coupled explicitly.

Scalability (<R.50>) becomes relevant in real world use of Smart Space Orchestration. Such use includes simultaneous interaction of different use cases that require support for different Smart Devices and that are connected via different services. The assessed middleware designs (Sec. 4.5) do not focus on scalability though most solutions scale to some extent.

Real World Usability

Real world usability concerns the deployment of a solution in the real world which implies diverse Smart Devices, diverse pervasive computing scenarios, diverse users, and diverse developers.

Role-independent unified interfaces to services (<R.6>) are fundamental for the development of services for Smart Spaces. They are the basis for modularization enabling service reuse and service mash up. Role-independent unified interfaces support programmers. Learning the use of a programming abstraction is facilitated by using identical concepts for creating adaptation services, orchestration services, and other services (Sec. 8.3). Only one assessed solution is offering such interface (One.World, Sec. 4.5.13).

Modularization helps structuring the complexity of programs (Sec. 3.8), and it is a concept humans are familiar with (Sec. 3.5). Therefore *modularization support* (<R.26>) with its facets (<R.32> - <R.38>) is needed for real world deployments. Several assessed middleware frameworks (Sec. 4.5) partly provide modularization support. None of the existing solutions makes extensive use of it by allowing developers to mash up complex services by reusing existing services that other users created.

Standardization of interfaces is necessary to enable *interface portability* concerning different devices with similar functionality. Only when the abstract interfaces of such devices are converged, services can access the functionality each of the devices provides, independent if it is from its vendor (Sec. 5.6.4). Only two related works offer standardized interfaces by using context models that are maintained by experts (SOCAM Sec. 4.5.9, OPEN Sec. 4.5.12). This approach does not scale for future Smart Spaces.

Context-awareness support is needed as context is the base for pervasive computing services. To be shareable, context models must be provided (<R.24>) using a suitable meta model (<R.40>). Both, the context models (<R.29>) and the meta model (<R.41>) must be simple-to-use. Simplicity includes that a context model can easily be understood, and that a simple interface to access context repositories is provided. Only two of the reviewed state of the art middleware designs (SOCAM Sec. 4.5.9, OPEN Sec. 4.5.12) use explicit context models. In both cases experts are maintaining the context models.

Transparency (<R.12> - <R.18>) is needed to hide the distribution and heterogeneity of software entities in Smart Spaces (Sec. 3.4). Most reviewed middleware provides transparency.

Portable service executables (<R.45>) enable portability of services between heterogeneous execution environments. Portability is required for real world deployment as hardware in Smart Spaces is likely heterogeneous (Sec. 3.2.4). Support for heterogeneous computing architectures and OSs is not explicitly considered by the assessed middleware implementations (Sec. 4.5). Most solutions are portable as they are implemented in Java, or in a scripting language. Both run interpreted code which leads to portability.

The availability of an *SDK* (<R.27>), and support for emulators (<R.47>) are relevant for facilitating the development of Smart Space services. Several of the assessed solutions provide programming container for implementing services. They can be considered as part of an SDK. Detailed documentation and emulator environments miss for all assessed solutions.

An *App store* (<R.46>) as repository for software, and as interface between developers and users is required to share and distribute services as it happens in the App economy in 2014 (Sec. 3.10). Only one assessed solution (OPEN, Sec. 4.5.12) provides a working solution. However, OPEN only allows sharing rules since its services are logic rules. This facilitates the work of an App store, and it makes the solution unsuitable for general real worlduse. The HomeOS developers state the requirement for an App store but do not provide a solution.

Dependability (<R.30>) of software orchestrated Smart Spaces is required for several reasons including safety (Sec. 2.6.2), and a user shift compared to current prototypes. The emergence of Smart Spaces will lead to a shift of the users from technically skilled makers [BLM+11] towards regular customers potentially without deep technical understanding, but with high expectations on the reliability of a system at the same time. Dependability is not explicitly handled by the assessed middleware systems.

Self-management is needed to provide dependability (sec. 3.7). It makes the complexity of diverse hardware and software systems in future Smart Spaces manageable. All assessed solutions (Sec. 4.5) provide self-management of the middleware platform. Few of the assessed middleware designs manages services locally. Only OPEN manages the deployment of its ECA-rule-based services into Smart Spaces. None of the solutions addresses self-management of context models.

Security-by-design is required for user-based software development in Smart Spaces. "Security-by-design" expresses that security is automatically handled without developers having to take explicit care. None of the assessed solutions (Sec. 4.5) provides security-by-design. Some of the solutions include authentication and access control.

Validation support for abstract interfaces of services (<R.39>), context models, and ideally for service implementations is required for providing dependable Smart Space Orchestration. Failing services and broken context models can compromise the dependability of the entire software infrastructure of future Smart Spaces. The literature about the assessed middleware designs does not explicitly discuss validation. The solutions that use context models typically use markup based context models. They automatically allow basic validation (Sec. 3.9). Services and their abstract interfaces are not validated in the assessed solutions.

Distribution support (<R.11>) of the middleware platform and of services is required for high scalability, and dependability. Distribution support for the context management is required to enable collaborative development of context models, and to implement standardization. Half of the assessed solutions (Sec. 4.5) are designed as distributed systems, or allow the execution of distributed services. Only Aura provides mechanisms to monitor and control services at run time. None of the solutions provides support for collaborative authoring of context models.

Dynamic extensibility of a middleware with services at run time (<R.23>) is required for real world deployment. It can be expected that the operating systems for future software orchestrated Smart Spaces are permanently running. Only with dynamic extensibility, support for new Smart Devices and pervasive computing scenarios can be added. Most assessed middleware platforms can be extended with services.

Dynamic extensibility of context models (<R.43>) is required to reflect dynamic context changes in a Smart Space (e.g. adding new entities). None of the assessed solutions uses a dynamically extensible ontology. OPEN (Sec. 4.5.12) and SOCAM (Sec. 4.5.9) use extensible ontologies that is maintained by experts. From the literature it is not clear if this ontology can be exchanged in Smart Space instances at run time.

Simplicity-to-use (<R.1>) is required for real world deployments. All platforms are self-managing which facilitates the operation of the platforms.
A simple API facilitates software development, enabling user-based development. A fixed APIs is a simple interface. Those middleware designs that support context management use fixed APIs. But they do typically not support service coupling.
The context models of the assessed solutions are managed manually. This solution is not simple enough for real world deployment.

Collaborative user-based development (<R.4>, <R.48>, <R.42>) is needed for different reasons as described before. It helps implementing diverse services for adapting Smart Devices, and diverse pervasive computing scenarios. This enables users to implement their ideas (Sec. 2.5). It makes the required software development scale with the high demand on software service for software-orchestrated Smart Spaces. Only two of the assessed solutions (OPEN 4.5.12, HomeOS, Sec. 4.5.5) foster user-based development. In OPEN only ECA-rule-based services are supported. In HomeOS only the idea is communicated.

Finally, *scalability* (<R.50>) of all components is required for a real world deployment. Within their limited functional support, most of the assessed solutions scale (Sec. 4.5). Self-organizing distributed solutions with context models that can efficiently be parsed scale best.

3.14 Chapter Conclusion

This section (Ch. 3) identified 50 requirements for real world deployment of pervasive computing for Smart Space Orchestration. Sec. 3.11 summarized the findings. Sec. 3.12 mapped the identified requirements to the objectives of this thesis. Sec. 3.13.2 discussed the fulfillment of the objectives by the state of the art on an abstract level. The following chapter contains a more detailed state of the art analysis that confirms the described viewpoint.

Subpart I–E

State of the Art

4. State of the Art

It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years.

John von Neumann, Hungarian born American Mathematician, (1903 - 1959).

Short Summary The chapter starts with an analysis of distributed user-based software development, virtualization as basic paradigm of cloud computing, network management, and building management.
The main part of this chapter analyzes relevant state of the art pervasive computing middleware. The selection is based on different surveys. The assessment criteria are also based on several surveys and extended with the requirements from Ch. 3.

Key Results Crowdsourced development of software for Smart Spaces is possible in 2014.
The encapsulation of software in containers facilitates their management.
Using fixed interfaces to access information simplifies the access and results in interface compatibility.
Solutions for network management and building automation are unsuitable for direct use as basis for software orchestrated Smart Spaces. Neither the data models nor the interaction methods they offer are sufficient to support the implementation of diverse pervasive computing scenarios.
The requirements that were identified in Sec. 3.12 are considered relevant in the research community. Different existing middleware designs solve different subsets of the requirements. Pervasive computing middleware typically provides functionality for *service management*, or *context management*.
Most middleware provides partial solutions for the adaptation of Smart Devices (<O.1>), and for the creation of orchestration services (<O.2>). Diversity support (<O.3>), and real world deployment support (<O.4>) are typically missing.

A fundamental problem of existing pervasive computing middleware is that it creates middleware silos that result in missing interoperability between services that implement pervasive computing scenarios in different domains. Additional missing aspects are a convergence mechanism for service interfaces and context models, missing support for user-based development, missing scalability, missing security, and missing mechanisms for global service deployment and management.
A comprehensive middleware that combines the features of the existing middleware is missing in 2014.

Key Contributions A survey of state of the art pervasive computing middleware is presented. It exceeds the surveys from literature [HIM05a, EBM05, HR06, BDR07, Kjæ07, CD07, HSK09, GGS09, Kru09, SHB10, BBH+10, Pos11, CD12, BCFF12, RCKZ12, KKR+13] in the amount of assessment criteria, and the combination of solutions that are assessed.

Summary The chapter starts with an analysis of distributed user-based software development (Sec. 4.2). It shows that such development is likely to be possible for Smart Spaces in 2014.
Then cloud computing mechanisms that support service portability on heterogeneous hardware are presented (Sec. 4.3). It is shown that encapsulation of services provides flexibility in the use of resources, and security as it limits the interaction between software to a defined interface. Virtualization is presented as general concept to implement portability (Sec. 5.6.3).
Network management systems are analyzed in terms of their suitability for Smart Space Orchestration (Sec. 4.4). At the example of the Simple Network Management Protocol (SNMP) it is shown how a descriptive representation of management information can be implemented. The analysis concludes that the methods of network management are partially suitable for Smart Spaces. The strictly hierarchic design and the used data models are too complex for crowdsourced development.
The analysis of the most widely used solution for building management, Building Automation Control network (BACnet), shows how interoperability can be enforced by standardizing limited functionality as interoperable interface.
The main part of this chapter is the analysis of relevant state of the art pervasive computing middleware (Sec. 4.5). It is shown that most existing pervasive computing middleware provides partial solutions for the adaptation of Smart Devices (<O.1>), and for the creation of orchestration services (<O.2>). Diversity support (<O.3>), and real world deployment support (<O.4>) are typically missing.
Based on existing surveys, 13 relevant middleware designs are selected. They are assessed according to 16 criteria from the domains *system design*, *context management*, *service management*, and *real world deployment support* (Sec. 4.5.2). A mapping of the assessment criteria to the requirements from Sec. 3.12 is presented (Fig. 4.2). The assessment results are summarized in table 4.1.
A combined reference model is presented (Sec. 4.5.16). It provides a combined overview of functionality that is provided by all existing state of the art pervasive computing middleware together. It is used as basis for comparison of the novel

Distributed Smart Space Orchestration System (DS2OS) design with the existing designs in Sec. 7.7.2.

4.1 Introduction

Different research areas are relevant for a real world implementation of pervasive computing as discussed in Ch. 3. This thesis focuses on the software side of the problem space. It targets the design of a reference architecture for enabling Smart Space Orchestration (Sec. 2.4.1).

Solutions for parts of the requirements that are summarized in Ch. 3.12 exist in different areas of computing. The chapter starts with a brief presentation of the 2014 state of the art in *distributed user-based software development*, *portable resource-usage*, and the *management of embedded systems*. The main contribution of this chapter is an analysis of the most relevant state of the art pervasive computing middleware.

4.2 Distributed User-Based Software Development

Distributed user-based software development, and sharing of software exist from the beginning of computing in the 1960s [LT03, DWJG02]. *GNU/Linux* is probably the best known example for a software that is released as so-called Open Source Software (OSS). In 2014 there are diverse websites that support the development and sharing of OSS. Examples such as GitHub[12] or Sourceforge[13] host millions of users and software projects.

Definition ***Crowdsourced Software Development***

> The term *crowdsourced (software) development* is used to describe distributed user-based software development in this document. In contrast to user-based development, crowdsourced development typically comprises competitive elements (Sec. 3.10).

The success of the App economy is based on crowdsourced development (Sec. 3.10). In 2012 more than 1 000 000 users developed Apps for smartphones [PMC+12]. At the beginning of 2013, about 800 000 Apps were available in the Apple App store. 20 billion Apps were downloaded in 2012[14]. This success shows the importance of software distribution mechanisms that are simple-to-use by non-experts (<R.46>, Ch. 7).

For maintaining the described software, so-called *software packages* consisting of a software and additional metadata are typically stored in central well-known repositories. The distribution of software happens via (semi-) automated installation and update mechanisms that are usable by non experts [MBdC+06, App13, Goo].

Conclusion

The brief discussion shows that crowdsourced development is possible in 2014. Skilled developers, and infrastructure components for enabling crowdsourced development development exist (e.g. central software repositories). Different studies acknowledge the existence of skilled and enthusiastic Do-It-Yourself (DIY) developers in the pervasive

[12]https://github.com/

[13]http://sourceforge.net/

[14]http://www.apple.com/pr/library/2013/01/07App-Store-Tops-40-Billion-Downloads-with-Almost-Half-in-2012.html

computing domain [BLM+11, MH12]. As expressed in <R.46>, besides a suitable programming abstraction for Smart Spaces (<O.0>), a suitable development and distribution mechanism for services for Smart Spaces is missing in 2014.

Sec. 2.4 and Sec. 2.5 discuss the relevance of crowdsourced development for pervasive computing [DM12, TWDT13, Fis98, Abo12, NMMA13]. Sec. 7 introduces Smart Space service development support and distribution mechanisms based on the Virtual State Layer (VSL) programming abstraction that is introduced in this thesis (Ch. 5).

4.3 Portable Resource-Usage

Large scale distributed computing is implemented as so-called cloud computing in 2014. Cloud computing enables the flexible use of distributed resources for executing computation tasks. The used resources are typically in the Internet, e.g. Amazon EC2/S3[15], Microsoft Azure[16], HP Cloud Services[17], Rackspace OpenCloud[18] [ASZ+10, LTM+11, VVE09, RCL09]

Cloud computing typically uses *virtualization* for providing services [ASZ+10]. In the context of cloud computing, virtualization describes the emulation of abstract system interfaces. Emulated interfaces (e.g. of a certain hardware) are offered to software components that run on top of them [TVS06].

A concrete example is providing a container, called Virtual Machine (VM), that emulates the components of an entire Personal Computer (PC). Having VMs that offers emulated PC interfaces to software allows to run regular operating systems in them (*paravirtualization*). This can be used for running multiple operating systems in controlled environments on one physical host for instance.

Cloud computing encapsulates software behind controlled interfaces. Such caging within well-defined interfaces can help enhancing the security of systems executing the VMs as it provides clearly defined and controllable interfaces to software. The freedom of the software is effectively limited by the externally enforced interfaces.

By introducing an emulated abstract interface between hardware and software, paravirtualization provides portability from physical hardware. The emulated software interfaces become the coupling point between the software and hardware and not the underlying hardware specific interfaces of a system. The encapsulation of software in containers allows migrating it between different physical machines [VVE09, LTM+11]. Having encapsulated services helps optimizing the resource usage and helps providing scalability in cloud computing.

Conclusion

Virtualization allows running the same application container (VM) on heterogeneous hardware. This is attractive for running software components in future Smart Spaces on heterogeneous hardware (Sec. 3.2.4). In this thesis, Java is used to provide the described portability from the underlying hardware platform (Sec. 6.3.1).

[15]http://aws.amazon.com/ec2/
[16]http://windowsazure.com/
[17]http://hpcloud.com
[18]http://www.rackspace.com/open-cloud/

In the context of cloud computing, virtualization is used to migrate software between computing hosts [VVE09, LTM$^+$11]. This is attractive for optimizing the resource usage in Smart Spaces. Sec. 7.2.4 presents basic mechanisms for migrating Smart Space services. As future research, migration techniques that exist for cloud computing already could be applied to the system presented in Sec. 7.

Virtualization is also relevant for future Smart Spaces as a general concept. The encapsulation of services behind defined interfaces is interesting to enhance the security and dependability of future Smart Spaces (Sec. 5.5.6).

Smart Spaces contain distributed heterogeneous Smart Devices (Sec. 3.2). A virtualization of Smart Devices would provide portability from the heterogeneity of the device interfaces. This thesis implements such a solution for virtualization of Smart Device interfaces. It allows to provide standardized virtual interfaces to different Smart Devices.

An example for heterogeneous Smart Devices are lamps of different vendors with heterogeneous interfaces. Via the presented solution (Ch. 5), they get the same virtual interface (Sec. 5.6.4). It can be used by software services to access the common functionality. This provides portability, uncoupling Smart Space services from concrete Smart Device interfaces (Sec. 5.6.3).

4.4 Management of Distributed Embedded Systems

As described in Sec. 3.7, complex computing systems require management. Software-based management solutions exist for different computing systems in 2014 including network elements and Building Automation Systems (BASs).

Network elements are systems that keep computer networks running such as switches, or routers. Network management is a known task since the time before the Internet became a commercial success ([Cer88], 1988). Looking at solutions developed for this domain is relevant for Smart Space Orchestration as computer networks are a major part of Smart Spaces, and as managing network elements shares aspects with managing Smart Devices. Network management solutions are built by experts for their own problems, giving an insight what software experts consider good practice.

Looking at management for BAS is relevant as BASs contain important components for software orchestrated Smart Spaces (Sec. 3.2).

4.4.1 Network Management

Network management manages network elements. Network management tasks include monitoring and controlling network elements with the goal of implementing effective and efficient operation of a managed system and its entities according to certain goals [hAN99].

For efficient implementation of network management, *open* interfaces are desired. An interface is called *open* if it provides interoperability with other systems. This can be implemented by using standard technology, and publishing interface specifications [TVS06, hAN99]. As few vendors dominate the market for network elements, a defacto standardization (Sec. 3.3.2) of such interfaces exists in 2014.

The typical organizational model of a Network Management System (NMS) is shown in Fig. 4.1 [KS94]. A *managed resource* such as a network element is shown on the left. It is typically a hardware device. The management interface is implemented by a software agent (*agent*). An *agent* either runs on the managed component or on an external device that is connected to the managed resource. An agent typically provides management information about the managed resource via a *Managed Information Base (MIB)*. The MIB provides information about the managed resource in a structured way. The managed resource together with the agent is called *Managed Object (MO)*. In the context of Smart Spaces it would be called a Smart Device in this work (Sec. 2.4.1).

The high level network management logic is usually situated on a manager that runs on a device that is external to the managed resource. It can have a user interface to enable users to communicate management goals or to monitor the system. Fig. 4.1 shows that a hierarchical management is implemented from the left to the right as only neighboring components are communicating.

In the presentation of the ubiquitous computing middleware state of the art in Sec. 4.5 such a strictly hierarchical management approach is called *vertical design*. A *vertical design* includes that components on one level of abstraction do not communicate with each other but only with higher or lower level components. The opposite approach where components within a layer (e.g. managers) communicate with each other is called *horizontal design*.

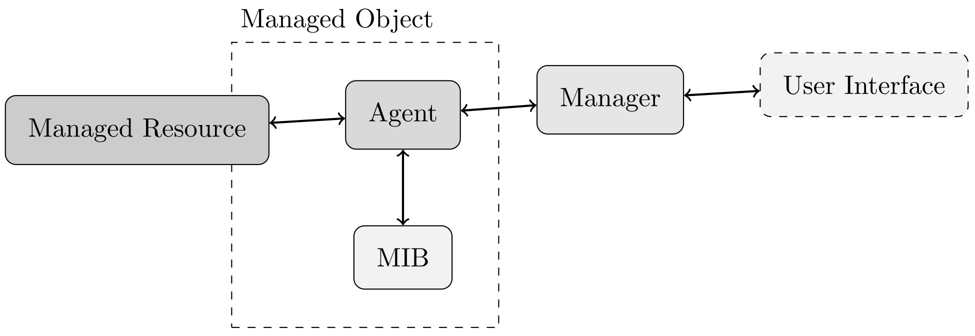


Figure 4.1: Abstract Organizational Model for Network Management Architectures.

Typical Network Management Systems

Many Network Management System (NMS) share common properties that are described next [KS94, hAN99, Sch08]. The Simple Network Management Protocol (SNMP) is used as a widespread and representative example in this paragraph (RFC3411-3418) [HPW02].

NMSs typically use a descriptive hierarchical structure to represent the manageable properties of Managed Objects (MOs). Sometimes the data model allows inheritance (e.g. Open Systems Interconnection (OSI) management). SNMP does not allow inheritance. Listing 4.1 shows the representation of a SNMP MIB in Abstract Syntax Notation One (ASN.1).

```
BGP4-MIB DEFINITIONS ::= BEGIN

                IMPORTS
```

```
                    MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE,
                    IpAddress, Integer32, Counter32, Gauge32
                        FROM SNMPv2-SMI
                    mib-2
                        FROM RFC1213-MIB;

                bgp MODULE-IDENTITY
                    LAST-UPDATED "9405050000Z"
                    ORGANIZATION "IETF BGP Working Group"
                    CONTACT-INFO
...
                bgpLocalAs OBJECT-TYPE
                    SYNTAX       INTEGER (0..65535)
                    MAX-ACCESS   read-only
                    STATUS       current
                    DESCRIPTION  "The local autonomous system number
                        ."
                    ::= { bgp 2 }
...
                bgpBackwardTransition NOTIFICATION-TYPE
                    OBJECTS { bgpPeerLastError,
                              bgpPeerState      }
                    STATUS       current
                    DESCRIPTION  "The BGPBackwardTransition Event is
                         generated
                            when the BGP FSM moves from a higher
                                numbered
                            state to a lower numbered state."
                    ::= { bgpTraps 2 }

END
```

Listing 4.1: Representation of a Border Gateway Protocol (BGP)4 MIB.

The typical organization of a NMS is *vertical* as described above. It consists of few managers that manage all distributed MOs. SNMP follows this scheme.

The major operation of a NMS is the retrieval or manipulation of states from the MOs. The methods that are used to access the information on the MOs are typically fixed. This is reflected by the MIB component in Fig. 4.1. Since a MIB has a structure similar to a database with keys and values (listing 4.1), a management interface with fixed functions can be used to access MIBs.

SNMP uses the operations `get` to query the value of a specified variable, `getNext` to traverse a MIB, `set` to change the value of a specified variable, and `trap` for asynchronous reports from an agent to a manager.

For providing portability, the structure of the information in the MIBs is defined via standardization [RM90]. Listing 4.1 shows such a definition for a specific router. NMS typically use the regular standardization process discussed in Sec. 3.3.2. This is possible as the relevant properties of network elements are rather static, and as few vendors are producing the specialized equipment.

Conclusion

Using a fixed interface with few methods to access information is attractive for accessing Smart Devices as it is simple-to-use (<R.1>). The solution proposed in this thesis also uses a fixed interface but it gives access to more dynamic information (Ch. 5).

Using a fixed interface, the structure of the accessed information becomes important. The structure must be standardized to enable portability. The implementation of a standardization process for the structure of the MIBs is not applicable to Smart Devices. Since many vendors are producing Smart Devices the described standardization process is unlikely to succeed within acceptable time (Sec. 3.3.2). The functionality of Smart Devices is too diverse which would require a large amount of standardization processes running permanently (Sec. 3.3.2). This seems infeasible as it requires time, personnel, and money.

Data models [PS03] are used to describe the structure of the MIBs. The data models that are used by major NMS solutions on the market in 2014 are suitable for experts. This includes SNMP (listing 4.1), IBM Tivoli [TBEJ08], Distributed Management Task Force (DMTF) Desktop Management Interface (DMI)/ Management Information Format (MIF) [Dis], and Web-Based Enterprise Management (WBEM) Common Information Model (CIM)/ Managed Object Format (MOF) [Dis]. For non-experts it is difficult to use them.

As described in Sec. 5.2, this thesis proposes a user-based creation of the data models of Smart Spaces (called context models). For the discussed reasons the data models of NMSs are not suitable for Smart Space Orchestration.

The strictly *vertical design* of typical NMSs is not flexible enough for the implementation of diverse pervasive computing scenarios (<O.3>). It does not provide sufficient support for the development of complex orchestration services to implement complex pervasive computing scenarios (<O.4>).

4.4.2 Building Management

As described in Sec. 3.2.1 most abstract management layer of a BASs is the management level. A relevant enabler for building automation management is the Building Automation Control network (BACnet) protocol [ASH95, TK10, MHH09, KFKC12].

BACnet is a standardized communication protocol to exchange information between entities in BAS. To conform with the BACnet standard, an entity (e.g. a Smart Device) must offer so-called *objects* with defined *properties*. A BACnet *object property* could be a temperature. *Properties* offer so-called *services*. To be readable, a property must offer a *ReadProperty* service. A call of a *ReadProperty* service returns the value of the *object property*.

To use BACnet, an entity sends a BACnet message and waits for a response. BACnet implements a *horizontal* management structure. The protocol allows to discover objects via network broadcast. Objects of a searched type respond and can be used.

The data model of BACnet is fixed. BACnet has 25 standardized *object* types including binary input and output, number, scheduling, trending, or alarming. Vendors can get a vendor ID to define their own proprietary additional *objects*. BACnet does not provide standardization mechanisms for such additional *objects* [Sno03, KNSN05].

Building automation management with BACnet can be implemented by writing applications that communicate using the BACnet protocol. Its standardization allows communication with all compliant devices. BACnet facilitates the creation of such services by providing portability from the heterogeneous Smart Device interfaces.

Conclusion

BACnet is designed as system of peers that communicate. A distributed system where the participating communication partners are equally functioning as client and server is called Peer-to-Peer (P2P) system [TVS06]. The P2P design suits not only for building automation but also for the implementation of general pervasive computing scenarios. The VSL middleware, which is introduced in this thesis (Sec. 6), is implemented as P2P system (Sec. 6.3).

BACnet provides a basic communication abstraction that overcomes the heterogeneity of device protocols (Sec. 3.2). This is fundamental for Smart Space Orchestration as well since it enables portability of services (Sec. 5.6.3).

Using BACnet as enabler for pervasive computing scenarios in future Smart Spaces is problematic. A first problem is that no concept or support for standardizing additional device *objects* is available. The diversity of Smart Devices that is relevant for pervasive computing (Sec. 2.4) requires a significant amount of *objects* to be defined and standardized to maintain the interoperability (Sec. 5.6.4).

A second problem of using BACnet as basis for Smart Space orchestration is that the system provides limited support for service developers. Processing protocol messages makes it complex and difficult for non experts to develop BACnet services. Services are intended to implement pervasive computing scenarios in software orchestrated Smart Spaces. Though limiting the complexity that is introduced by the protocol heterogeneity of the devices, the remaining complexity is too high for user-based software development that is targeted by this work (Sec. 4.2, <O.4>).

The most fundamental problem that makes the BACnet approach unsuitable as basis for Smart Space Orchestration is the functional limitation of the interface. A *property* of a certain type always has a fixed *data type*. Standardized *objects* have a fixed format. To enable portability, information can only be represented in a predefined (standardized) way. This limitation is not a problem for BACnet. Providing a fixed interface for a defined set of functionality offered by Smart Devices in a building is the key to interoperability in BACnet [BN91, ASH95, TK10, MHH09, KFKC12].

A drawback of this approach is the limited support for custom functionality. For the diversity of pervasive computing scenarios and Smart Devices limitation is a problem (<O.3>). The limitation to one protocol for an application domain (one BACnet object type for a certain functionality) can be expected to lead to information loss as discussed in Sec. 3.3.3.

Using a single protocol for all devices is too limiting for the diversity of pervasive computing scenarios and Smart Devices. Therefore this thesis introduces a dynamically extensible type system that supports multi-inheritance (Sec. 5.2).

4.5 Pervasive Computing Middleware

An indicator for the seniority of pervasive computing research [Abo12, DM12] is the availability of surveys that review different pervasive computing middleware approaches that emerged over the years. At the same time, the availability of such surveys shows that the search for a suitable programming abstraction (<O.0>) for pervasive computing is still ongoing [Abo12].

Concerning the requirements in Fig. 3.15, typical pervasive computing middleware solves Smart Device adaptation for (some) devices (<O.1>), and provides support for orchestration workflow implementations in the supported domain (<O.2>). It provides domain- or workflow-specific support functionality (see 4 in Fig. 4.3).

A problem with such specific functionality is that it typically does not provide sufficient support for diverse pervasive computing scenarios (<O.3>):

- If support for diverse Smart Devices cannot be added to a middleware, the possibilities of Smart Space Orchestration are limited (Sec. 3.2).
- If domain- or use-case-specific functionality is supported by a middleware this often limits the way developers program their services [EBDN02, KKR$^+$13].

An example for domain- or use-case-specific functionality in a middleware is an interface to parameterize given rules that do certain actions (e.g. Sec. 4.5.11).
Developers are likely to use such rules as they offer a comfortable interface to implement functionality. If adequate mechanisms for implementing functionality are only provided for one way of programming, it is likely that developers follow this predefined way. As discussed in Sec. 2.5 this opposes the goal to enable diverse implementations of pervasive computing scenarios.
If a middleware is built around domain- or use-case-specific functionality it inhibits the implementation of pervasive computing scenarios from other domains in the worst case [EBDN02, KKR$^+$13].

The term *μ-middleware* is introduced in this thesis in Sec. 6.2 to describe if a middleware contains domain- or workflow-specific functionality or not. A middleware is called *μ-middleware* if it does not contain workflow-specific functionality, but it provides dynamic extensibility with workflow-specific functionality.

Having domain- or workflow-specific middleware does not provide sufficient support for diversity (<O.3>). Instead it lifts the heterogeneity on a higher level of abstraction. With typical state of the art pervasive computing middleware, the middleware becomes a silo on a higher level of abstraction.

Typically, a middleware that was designed for the health domain cannot interact with a middleware for the entertainment domain. It overcomes the heterogeneity of the Smart Devices but it becomes a silo itself by supporting a certain domain.

Middleware silos complicate the shared use of Smart Devices for different purposes since devices are typically connected to one middleware only. They inhibit the implementation of domain comprehensive pervasive computing scenarios (<O.3>).

For real world usability diverse properties have to be fulfilled including usability (<R.1>), scalability (<R.50>), dependability (<R.30>), and security (<R.49>).

Scalability is often not given because of central components that may become a bottleneck, implementations of the context repository that do not scale with large amounts of context, and meta models that are resource intensive to parse.

Real world dependability is not an issue as the solutions are only used within controlled environments. Security and privacy are typically not taken into account.

User-based development (<R.4>) is typically not supported by research middleware. The meta models (Sec. 3.9) and their Application Programming Interfaces (APIs) are often complicated. The resulting complexity of services can be expected to lower their dependability and security.

Real world usability (<O.4>) is typically out of scope for research project middleware designs.

The section is structured into two parts. The first part surveys relevant middleware from the past years (Sec. 4.5.1). The second part introduces a reference architecture that combines all major features of the surveyed middleware designs as common classifying denominator (Sec. 4.5.16).

The related work is presented mainly for three reasons,

- to show the relevance of the problems handled in this thesis for the research community,
- to identify shortcomings of the existing work in relation to the identified requirements (Sec. 3.12), and
- to compare the existing work with the VSL programming abstraction that is introduced in this thesis (Ch. 5) and the Distributed Smart Space Orchestration System (DS2OS) framework that extends the VSL with service management functionality (Ch. 7).

4.5.1 Assessment Criteria

The selection of the middleware that is assessed in this section is based on the surveys [HIM05a, EBM05, HR06, BDR07, Kjæ07, CD07, HSK09, GGS09, Kru09, SHB10, BBH$^+$10, Pos11, CD12, BCFF12, RCKZ12, KKR$^+$13]. The surveys do not cover all requirements that are identified in Sec. 3.12. Therefore this section combines assessment criteria from the different surveys and adds some additional parameters from the requirements. The amount of assessed pervasive computing middleware solutions and applied assessment criteria exceeds that of the referenced surveys.

The assessment criteria are explained below. Fig. 4.2 shows a mapping of the 50 requirements that are identified in Sec. 3.12 to the assessment groups used in this survey.

The added criteria are *Fixed Service API*, *μ-middleware*, and *Support for user-based development*. The listed criteria are mainly relevant for *diversity support* (<O.3>) and *real world deployment* (<O.4>). That they are not in the focus of existing surveys shows that those objectives are not well covered by pervasive computing middleware so far as described before (Sec. 4.5). In addition it shows that the relevance of the criteria seems new for the research community.

The state of the art middleware can be clustered into two groups:

- *Service management* middleware focuses on services and their composition.
- *Context management* middleware focuses on context provisioning.

The presentation of the state of the art in Sec. 4.5.2 starts with *service-centered* approaches and ends with *context-centered* approaches. Table 4.1 summarized the assessment results.

The last entry of table 4.1 is the DS2OS middleware framework. This thesis introduces a novel kind of pervasive computing middleware, the VSL (Ch. 5, Ch. 6). The VSL is a context-provisioning middleware with elements for direct inter-service communication. The DS2OS framework is implemented using the VSL (Ch. 7). It adds functionality for *service management* to the VSL. DS2OS contains the VSL as central component.

DS2OS provides a hybrid of the functionality typically provided by *Service management* middleware and *Context management* middleware. Therefore the state of the art pervasive computing middleware is compared to DS2OS as superset of the VSL in the following. The assessment of DS2OS according to the criteria presented next can be found in Sec. 7.7.1.

Assessment Criteria

The pervasive computing middleware designs are assessed according to the following criteria that reflect the essence of the criteria from the different surveys.

After the description of each criterion a mapping to the most correlated requirements that were identified in the analysis section (Ch. 3) is given in parenthesis. The mapping shows that each survey identified a subset of the criteria identified in this work.

The assessment criteria are structures into the four groups *system design*, *context management*, *service management*, and *real world deployment support*. Fig. 4.2 shows the mapping of the requirements from Ch. 3.12 to the assessment groups.

The following criteria are related to the *system design* of a middleware:

- **System Architecture**. The middleware architecture can be a central server (`c`), a central server that coordinates distributed components (`c+`), and P2P with distributed components that act as clients and server (`p2p`). (<R.11>)
- **Communication Technology**. The communication technology section contains information about the way system components are communicating in the middleware design. Options are `RPC` for Remote Procedure Call (RPC), `RMI` for Remote Method Invocation (RMI), `web` for Hyper Text Transfer Protocol (HTTP) based communication, `CORBA` for Common Object Request Broker (CORBA), `OSGI` for Open System gateway Initiative (OSGI), and `o` for other technology. (<R.11>)
- **System Organization**. The system organization can be `v` for a *vertical design*, with vertical communication over layers and different components within one layer (see Fig. 2.6) typically not cooperating, and `h` for *horizontal design* where components are typically objects that encapsulate partial functionality and collaborate via horizontal communication within a layer. A combination of both designs (`vh`) means that the system implements a vertical design but components on one layer can communicate. (<R.11>)

The following criteria are related to the *management of context*:

- **Context Model & Representations**. The context model and its representation (Sec. 3.9) can be markup scheme based (`m`), object oriented (`o`), and ontology based (`Ont`). `Ont` expresses that additional relationships to type-based inheritance can be expressed with the meta model (Sec. 3.9). Several middleware designs do not have an explicit context model. (<R.40>)

- **Extensible Smart Devices support** describes if support for diverse Smart Devices can be added to the system, or if the set of supported Smart Devices is fixed. (<R.23>, <R.42>)
- **Dynamic creation of context models** describes if context models can be added to the system or if they are fixed. This is relevant for orchestration services as it defines which information can be exchanged via context over the middleware. For adaptation services it is relevant as it defines which Smart Device properties can be represented as context. (<R.23>, <R.40>, <R.42>, <R.43>)
- **Support for context convergence** describes if the middleware provides mechanisms to converge/ standardize context models (Sec. 3.3.2). This is relevant for the portability of services. Only when Smart Devices have unified (converged, standardized) interfaces, services can run in different Smart Spaces with different instances of Smart Devices that provide similar functionality (e.g. light). (<R.6>, <R.10>, <R.8>, <R.40>, <R.42>)
- **Support for Semantic Extensibility** describes if the semantics that the system supports (e.g. dependencies) can be added to the middleware dynamically or if it is fixed. (<R.23>, <R.40>, <R.42>, <R.43>)
- **Support for historic context** describes if historic context versions are stored and can be obtained via the middleware. (<R.28>, <R.40>)

The following criteria are related to the *service management*:

- **Fixed service API** describes if services can use a fixed API to exchange information, or if functionality in services is interfaced via varying function calls. (<R.6>, <R.26>)
- **Inter-service communication** identifies if services can communicate with each other directly or over the middleware. Inter-service communication is the basis for a modular software design when developing applications (Sec. 3.8). The communication modes (Sec. 3.4) of the inter-service communication can be `d` for direct (synchronous, referentially coupled), `m` for message based (referentially coupled), `p` for group based (synchronous, referentially uncoupled), and `g` for generative (asynchronous, referentially uncoupled). (<R.23>, <R.26>, <R.25>, <R.31> - <R.38>, <R.19>, <R.20>, <R.21>, <R.22>)
- **μ-middleware** describes if the middleware only provides basic, use case independent functionality or if it provides use case specific support functionality. In addition it describes if the functionality of the middleware can be dynamically extended via adding services at run time. See Sec. 6.2, Sec. 4.5. (<R.23>, <R.26>)
- **Service management** describes if the middleware framework provides functionality to manage services. This includes deployment, monitoring, and optimization of services at run time. (<R.3>, <R.23>)

The following criteria are relevant for the *real world usability and deployment of a solution*:

- **Scalability** refers to the ability of a middleware to scale with a big amount services resulting in a big amount of context data where context is managed. (<R.11>, <R.50>)
- **Security** identifies if the solution provides security mechanisms such as authentication, authorization, or encryption. Validation of context models, abstract interfaces to services, and services is also relevant for securing future Smart Spaces. (<R.39>, <R.44>, <R.30>, <R.3>, <R.49>)
- **Support for user-based development** describes if the middleware framework is designed to support users in the implementation of pervasive computing scenarios. Such support comprises software services that implement pervasive computing scenarios, and services that adapt Smart Device. The term "user" refers to people that are typically neither domain experts nor programming experts. (<R.1>, <R.3>, <R.4>, <R.24>, <R.26>, <R.27>, <R.29>, <R.41>, <R.42>, <R.47>, <R.46>, <R.48>, <R.30>)

Fig. 4.2 shows a mapping between the requirements that were identified in Ch. 3 and the four domains that are used for assessing of the state of the art.

Structure

In the following 13 middleware designs are presented. The selection is based on the surveys [HIM05a, EBM05, HR06, BDR07, Kjæ07, CD07, HSK09, GGS09, Kru09, SHB10, BBH$^+$10, Pos11, CD12, BCFF12, RCKZ12, KKR$^+$13].

First typical middleware as it was developed for many research projects in the past is presented. Such middleware is tailored to a specific scenario and can hardly be reused.

The presentation of generic middleware designs begins with middleware that focuses on *service management*, then middleware that focuses on *context management* is presented. Service management and context management are the typical tasks of pervasive computing middleware.

Each middleware is presented following the same structure:

- A **short description** of the general properties such as the motivation, and the general functionality of a pervasive computing middleware is given.
- **System Design** discusses the aspects concerning the *system design* as described above.
- **Context Management** discusses the aspects concerning the *context management* as described above.
- **Service Management** discusses the aspects concerning the *service management* as described above.
- **Real World Deployment Support** discusses the aspects concerning the *real world usability and deployment of a solution* as described above.
- The paragraph **comparison to DS2OS** discusses *major difference to the DS2OS* middleware framework.

Table 4.1 shows the results of the state of the art analysises. Though shown on the bottom of the table already, the assessment of the DS2OS framework follows in Sec. 7.7.1.

4.5.2 Typical Pervasive Computing Middleware

Many middleware solutions exist that were developed for specific problems in specific domains since 1991. Such middleware is characterized by providing high usability for the specific scenario it was developed for and typically no reusability for other use cases.

As an example, a middleware for creating tour guides is assessed as it was selected as relevant middleware in the survey [KKR$^+$13]. It is representative for the described class of middleware that solves a specific problem. More examples can be found in [EBM05, BDR07, KKR$^+$13, RCKZ12].

The assessed middleware is the Context-Awareness Sub-Structure (CASS) [PF04]. It provides support for creating tour guides. CASS aims to provide a solution for context management that facilitates the development of context-aware applications on resource limited devices such as hand-held computers.

The application domain requires the correlation of different states to present information to a user based on the correlation. Consequently the CASS middleware supports this functionality. The provided example is an inference from a rain sensor reporting “`wet`”, a brightness sensor reporting “`dull`”, and a temperature sensor reporting “`cold`”. CASS supports reasoning about this context via defining conditions and actions. The result of the described context constellation is the recommendation of an `indoor activity` by a tour guide service in the example.

System Design

The system is based on a central server. The organization is vertical. Services act as clients that use the functionality offered by the middleware.

Context Management

The context management happens in a database. There are no context models. Though not described, the paper suggests to have historic context.

Service Management

Services are not managed. They act as consumers of the context that is managed by the middleware

Real World Deployment Support

Due to its limited functionality the system scales with the performance of the database. The centralized architecture limits the possible performance.

No security mechanisms are provided.

User-based development for the specific workflow that was described above is included. Other use cases are not supported. No sharing mechanisms for services are provided.

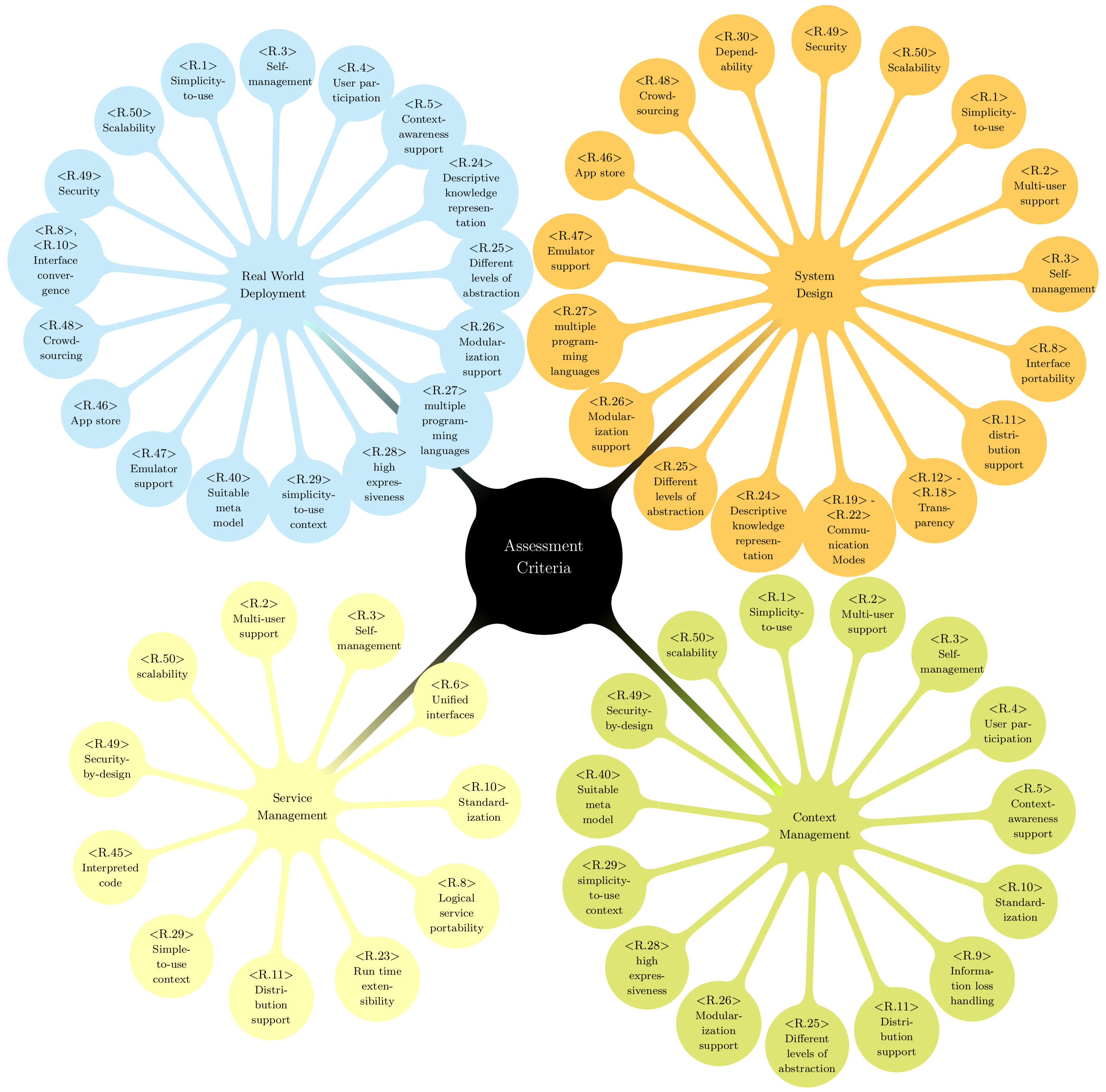

Assessment Criteria
Real World Deployment
<R.1> Simplicity-to-use
<R.3> Self-management
<R.4> User participation
<R.5> Context-awareness support
<R.24> Descriptive knowledge representation
<R.25> Different levels of abstraction
<R.26> Modularization support
<R.27> multiple programming languages
<R.28> high expressiveness
<R.29> simplicity-to-use context
<R.40> Suitable meta model
<R.47> Emulator support
<R.46> App store
<R.48> Crowd-sourcing
<R.8>, <R.10> Interface convergence
<R.49> Security
<R.50> Scalability
System Design
<R.49> Security
<R.50> Scalability
<R.1> Simplicity-to-use
<R.2> Multi-user support
<R.3> Self-management
<R.8> Interface portability
<R.11> distribution support
<R.12> - <R.18> Transparency
<R.19> - <R.22> Communication Modes
<R.24> Descriptive knowledge representation
<R.25> Different levels of abstraction
<R.26> Modularization support
<R.27> multiple programming languages
<R.47> Emulator support
<R.46> App store
<R.48> Crowd-sourcing
<R.30> Dependability
Service Management
<R.2> Multi-user support
<R.3> Self-management
<R.6> Unified interfaces
<R.10> Standardization
<R.8> Logical service portability
<R.23> Run time extensibility
<R.11> Distribution support
<R.29> Simple-to-use context
<R.45> Interpreted code
<R.49> Security-by-design
<R.50> scalability
Context Management
<R.50> scalability
<R.1> Simplicity-to-use
<R.2> Multi-user support
<R.3> Self-management
<R.4> User participation
<R.5> Context-awareness support
<R.10> Standardization
<R.9> Information loss handling
<R.11> Distribution support
<R.25> Different levels of abstraction
<R.26> Modularization support
<R.28> high expressiveness
<R.29> simplicity-to-use context
<R.40> Suitable meta model
<R.49> Security-by-design

Figure 4.2: Mapping between the requirements from Ch. 3 and the functional areas that are used for the assessment of the state of the art.

Middleware/ Project	System Design			Context Management						Service Management				World		
	System Architecture¶	Communication Technology‡	System Organization§	Context Meta Model†	Extensible Smart Device supp.*	Dyn. Creation of Cntxt Models*	Support for Context Convergence*	Semantic Extensibility*	Support for Historic Ctxt*	Fixed Service API*	Inter-Service Communication‖	μ-middleware*	Service Management*	Scalability*	Security*	Supp. for User-Based Developm.*
Typical middleware	c		v	-	(✓)	-	-	-	(✓)	✓	-	-	-	(✓)	-	(-)
Gaia OS [RHC$^+$02]	c+	CORBA, RPC	v	(M) (files)	✓	-	-	-	(-)	-	dp	-	(✓)	(✓)	(✓)	(✓)
Aura [SG03]	c+	(*)	h	M	-	-	-	-	-	(✓)	d	-	(✓)	(✓)	(✓)	-
HomeOS [DMA$^+$12]	c	o	v	-	✓	-	-	-	-	-	-		(✓)	-	✓	✓
CORTEX [SWS$^+$04]	p2p	o	h	(M)	(✓)	(-)	-	(✓)	-	✓	p	-	-	✓	(-)	-
BOSS [TKDHC13]	p2p	web	v	M	✓	(✓)	-	-	(✓)	-	-	(✓)	-	(✓)	(✓)	(✓)
Context Toolkit [DSFA99]	p2p	web	v	(M)	✓	-	-	-	(✓)	(✓)	-	(✓)	-	(✓)	-	(✓)
SOCAM [GPZ05]	c+	RMI, OSGI	h	Ont	✓	(✓)	(✓)	✓	(-)	✓	-	-	(✓)	-	-	(✓)
JCAF [Bar05]	p2p	RMI	h	oo, Ont	✓	((✓))	-	-	((✓))	✓	pg	(✓)	(✓)	(✓)	(✓)	(✓)
PACE [HI04]	c+	RMI, (*)	vh	oo, Ont	✓	(✓)	-	-	(-)	(✓)	pg	-	-	(✓)	(✓)	(✓)
OPEN [GZI10]	c	(web)	v	Ont	✓	✓	(✓)	(✓)	(-)	✓	-	-	✓	-	-	✓
One.World [GDLM04]	p2p	o	vh	(M)	✓	(✓)	-	-	-	✓	m	(✓)	(✓)	-	(✓)	(✓)
ACE [ZPOW11]			v	Ont						(✓)	-	-	-	✓	(✓)	-
DS2OS	p2p	RPC, (*)	vh	M, oo, Ont	✓	✓	✓	✓	✓	✓	dmpg	✓	✓	✓	✓	✓

__ = not applicable or unknown, i.e. not discussed in available literature.
‡ web=web based; RPC=Remote Procedure Calls; RMI=Remote Method Invocation; CORBA=Common Object Request Broker Architecture; OSGI=OSGi based; o=other; *=various interfaces can be generated.
§ v=vertical design, h=horizontal design.
¶ c=Central Server, c+=Central Server with Distributed Components, p2p=Peer-to-Peer
† m = Markup Scheme; oo = Object Oriented; ont = Ontology
* ✓=support; (✓)=limited support; (-)=no support but could be easily added; -=no support.
‖ d=direct; m=message based; p=group based; g=generative; see Sec. 3.4.

Table 4.1: Comparison of different ubiquitous computing middleware.

Comparison to DS2OS

Like most of the related work presented here, DS2OS aims to provide a middleware that is usable for implementing diverse pervasive computing scenarios. CASS does not fulfill this property.

4.5.3 Gaia OS

The Gaia project [RC00, RKC01, RHC$^+$02, Rom02] shares the goal with this thesis, to provide a suitable software infrastructure for developing ubiquitous computing applications [RHC$^+$02]. The Smart Spaces that are operated by Gaia are called active spaces.

Gaia associates services with users in their *user virtual space* that moves with the user. Gaia provides the operating system basis for a multi-agent system where software agents coexist with users to support them.

System Design

Gaia is implemented on top of CORBA that uses RPC, and it allows direct RPC between services for communication. The authors mention that it could as well be implemented using Service Oriented Architecture Protocol (SOAP) and RMI.

Gaia uses a layered design of coexisting core services that are accessed vertically by services.

Context Management

Gaia provides context management over its *context file system*. It does not have a formal context model. Context is typed. Context of the same type is aggregated to the same virtual directory in the context file system [HC03b].

Gaia allows to associate context to files as additional meta data [RHC$^+$02, HC03b, HC03a]. Context is accessed via hierarchically structured file names in a file system directory tree. As the used file names are not restricted, diverse context can be addressed.

Convergence support for context models is not provided by Gaia. No meta model that could be semantically extended is used. The file meta data consists of key-value pairs.

From the literature it is not clear if context is versioned. As a file system is used as basis, this feature may depend on the file system of the used Operating System (OS). No special functions to access historic context are described for Gaia.

Service Management

The access to the Gaia context file system is done via fixed functions. Services can offer any kind of interface.

Services can communicate using event based asynchronous mailbox communication (p) or synchronous direct communication (d) based on RPC [RHC$^+$02].

Gaia includes in its kernel a *space repository service*, an *event manager service*, a *context file system*, a *presence service*, and a *context service* [RHC+02]. Therefore it is not a μ-middleware (Sec. 6.2).

Gaia provides local resource management for services and devices [RHC+02].

Real World Deployment Support

Gaia is a distributed system and uses the file system as base. Therefore it scales. Gaia has central components that reduce the scalability.

Gaia supports the collection and processing of error reports about software and hardware failures which enhances the reliability of Gaia operated *active spaces* [RHC+02].

Gaia is designed to support user-based development but it does not provide mechanisms to share software over a repository [RC00, RHC+02]

Comparison to DS2OS

The central entities of Gaia are services that move with the user and are mapped to locally available resources. The central entity of DS2OS is context that stays with a Smart Space while users move in and out. The arrival or departure of users may trigger the start of specific software in DS2OS similar to Gaia (Sec. 5.6.6).

4.5.4 Aura

Aura [SG03, SG02, GSSS02] provides automatic configuration and reconfiguration of software in ubiquitous computing environments according to a user's tasks and intents.

Aura contains a component for resource monitoring and application adaptation (*Odyssey*). It provides a file storage back-end (*Coda*), and it manages remote execution of software (*Spectra*).

Though not explicitly listed in the table, another research project follows a similar goal. The Seamless Service Composition (SeSCo) as part of the Pervasive Information Community Organization (PICO) implements components for managing and enabling the composition of existing services (e.g. a word processor) [KKS05, KSD+03, KK07].

System Design

Aura consists of different components that can run on distributed machines. Aura is built to support diverse communication protocols to address heterogeneity. Functionality is implemented by orchestrating different existing services on distributed hosts. Therefore, a horizontal design is implemented.

Aura consists of the *Prism task manager* that manages services that belong to a user, the *Context Observer* that provides information about the physical context, the *Environment Manager* that provides gateways to the environment, and *Suppliers* that act as building blocks (e.g. text editing, video playing) for implementing Aura services.

In Aura, services communicate over so-called *ports* that implement direct communication (d).

Prism moves tasks that are associated with a user when it learns via the *Context Observer* that a user moved to another physical environment. The *Suppliers* provide abstract interfaces to services. They hide the heterogeneity of the orchestrated application, e.g. using the editor xemacs in one space and the editor Microsoft Word over the same editor programming interface in another environment.

Instances of one of the described types (*Context Observer*, *Suppliers*, etc.) have fixed APIs.

Context Management

The abstract interfaces to services are described via Extensible Markup Language (XML) markup.

No details about the *Context Observer* and its context model are given. Context representation is not in the focus of Aura.

As distributed system, Aura scales. The use of one instance of the central components per Smart Space [SG03] might be limiting the scalability.

Service Management

The *Prism* task manager manages all tasks that belong to a user. It orchestrates the services that are associated with the *Suppliers*. Services in Aura orchestrate applications not Smart Devices.

Suppliers use a fixed API. Capabilities of the managed applications are mapped to so-called *task layouts*. Following its task-oriented paradigm, Aura allows to connect services by specifying a graph of relations. The Aura *Environment Manager* tries to interconnect the specified services according to such a graph, e.g. speech recognition > language translation > speech synthesis. This service connection is transparent for the services that get connected. Services are not aware of being inter-connected allowing the use of standard software (e.g. Microsoft Word).

Aura only allows to create applications in the limited range that is defined by the service composition. It allows configuration and executes all necessary tasks. Therefore it is not a μ-middleware.

Aura manages its orchestration services at run time. Deployment or dissemination mechanisms are not presented in the literature.

Real World Deployment Support

Though not described in the context of the presented functionality, the underlying file system primitives allow encryption of data and the communication that is used to exchange them [GSSS02]. This could be used to provide privacy and access control.

In the limited scope of defining simple orchestration workflows on the connected applications, experts can create Aura services.

Comparison to DS2OS

Aura facilitates the orchestration of computer programs while DS2OS can be used to orchestrate all entities including physical environments, and software.

The principal entity of Aura is a user task while the principal entity of DS2OS is the context of an entity that is independent of the tasks that create it. Aura is activity based, DS2OS is state based.

Aura migrates services with users. Services are bound to users. In DS2OS, services are bound to the Smart Space. Users move in and out a space which changes the context of the Smart Space. The services remain in the Smart Space and typically do not leave it with users.

4.5.5 HomeOS

HomeOS [DMA+12, DMA+10] focuses on home environments. It unifies the interfaces of distributed heterogeneous devices in homes, and protects the access to and use of them. The focus is on entertainment. The system is developed at Microsoft and follows the architecture of a Microsoft PC OS.

HomeOS mentions the concept of an App store (Home store) to be required for future home automation (<R.46>).

System Design

HomeOS runs on a single host and offers access to distributed peripherals via locally running device drivers.

The system is designed for vertical communication following the PC OS analogy. HomeOS is implemented in C#.

Context Management

HomeOS does not manage context of a Smart Space. It only manages access rules that define device access policies (see below). New devices can be connected via device drivers. The communication happens via system calls in the OS.

Service Management

The abstract interfaces of HomeOS drivers are not unified. Drivers can expose diverse function signatures. Inter-service communication is not intended. HomeOS only provides connectivity and access control which makes it a μ-middleware.

Though no implementation details are presented in the literature, the Home store concept can be expected to provide functionality to deploy HomeOS Apps into homes. In the described concept, it checks the compatibility with the available devices and does quality assurance of services.

Real World Deployment Support

HomeOS scales with the resources of the PC it is running on. Those resources are limited by the hardware configuration and the running applications.

HomeOS secures the access to drivers via datalog rules [AG89] that model entities and time. Access to entities is controlled based on *capabilities* that can be delegated to applications at run time. Validation of software happens with the mechanisms of the .net programming framework that can be used over the Visual Studio Integrated Development Environment (IDE).

HomeOS facilitates the access to distributed devices for experts that are used to write applications for PCs. The use of C# and .net facilitates the creation of applications in the different programming languages that belong to the .net-family.

Comparison to DS2OS

In contrast to DS2OS, HomeOS does not provide context management and runs only on a single PC. HomeOS manages device access while DS2OS provides an infrastructure for implementing diverse scenarios with diverse entities in software and hardware in future Smart Spaces.

4.5.6 CORTEX

The CORTEX (CO-operating Real-time senTient objects: architecture and EXperimental evaluation) [SWS+04] middleware facilitates the implementation of a dynamic context overlay of consumers and producers that communicate over multicast via publish-subscribe. The project is motivated by the use case of car-to-car communication.

CORTEX provides sensor fusion (sensor data aggregation) and inference. It focuses on Quality of Service (QoS) for real-time interaction. A CORTEX system consists of distributed software components that couple in an ad hoc way. The distributed components that produce and consume context are called sentient objects.

System Design

CORTEX is designed as P2P system that uses a multicast based publish-subscribe mechanism for communication. Sentient objects spontaneously connect and implement intelligence bottom-up.

Context Management

CORTEX uses a publish-subscribe mechanism based on locality and on groups to disseminate current context events. CORTEX does not provide context repository functionality.

Context is represented in XML. The meta model is a generic XML profile. Context model are not mentioned in the literature.

Service Management

CORTEX scales as all sentient objects are autonomous.

Sentient objects use the publish-subscribe event bus as fixed interface for inter-process communication (p). With its inference and sensor fusion functionality CORTEX is not a μ-middleware.

Services are not managed. They group dynamically in an ad-hoc way.

Real World Deployment Support

No security features are discussed in literature.

Though developer support is given via so-called component frameworks, the middleware targets expert developers.

Comparison to DS2OS

The design goals of CORTEX and DS2OS are orthogonal. CORTEX focuses on real-time ad hoc context exchange in mobile environments for sharing current information via services created by experts.

DS2OS focuses on providing current and past context in less dynamic environments, service interaction, convergence, and user-based development.

4.5.7 Building Operating System Services (BOSS)

The Building Operating System Services (BOSS) [TKDHC13, KFKC12, DHKT^{+}13] targets the automation of professional buildings (Sec. 3.2.1). The use case of the project is saving energy.

BOSS creates an abstraction on top of the Smart Devices of professionally used buildings (Sec. 3.2.1) to enable the portability of orchestration services.

System Design

BOSS is implemented as layered system with distributed components for different functionality (e.g. transactions, context storage). It implements a vertical architecture.

The BOSS middleware provides authentication, and so-called *transaction managers* that control the Smart Device access. It provides so-called *time series services* that act as context repository for selected data.

The system is implemented in Python and C. It uses web technology for communication (JavaScript Object Notation (JSON) over HTTP).

Context Management

BOSS uses different context models on different layers of abstraction. On the lowest layer, the *Hardware Presentation Layer*, an XML based markup is used. The technology used to interface hardware is an open-source project called *sMAP*. All driver services inherit from a basic interface.

On the above *Hardware Abstraction Layer*, spatial and functional relations of Smart Devices are stored as a directed graph of objects. BOSS uses *types*, *attributes*, *functional relation*, and *spatial relation* as *primary context* to select context (Sec. 5.2.12).

Context models are not published to a directory. Therefore there is no explicit convergence. The use of the drivers as aggregation point for similar functionality implements an implicit convergence process.

The BOSS meta model supports fixed semantics with spatial and functional relationships. The abstract interfaces of the drivers are not fixed and can be extended. The abstract interfaces are only implicitly specified within the driver implementations. No explicit abstract service interfaces are specified.

A *history service* can explicitly be associated with a context source. It provides context versioning for that source. BOSS does not automatically version context.

Service Management

BOSS uses a function-based interface (e.g. `get_speed();`) to access device functionality. Only the interface to the *time series services* is fixed.

There is no inter-service communication. BOSS only provides basic functionality that is related to context management but it does not provide extensibility of the core functionality via services. BOSS is partly (only basic functionality) a μ-middleware.

No functionality for managing services is provided.

Real World Deployment Support

BOSS scales as it is distributed and as it uses a markup-based context representation.

Though no details are given in the literature, BOSS is described to provide access control.

Writing orchestration services for professionally used buildings is facilitated by BOSS but requires expert knowledge. No infrastructure for sharing services between spaces is presented.

Comparison to DS2OS

In contrast to DS2OS, BOSS does not provide inter-service communication with its vertical design. The sMAP drivers implement a de-facto standardization of Smart Device interfaces. No explicit convergence processes are provided.

BOSS follows a mash up approach for implementing its middleware functionality. Distributed isolated services are connected to a whole. This is similar to BACnet

(Sec. 4.4.2). DS2OS provides all necessary middleware functionality in distributed peers.

The BOSS approach could lead to problems for larger systems as central components could be overloaded or, if distributed, synchronization could become a relevant issue. In addition the distribution of core functionality in distinct modules requires multiple components and multiple communication paths to be secured. These issues are solved with the DS2OS design of distributed peers that synchronize autonomously and encapsulate all middleware functionality (Sec. 6.4).

Context models do not have an exposed meaning in BOSS which limits the reusability and the convergence of abstract interfaces to Smart Devices. Services cannot be coupled or reused to mash up complex functionality in BOSS.

4.5.8 Context Toolkit

Context Toolkit [DSFA99] is one of the first middleware designs for facilitating the creation of context-aware services that implement pervasive computing scenarios. It provides an object-oriented architecture to support rapid prototyping of context-aware applications.

A major contribution of the work is that it identifies the advantage of separating sensor context acquisition from the application logic by using a unified interface for the *Widgets* that interface diverse Smart Devices.

The separation of service logic from service state facilitates the creation of orchestration services and enables portability via exchanging information only via context and offering a uniform interface to access the context. Services that provide abstract interfaces to Smart Devices are called *context widgets* in Context Toolkit.

System Design

Context Toolkit is implemented in Java. It supports distributed *context widgets* on a central host [DSFA99].

Communication is implemented in XML over HTTP using polling and notifications.

Context Management

Context Toolkit *widgets* act as repositories for their context. Context Toolkit supports *interpretation* and *aggregation* of context in the middleware.

Interpretation is used to transform context between different representations (Sec. 3.3). *Aggregation* is used to collect information about an entity from different, possibly distributed, *widgets.*

Widgets can interface drivers for Smart Devices. Context Toolkit does not use an explicit context model. Context is versioned in the *Widgets.*

Service Management

All services inherit from a so-called `BaseObject`. This makes them interface compatible. There is no inter-service communication.

Context Toolkit provides *interpreters* and *aggregators* that provide interface transparency and location transparency. Context Toolkit provides only basic functionality for context access. It does not provide extensibility of the core functionality via services making it partly (only basic functionality) a μ-middleware.

Services are not managed by Context Toolkit.

Real World Deployment Support

Context Toolkit scales as the components can be run on distributed hosts. The use of *aggregators* for collecting all context that is related to an entity may lead to bottlenecks when a big amount of context is available.

Security and privacy are identified as relevant but methods to provide them are not implemented.

Context Toolkit facilitates the implementation of pervasive computing scenarios for experts. Services are not managed.

Comparison to DS2OS

Published in 1999, Context Toolkit shares several relevant design considerations with DS2OS that are not considered for many middleware designs that emerged after 1999.

Like Context Toolkit, the VSL supports the principle to couple entities over their exposed context (Sec. 5.3.2). Like Context Toolkit, the VSL propagates a separation of service logic from service state to facilitate the implementation of Smart Space services (Sec. 5.6.5).

Major differences are:

Context Toolkit does not use explicit context models.

Context Toolkit separates between adaptation services (context widgets) and orchestration services (applications) which DS2OS does not do (Sec. 3.1.2).

Inter-service communication between orchestration services is not supported in Context Toolkit.

4.5.9 Service-Oriented Context-Aware Middleware (SOCAM)

The Service-Oriented Context-Aware Middleware (SOCAM) [GPZ05] aims to support the building and rapid prototyping of context-aware services.

SOCAM divides pervasive computing into different context domains (ontologies) such as vehicle, or *home*. The central concept of SOCAM is that context moves with the user through different domains.

Each SOCAM domain has specific context models. The models of the upper ontology (Sec. 3.9) are shared between the domains providing basic interoperability.

System Design

SOCAM consists of one centralized context databases per domain, and distributed context providers and context consumers.

SOCAM is implemented in Java for supporting heterogeneous platforms. For communication RMI is used. The SOCAM components are implemented as OSGI bundles.

The SOCAM design is layered with the database in the middle, the context providers at the bottom, and context consumers at the top.

Context Management

SOCAM uses Web Ontology Language (OWL) with Resource Description Framework (RDF) (Sec. 3.9) for representing context in the form *Predicate(subject, value)*, e.g. *Location(John, bathroom)*.

The SOCAM ontology is divided into two parts (Sec. 3.9). The upper ontology that contains general concepts, and different domain-specific lower level ontologies. Only the upper ontology is shared between all domains.

Context models can be dynamically created and bound at run time. The upper ontology is the common denominator between all domains and acts as standardization for a subset of the context of a domain. No further sharing or convergence mechanisms for ontologies are provided.

The use of OWL allows extending the meta model with additional semantics. After a change, the meta model must be exchanged in all SOCAM spaces.

No API functions to access historic context are presented in the paper.

Service Management

SOCAM provides a *service locating service* that allows services to discover each other (probably based on identifiers).

The functional interface consists of a fixed API with the functionality to query, add, delete or modify context.

The SOCAM core contains rule-based first order logic reasoning capabilities therefore it is not a μ-middleware.

Real World Deployment Support

The authors divide the context representation into an upper and a lower ontology for enhancing performance of the ontology processing. However, the processing of OWL ontologies does not scale with larger ontologies (Sec. 3.9) [KKR$^+$13].

No security or privacy features are introduced.

The creation of user-based orchestration services is facilitated via configuring the reasoner with logic expressions that lead to actions similar to the DS2OS trigger service presented in Sec. 8.4.1. The creation of the ontologies needs expert knowledge. Though the ontologies can be locally extended, convergence mechanisms are missing.

Comparison to DS2OS

In contrast to DS2OS, SOCAM separates between context providers and context consumers. User-based services are context consumers that do not communicate with each other. Context providers are part of the middleware layer. In DS2OS all services are outside the middleware. They can produce and consume context, and use it for communication with each other.

With the distinct ontology domains that only share a small subset of semantics via the upper ontology (Sec. 3.9), SOCAM lifts the diversity problem of pervasive computing on a higher level of abstraction by creating isolated domains in the ontology layer (Sec. 4.5). DS2OS instead has only one ontology that is created collaboratively and that comprises all domains (Sec. 5.2.8).

4.5.10 Java Context Awareness Framework (JCAF)

The Java Context Awareness Framework (JCAF) [Bar05] faces the problem that most existing pervasive computing middleware designs only lift the heterogeneity of the Smart Devices to the middleware layer (middleware silos, Sec. 4.5). The paper describes the phenomenon as emergence of dedicated domain-specific context-awareness sub-systems. JCAF aims to provide a framework that is generic enough to be usable for different domains.

System Design

JCAF is implemented in Java as P2P system. JCAF services are distributed. Services are implemented as Java objects that communicate with each other over RMI.

Context Management

JCAF has no specific context storage by itself. Context is stored in the distributed services. For context discovery the standard RMI registry is used. JCAF provides a fixed `ContextService` interface. Between services, a topic-based (*event*) publish-subscribe architecture is implemented via JCAF.

Context is represented as Java Objects which allows the use of inheritance. Additional relations such as *located in* can be modeled. JCAF implements an object-based ontology.

Each JCAF service can implement its own context model. This allows to add and remove contexts dynamically at run time. There is no shared directory for context models and therefore no support for convergence. The JCAF meta model is fixed.

In a use case that is described in the paper, a *history module* was implemented that provides access to historic events. As the JCAF architecture does not act as context provider itself, it enables historic context via services but it does not provide it.

Service Management

JCAF implements a Service Oriented Architecture (SOA) (Sec. 3.8). Services are implemented in so-called `EntityContainer`s that provide the JCAF functionality.

JCAF provides a fixed API to access context. Context can be exchanged via group based (p) and generative communication (g). The JCAF architecture does not provide use case-specific functionality. The core functionality can not be extended via services making JCAF partly (only basic functionality) a μ-middleware.

JCAF uses the Java RMI service registry as directory for services to find each other. Service executables are not managed by the system.

Real World Deployment Support

As fully distributed system, JCAF scales. Only the RMI registry could become a bottleneck with many services.

JCAF provides mechanisms for access control and authentication via certificates. Encryption and code signing are not supported (Sec. 7.3).

User-based development is supported via the `EntityContainer`s. The mechanisms are comfortable for experts but probably too complicated for beginners. No sharing mechanisms are provided.

Comparison to DS2OS

The goals of JCAF and DS2OS are similar. A major difference is that DS2OS separates the context from the logic of services while JCAF encapsulates both in the `EntityContainer`s.

An additional important difference is that DS2OS provides context management including context storage. This takes complexity away from service developers.

4.5.11 Pervasive Autonomic Context-aware Environments (PACE)

The Pervasive Autonomic Context-aware Environments (PACE) middleware [HIM05a, HI04] aims to facilitate the prototyping of context-aware applications. PACE consists of the six layers shown in Fig. 2.6.

It provides direct support for applications that use *branching* as method to make a context-dependent choice among a set of alternatives, and *triggering* as asynchronous event-based method to trigger actions on transitions of context states based on Event-Condition-Action (ECA) rules.

System Design

PACE is built as a central [HI04] or independent distributed [HIM05a] context repositories.

The central version uses Java RMI and offers a web interface that can be used via XML commands over HTTP. The distributed version uses a publish-subscribe content-based message routing scheme.

As shown in Fig. 2.6, PACE uses a layered design, supports events, and allows horizontal communication over context and the mechanisms of the routing scheme.

Context Management

The center of PACE is the context repository. It represents context as relation tuples [HIM05b]. An example for a relationship tuple is "*Person* `is located at` *Location*". Context information is stored in a relational database.

Context models can dynamically be added by services at run time. PACE does not have a Context Model Repository (CMR) (Sec. 5.2.8) and does not support convergence therefore. The PACE meta model cannot be extended with additional semantics [HIM05b].

No API for querying historic context is presented in the literature.

Service Management

The possibility to use the context repository for inter-service communication results in a fixed API. Direct communication over the message broker [SA97] also has a fixed API but the subscription topics are not shared over a repository. This makes it necessary that services know the subscription topics in advance.

Inter-service communication is not directed. Via *triggering*, synchronous communication can be implemented (g). Via storing context, asynchronous communication can be implemented (p).

With the support for the *branching* and *triggering* workflows, PACE is not a μ-middleware.

Real World Deployment Support

PACE exists as centralized and distributed version (see above). In the distributed version the system scalability is only limited by the scalability of the used publish-subscribe message broker [SA97].

A PACE schema compiler validates context models before instantiation. PACE provides access control based on preferences that express situations in which information should be revealed [HIM05a]. Authentication and further access control are not provided.

Support for different programming languages is conceptually provided. A Java programming toolkit is provided that facilitates PACE application development. The sharing of services is not supported.

Comparison to DS2OS

Like PACE, DS2OS provides mechanisms to validate context models (Sec. 5.5.6).

In contrast to DS2OS, PACE does not offer a CMR. This makes the (re-)use of context between services difficult.

In contrast to DS2OS, PACE separates between adaptation services interfaces, middleware services interfaces, and orchestration services interfaces. This complicates crowd-sourced development and flexible coupling of functionality (Sec. 5.4.1, <R.6>).

4.5.12 OPEN

OPEN [GZI10] targets rapid prototyping, sharing, and personalization of context-aware applications. The project aims to support a diverse group of developers with different skill levels.

Orchestration services in OPEN are represented as ECA rules (Sec. 3.6.4). According to the skill level of developers, ECA rules can be created (expert developer), combined (medium developer), or parameterized (beginner developer).

System Design

The OPEN middleware and all user services run on a central server. OPEN implements a vertical design. All services run inside the middleware not requiring additional communication. Services can be developed and managed on the server over a web interface.

Context Management

OPEN uses an OWL ontology as context model (Sec. 3.9). OPEN context models are stored and disseminated from a central repository to distributed Smart Spaces. An OPEN Smart Space contains a central context repository that is implemented as one database.

The context models of the ontology, and the upper ontology are maintained by experts. Smart Devices are connected via so-called *Wrappers* that are maintained by experts like the context models. Via automated updates of the local ontologies, context models are dynamically added.

Via the experts, context models are converged. Extensions of the OPEN meta model may be possible via the central distribution mechanism like the context models. The paper does not provide such information. There is no information about context versioning.

Service Management

OPEN ECA rules can be designed and configured via a unified web interface. The syntax is fixed.

Services are pure consumers of context that is available in the system. There is no inter-service communication. The middleware contains the entire execution of services. OPEN is the opposite of a μ-middleware.

Via a shared repository, developers and users can exchange their rules. OPEN provides full management of the rules.

Real World Deployment Support

Via the central database and the use of OWL ontologies OPEN does not scale.

Security is not addressed by OPEN.

As described in the introductory text, OPEN provides different mechanisms that foster user-based development.

Comparison to DS2OS

Compared to the previously introduced middleware, OPEN is the first approach that targets real world developers. Like OPEN, DS2OS targets a heterogeneous group of developers and crowdsourced development (Sec. 7.4). Both provide mechanisms for deploying services to Smart Spaces (Sec. 7.2.6).

A fundamental difference is that services are ECA rules in OPEN. This limitation facilitates the described tasks. DS2OS manages the development and distribution of regular Java services. OPEN targets a small user and developer group while DS2OS targets world-scale (Sec. 7.5).

OPEN orchestration services consist of ECA rules only. This limitation is not suitable for implementing diverse pervasive computing scenarios. However, ECA rules are simple to parameterize. The functionality of DS2OS exceeds that of OPEN, enabling the implementation of complex pervasive computing scenarios with simple ECA rules (Sec. 8.4.4).

4.5.13 One.World

One.World [Gri04, GDLM04] targets the use case of a laboratory environment where data should be shared, and guest researchers come and go. Its design goals are supporting context change handling in applications, ad hoc composition of services, and file sharing between applications and devices.

One.World encapsulates services in self-contained *Environments* that contain a service executable and its data. The framework manages the discovery and migration of services.

Context changes are not handled transparently for services. Instead, services should be created in a way that makes them explicitly adapt to environment changes (e.g. migration) by rediscovering resource bindings using the One.World directory service for instance.

System Design

One.World is implemented in Java. Communication happens via asynchronous events. A proprietary message format over a Transmission Control Protocol (TCP) connection is used to exchange information between distributed hosts.

The interaction with the One.World middleware is vertical but the middleware runs distributed on multiple hosts which also leads to horizontal communication for context exchange in the middleware.

Context Management

One.World uses self-describing tuples to represent context. Tuples are implemented as Java objects. No explicit context model representation is used.

One.World synchronizes context between *Environments* that share the same namespace. Context search is supported via data types and values.

Adaptation services that support new Smart Devices can be added at run time. They can be found if their type is known to other services. There is no explicit mechanism for versioning context tuples.

The meta model is fixed and supports typing.

Service Management

One.World uses a fixed API with few methods. Inter-service communication happens over events that are represented in tuples that are exchanged over a fixed API messages (m).

Services are encapsulated in so-called *Environments* that contain the executable and the tuple storage of a service. Resources can be discovered via their descriptors that are registered on a central *discovery server*.

Services can be composed via nesting and via exchanging events between *Environments*. *Event handlers* of remote services can be identified using the *directory*.

One.World only provides basic context handling functionality but the core cannot be extended via services. One.World is partly (only basic functionality, no extensibility) a μ-middleware.

Service migration is supported by moving *Environments* to another host. This is facilitated as an *Environment* comprises the entire service state (Sec. 4.3). Bindings to other resources are deleted on migration. A (global) service repository is not part of One.World.

Real World Deployment Support

The implementation experience with DS2OS shows that Java objects do not scale with large contexts. This was the reason for basing the VSL on a database (Sec. 6.3.5). The directory service (see above) could become a bottleneck.

One.World uses implicit mechanisms to protect context from unauthorized access. Storing contexts in the *Environments* automatically protects context against access from outside. Via using the same namespace for distributed *Environments*, context sharing can be implemented in a controlled way.

Context is only protected against readout from applications running outside One.World by Java. Communication is not encrypted.

The chosen abstraction simplifies the development of services. The missing of an explicit context model and of abstract interfaces that describe the functionality of services make the solution not transparent enough for real-world use. No infrastructure for sharing services is provided.

Comparison to DS2OS

Like DS2OS, One.World explicitly separates between application logic and data and provides full service encapsulation (Sec. 5.6.5).

Major differences are that One.World does not have explicit context models, that it does not provide context management, and that it does not support synchronous communication between services.

4.5.14 Application Context Engine (ACE)

The Application Context Engine (ACE) [ZPOW11] externalizes reasoning logic for acquiring desired context from services and puts this functionality into a middleware-addon.

Services describe their context acquisition logic in OWL (Sec. 3.9) and pass the RDF description (Sec. 3.9.6) to the ACE that validates given constraints when starting and stopping a service and applies the given logic during the run time of the service.

ACE does not contain a context-provisioning middleware. Instead it is intended to run on top of different of the presented middleware frameworks.

Though below it is described how ACE could complement a μ-middleware as a service, the motivation of ACE is that the provided functionality is missing in the existing middleware kernels.

This motivation of the authors illustrates the identified trend of existing pervasive computing middleware to create middleware silos (Sec. 4.5). The sparse population of the μ-middleware row in table 4.1 shows that many designs follow this silo approach. See Sec. 6.2 for a discussion.

System Design

ACE acts as preprocessor between a service implementation and a context-provisioning middleware. It handles the context acquisition and pre-evaluation of context for services.

Context Management

ACE does not manage context but uses context that is managed by its underlying middleware. ACE instances can run independently. As ACE uses an OWL ontology it provides limited scalability.

Service Management

ACE constraints are specified in OWL using a fixed meta model.

Real World Deployment Support

An ACE instance can be started for each service on distributed hosts as it acts as a building block for a service developer. Therefore it scales.

ACE is positioned between a service implementation and a context-provisioning middleware. This position in the middle may introduce security problems as the component is external to a service and external to the middleware. ACE provides limited service management by checking the constraints of a service during its life-time. ACE does not provide additional security or privacy.

The complexity of the constraint specifications in OWL make it difficult to use for non experts.

Comparison to DS2OS

ACE could be run as a filter service in DS2OS that provides the described functionality (Sec. 8.4.1). It could become a valuable building block for DS2OS services.

The presented examples for expressing the ACE constraints are too complex for user-based development. In addition, the assumption for DS2OS services is that their constraints on context acquisition may change over their lifetime. This is not reflected in ACE design where the context acquisition rules are initialized at service start time.

4.5.15 Distributed Smart Space Orchestration System (DS2OS)

The DS2OS middleware framework is evaluated in Sec. 7.7.

4.5.16 Combined State of the Art Reference Model

Combining the main features of all assessed middleware, a combined reference model is introduced. Different reference models for pervasive computing middleware are presented in the surveys [RCKZ12, BCFF12, KKR+13]. Fig. 4.3 shows a combination of the existing reference models and additional functional components that were identified in the state of the art analysis (Sec. 4.5.2). It is explained in the following text. The numbers in brackets refer to the numbers in the figure.

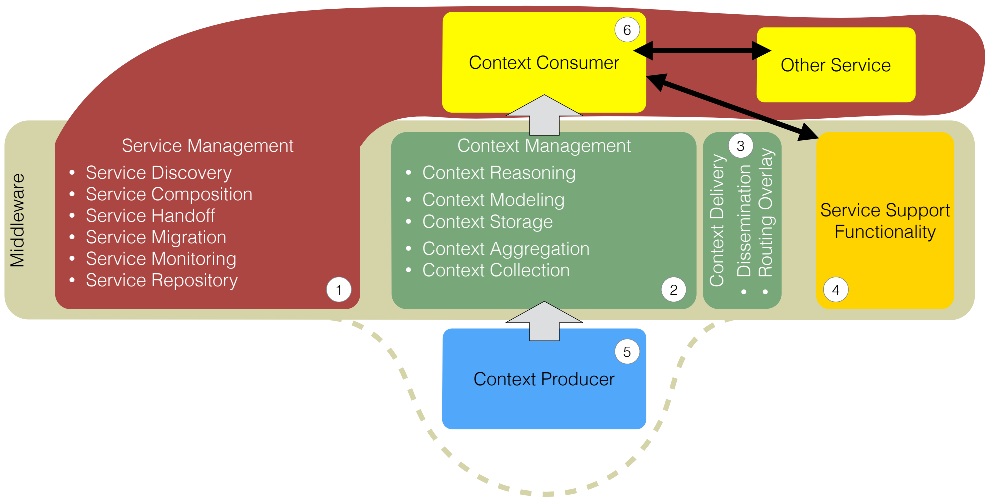


Figure 4.3: A combined reference model for the state of the art with aspects from [RCKZ12, BCFF12, KKR+13].

Middleware that provides *service management* (1), typically offers a subset of the functionalities: service discovery, service composition, service handoff for mobile services, service migration, and a service repository. Managed services are in all assessed cases *orchestration services* that implement application logic. Management for *adaptation services* that interface Smart Devices is not provided by the assessed solutions.

Middleware that provides *context management* (2) typically offers a subset of the functionalities: context reasoning, context modeling, context storage, context aggregation, and context collection.

In case of a distributed design, context dissemination, and context routing are additional tasks for a middleware (3). Several of the presented solutions provide workflow-specific support functionality (4) that is expected to be used as a building block for developing services.

Context is produced by context producers (5). Typical context providers are *adaptation services* that connect Smart Devices to the middleware (Sec. 8.3.1). In several middleware architectures context provider are part of the middleware. The *orchestration services* that implement the pervasive computing scenarios are illustrated as yellow services at the top of Fig. 4.3 (*context consumer*) (6). Often they act as consumer of context only.

The shown functionality is typically provided by monolithic components in case of a *vertical design.* In a *horizontal design* the functionality is typically distributed over multiple modules.

If services can communicate with each other this often happens as shown at the top of Fig. 4.3 between the *context consumer* and the *other service.* In some architectures inter-service communication happens over the middleware either via special communication mechanisms, or using context for exchange in the combined role of context producers and context consumers for bidirectional communication.

Though not shown on the picture, security is relevant for all components for a real world deployment. The assessment in table 4.1 illustrates that about half of the existing solutions provide at least some security features.

To illustrate the differences between the existing middleware designs and DS2OS, the colors, elements, and numbers of Fig. 4.3 are used to illustrate the architecture of DS2OS in Fig. 7.6 in Sec. 7.7.2.

4.5.17 Discussion of the Assessment Result

The results of the assessment that are summarized in table 4.1 are discussed going through the assessment criteria from left to right in the table, following their order introduced in Sec. 4.5.1.

Concerning the *system architecture*, centralized systems have scalability problems for large amounts of services and context. P2P architectures scale better.

The used *communication technology* has influence on the developer's freedom of choosing a programming language to implement their services. It determines the requirements on implementing additional components for communication at the developer side, and required components in the network (e.g. directory server). Systems that can be adapted to multiple communication technologies offer most freedom.

A vertical *system organization* in layers structures the functionality that is needed to implement pervasive computing. A horizontal design is more challenging but it provides better scalability.

The *context meta model* has influence on the performance of a middleware system. Complex context models such as ontologies need more resources to be processed than simple context models that are based on markup schemes (Sec. 3.9). The reason are dependencies that have to be resolved to interpret ontologies. The meta model defines the expressiveness of context models. It defines what can be represented (e.g. type relations, location relations, functional relations).

Most solutions provide *extensible support for Smart Devices* but many solutions do not use explicit *context models*, or abstract service interfaces. This is problematic as the interfaces define how the abstraction of a Smart Devices can be accessed by services. Service developers have to know the context models or abstract service interfaces of services that connect devices for being able to use the corresponding Smart Devices.

Only OPEN (Sec. 4.5.12) and SOCAM (Sec. 4.5.9) support a convergence of context models. Both implement the approach of standardization by experts. As discussed in Sec. 3.3.2, the used classical standardization mechanisms do not scale with the high amount of context models that are needed for real world pervasive computing. A standardization by a small group of people does not meet the requirements of future Smart Spaces.

An *extension of the meta model* raises the level of expressiveness of context models (Sec. 3.9). Only SOCAM implements meta model extensibility. It is based on replacing the meta model with an updated version. This mechanism results in a major change of the ontology of a Smart Space that is difficult to communicate to developers. It is also not dynamic as all meta model in all spaces have to be exchanged to offer the new semantics. This is unlikely to scale.

The availability of *historic context* is important to allow services to learn from the past. Functionality that archives all context is not available in the assessed solutions. However, it could be added to most solutions without major design changes.

The use of a *fixed API for services* facilitates their development. It allows services to use the middleware functionality, or to communicate with each other (e.g. over context) in a unified way. As described for Context Toolkit (Sec. 4.5.8), using context for information exchange simplifies the inter-service communication (Sec. 5.3.1). In addition it inherently makes all services interface compatible which fosters reuse and mash up in a SOA (Sec. 5.6.6).

A problem of using context for coupling services is that it implements asynchronous and static communication via generative or mailbox communication (Sec. 3.4.1). Only context that was put into the context repository can be read. This introduces latency.

Only some of the middleware designs support *inter-service communication*. The lack of inter-service communication prevents modularization and reuse of functionality for other services.

Different inter-service communication modes have different properties as described in Sec. 3.4.1. Temporally coupled mechanisms require all communication partners to exist at the same time, and to process communication messages immediately. The latency of direct coupling is low. Temporally uncoupled communication implements looser coupling and temporal independence of communication partners for the price of higher latency (see Sec. 9.3).

Referential coupling has implications on the security of communication and on the level of control of an interaction. In Smart Spaces it can be relevant specifying who is the receiver of a communication message. The exchanged information could be sensitive (e.g. safety, privacy). The support for different communication modes impacts the possibility to implement diverse pervasive computing scenarios (Sec. 5.4.3).

A fundamental problem of existing middleware is that it is not a *µ-middleware*. It offers use-case-specific or workflow-specific support functionality. As discussed in Sec. 4.5, such functionality can prevent the implementation of pervasive computing scenarios that were not planned when the support functionality was added. Providing workflow-specific support has negative implications on the application design [DM12] (Sec. 2.5).

Only three of the analyzed middleware designs do not offer functionality that interferes with the implementation of services. But those middleware designs cannot be extended with core functionality which makes service development unnecessarily complex (Sec. 6.2).

Service management is implemented locally by several middleware designs. The term *Smart Space Orchestration* (Sec. 2.4.1) describes that functionality of pervasive computing scenarios is implemented in software services. For distributing the implementations of the scenarios, a distribution mechanism is needed.

Only OPEN (Sec. 4.5.12) provides a global distribution mechanism for services. OPEN only supports ECA rules as services. This limitation facilitates the service distribution. HomeOS does not provide a solution but discusses the distribution of more complex service programs.

Smart Space Orchestration in combination with the demand of ubiquitous computing – to be ubiquitous– requires a pervasive computing middleware to *scale*. The assessment shows that most solutions have either central components, or use a meta model that need compute intense processing. Both decreases the scalability.

Most assessed solutions provide few *security* mechanisms. Functionality for authentication of communicating components as basis for access controls is typically missing. The same applies to encryption that can protect from eavesdropping or theft via parties that get physical access to the communication or storage infrastructure.

Only PACE (Sec. 4.5.11), and ACE (Sec. 4.5.14) support the validation of context. Such validation can help to increase the dependability and security of services that use a middleware (Sec. 5.5.6).

User-based development is partly supported by all solutions that allow extension with services. Simple-to-use concepts that support developers with different skills, and tools that support the development are typically missing. HomeOS and OPEN provide most developer support of the assessed solutions concerning real world use. Large scale service deployment mechanisms are only discussed in these two works.

HomeOS (Sec. 4.5.5) uses the *.net* framework for development. This introduces a comfortable IDE that can be used to develop services for HomeOS.

OPEN (Sec. 4.5.12) is the only solution that addresses developers with different skills. For the highly limited use case of representing services as ECA rules, different levels of developers are supported by offering different interfaces.

4.5.18 Middleware Assessment Conclusion

The requirements from Sec. 3.12 are used as basis for the assessment of the state of the art pervasive computing middleware. They overlap with assessment criteria from different pervasive computing middleware surveys. The assessment shows the relevance of the criteria for the research community.

The presented survey exceeds existing surveys by including *service management* and *context management* middleware, and by assessing both with the same extended set of criteria. A joint analysis helps identifying shortcomings and the sum of supported features more clearly. Besides the shortcomings several interesting aspects are identified in the different middleware designs.

Analyzing the state of the art identifies the missing of a middleware that solves the overall objective of this thesis (<O.0>). Most existing solutions focus on connecting (selected) Smart Devices (<O.1>), and facilitating the creation of (certain) *orchestration services*. Diversity support (<O.3>), and real world deployment support (<O.4>) are typically missing. Distributed in the state of the art, Several promising solutions for parts of the requirements (Sec. 3.12) exist. A comprehensive solution is missing in 2014.

DS2OS offers such a comprehensive solution as shown in the remainder of this thesis. The most relevant parallels and differences between a related work and DS2OS are discussed in each middleware assessment as last paragraph.

4.6 Chapter Conclusion

With the right tools, crowdsourced software development for Smart Spaces is possible in 2014 (Sec. 4.2).

Virtualization of interfaces enables portability (Sec. 4.3). This is relevant for making the software of future Smart Spaces independent from specific hardware platforms. The virtualization concept is important for making future Smart Space services independent from the heterogeneous Smart Device interfaces (Sec. 3.3.1). Sec. 5.6.3 discusses portability in detail.

A descriptive representation of management information and access via a fixed API facilitate the management of complex systems (Sec. 4.4). Standardization is needed to provide interoperability (Sec. 3.3.2).

The assessment of the 13 middleware designs shows that most middleware provides partial solutions for the adaptation of Smart Devices (<O.1>), and for the creation of orchestration services (<O.2>). Diversity support (<O.3>), and real world deployment support (<O.4>) are typically missing.

Middleware silos are the fundamental problem of current pervasive computing middleware. They prevent the implementation of diverse scenarios. The missing of suitable standardization mechanisms for abstract interfaces in Smart Spaces is a problem as it inhibits portability.

Existing middleware solutions typically do not provide sufficient support fo user-based development. The offered development interfaces are too complex for non experts. This includes the creation of context models. Current solutions often do not scale. They lack sufficient security mechanisms, and do not provide mechanisms for distributing software to Smart Spaces.

The functionality of all assessed middleware can be combined as done in Fig. 4.5.16. The combined reference model shows the sum of functionalities that are distributed among the existing solutions. A middleware that combines all aspects was not found in literature. Even if it existed it would still have shortcomings as discussed in this section.

The remainder of this work shows how a single pervasive computing middleware framework can offer all functionality. To compare the differences to existing approaches, all elements of Fig. 4.5.16 are shown again in Fig. 7.6 in Sec. 7.7.2. The assessment criteria that were applied to the state of the art are applied to the solution that is presented in this thesis in Sec. 7.7.1. The entries of the last row of table 4.1 are discussed in that chapter.

Part II

Design & Implementation

5. The Virtual State Layer Programming Abstraction

A language that doesn't affect the way you think about programming, is not worth knowing.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

Short Summary The chapter introduces the Virtual State Layer (VSL) programming abstraction. It is based on the context-provisioning middleware concept.
A hybrid meta model is presented. It is used for creating context models that are shared over a global Context Model Repository (CMR) that becomes a self-managed collaborative ontology. A collaborative automated convergence mechanism for context models is introduced. The context models are used to offer context storage for services over a fixed Application Programming Interface (API) with 13 methods.
Via a novel concept, called Virtual Nodes, it becomes possible to use context models as abstract service interfaces. This enables a Service Oriented Architecture (SOA) of Smart Space services.
It is shown how the programming abstraction implements distribution transparency and autonomy, and how it fosters portability of services and security-by-design.

Key Results Most of the requirements from Sec. 3.12 are fulfilled by the programming abstraction.
A *hybrid meta model* is developed that combines the positive aspects of key-value pairs, markup schemes, object oriented models, and logic based models. It is used as basis for creating a *self-managing collaborative ontology* for Smart Spaces.
The novel concept of *Virtual Context* is introduced. It enables the use of *context models as abstract service interfaces*. This makes all concepts presented for context management usable for inter-service communication *reducing the complexity of the problem space*, and enabling new features such as *security-by-design*.

It is shown how the context-awareness support of the VSL allows to *separate service logic from service state* which facilitates the development of Smart Spaces services, and enhances their dependability.

Key Contributions The concept *Virtual Context* is introduced. It allows transparent direct coupling of services over a descriptive data structure. It is the key for dynamic transparent extension of core functionality of the programming abstraction, and for implementing a SOA of Smart Space services.
A *dynamically extensible, self-managing collaborative ontology* is introduced. It exceeds existing approaches for collaborative ontologies [HJ02, KA06, LST13] as it is self-managing meaning that it maintains its integrity, and it provides automated convergence mechanisms.
The dual use of context models for structuring context, and as abstract service interfaces (see below) makes the VSL ontology in the Context Model Repository (CMR) not only a self-converging ontology, but also a *collaborative standardization mechanism for abstract service interfaces*. This is a novel approach for implementing interface portability that is needed as basis for creating services that can run in different Smart Spaces.
A *hybrid meta model* that combines the desired properties from table 3.3 is introduced. Its major advantages are simplicity-to-use via supporting object oriented principles, high expressiveness, and fast processing via self-containment.
The use of *context for inter-service communication* is introduced. It allows using context models as abstract service interfaces. Such an approach was only proposed to a smaller scale by two existing solutions, Context Toolkit (Sec. 4.5.8), and One.World (Sec. 4.5.13).
Using *context models as abstract service interfaces* reduces the complexity as only one abstract data structure has to be maintained. At the same time it introduces the features of context modeling to the creation of abstract service interfaces. The hybrid VSL context model features include inheritance and validation. The use of an explicit descriptive data structure for coupling services structures and facilitates the development of services, and enable mash ups. It allows handling context sharing and access control outside of services lowering their complexity and increasing the security.
A concept for providing security-by-design is introduced. The use of context and Virtual Context enables coupling services using the VSL programming abstraction only. The resulting coupling over the middleware allows the enforcement of access control and encryption.
A strict *separation of service state and service logic* is proposed. Only One.World (Sec. 4.5.13) proposes such a separation [GDH01]. The concept that is described in this chapter exceeds the One.World concept. The separation structures and facilitates the development of services, and the coupling of services. It enables generative coupling between services (Sec. 3.4.1, services do not have to run for communication).

Challenges Addressed <R.1> *Simplicity-to-use*, <R.2> *Multi-user support*, <R.3> *Self-management*, <R.4> *User participation*, <R.5> *Context-awareness support*, <R.6> *Role-independent unified interfaces*, <R.10> *Standardization*, <R.9> *Information loss prevention*, <R.8> *Portability of services*, <R.11> *Distribution support*, <R.12> *Ac-*

cess transparency, <R.13> *Location transparency*, <R.14> *Migration transparency*, <R.15> *Relocation transparency*, <R.16> *Concurrency transparency*, <R.17> *Failure transparency*, <R.18> *Persistence transparency*, <R.19> *Direct communication support*, <R.20> *Mailbox communication support*, <R.21> *Publish-subscribe communication support*, <R.22> *Generative communication support*, <R.23> *Dynamic extensibility*, <R.24> *Descriptive knowledge representation*, <R.25> *Interaction on different levels of abstraction*, <R.26> *Modularization support*, <R.28> *High expressiveness context representation*. <R.29> *Simple-to-use context abstraction*, <R.30> *Dependability*, <R.31> *Loose coupling of services*, <R.32> *Service contracts*, <R.33> *Service autonomy*, <R.34> *Service abstraction*, <R.35> *Service reusability*, <R.36> *Service composability*, <R.37> *Service statelessness*, <R.38> *Service discoverability*, <R.39> *Abstract interface validation*, <R.40> *Suitable meta model*, <R.41> *Low complexity meta model*, <R.42> *Collaboration support context models*, <R.43> *Dynamic extensibility context models at run time*, <R.44> *Context validation*, <R.47> *Emulator support*, <R.48> *Crowdsourced development*, <R.49> *Security-by-design*, <R.50> *Scalability*

Summary The chapter starts showing the embedding of the VSL programming abstraction in the real world (Sec. 5.1.1), and a discussion of the key innovations of the VSL (Sec. 5.1.2).
Then the *hybrid meta model* of the VSL programming abstraction is introduced (Sec. 5.2). It combines features from key-value pairs, markup schemes, object oriented models, and logic based models. The VSL meta model knows three *basic data types*, and the concepts *subtyping* and *composition*. The *hybrid properties* enable *context validation*.
Types are used as *primary context* by the VSL. It implements a *locator-id split* that enables *service portability* (Sec. 5.2.13). Context instances can be identified based on their *type identifier* independent of their instance addresses (locator).
Next the API for *context access* is introduced (Sec. 5.3). It consists of 13 fixed functions, supports *transactions*, and implements *access controls*.
Then *Virtual Context* is introduced in Sec. 5.3.4 as a novel concept that combines the structuring properties of a *static* context model with the dynamics of service function implementations.
Next the VSL service interface is presented (Sec. 5.4). It uses context models as abstract service interfaces implementing a SOA (Sec. 5.4.4).
How the VSL programming abstraction supports *security-by-design* for context access and inter-service communication is presented in Sec. 5.5.
Next, the properties of the programming abstraction are analyzed (Sec. 5.6). They are *distribution transparency* (Sec. 5.6.1), *autonomy* (Sec. 5.6.2), *portability support* (Sec. 5.6.3), *collaborative convergence support* (Sec. 5.6.4), the structuring of service implementations via the *separation of logic and state* (Sec. 5.6.5), support for *functionality emulators* (Sec. 5.6.7), and support for *formal modeling* (Sec. 5.5.6).
Sec. 5.7 shows how the VSL uses its own Virtual Context mechanism to provide functionality. The implementation of the *distribution of context models* within a VSL overlay (Sec. 5.7.1), and the implementation of the *type search* (Sec. 5.7.2) are presented.
Virtual Context can be used to add diverse support functionality (see Fig. 4.3) to the VSL (Sec. 5.7.3). The presented example is a dynamic extension of the VSL context

with *primary contexts* which is a fundamental operation for a context-provisioning middleware. The example demonstrates the flexibility and power of the VSL programming abstraction.
Finally the practical use of the VSL programming abstraction is demonstrated at an example service (Sec. 5.8).

5.1 Introduction

This chapter contains the main contribution of this thesis, the Virtual State Layer (VSL) programming abstraction that comprises:

- A dynamically extensible *meta model* that is used to define *context models.*
- A dynamically extensible global *collaborative self-managing ontology.*
- Globally shared *abstract service interfaces.*
- The *separation of logic and state* in service implementations.
- A novel concept called *Virtual Context* that allows to use (descriptive) context models for directly coupling services.

A possible implementation of the programming abstraction is introduced with the Virtual State Layer (VSL) μ-middleware in Ch. 6. Additional functionality such as service management and real-world deployment support are introduced on top of the VSL μ-middleware using the VSL programming abstraction in Sec. 7.

In the remainder of this thesis, the term VSL is typically used to describe the conceptual layer that contains the knowledge base of a Smart Space instance.

5.1.1 Embedding of the VSL in the Real World

Sec. 3.1.2 structured the problem space pervasive computing into seven layers of abstraction and four functional rings in Fig. 3.1. Fig. 5.1 shows the embedding of the VSL programming abstraction in this topology.

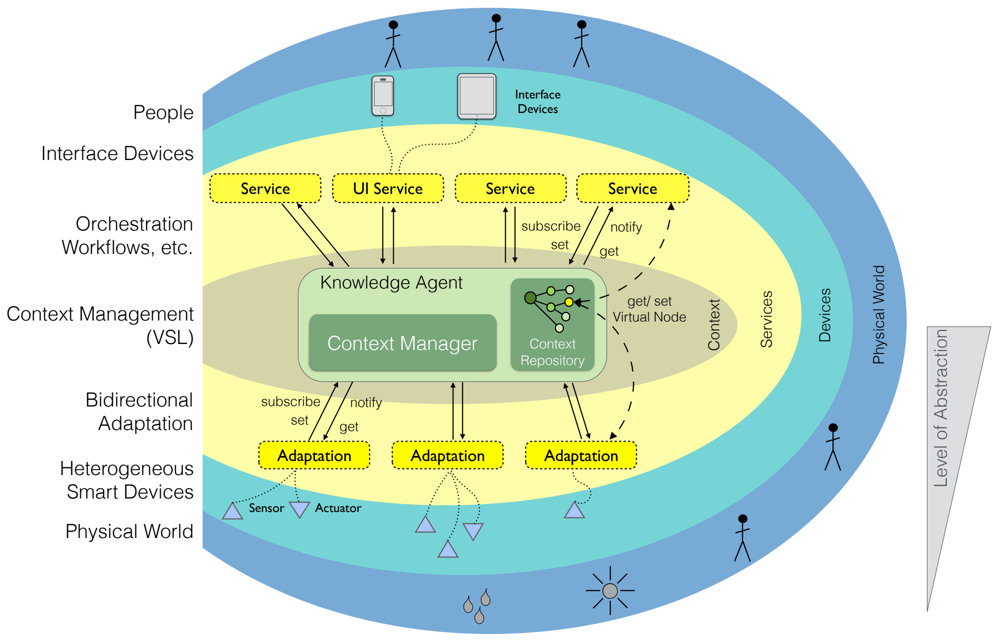


Figure 5.1: Context Management Architecture of the VSL with Logic Layers.

The functionality of the layers was described in Sec. 3.1.2 and is summarized on the left of Fig. 5.1. For basic understanding of the architecture, the VSL entities shown in Fig. 5.1 are briefly introduced.

To overcome the distribution and the heterogeneity of computing hardware in Smart Spaces (Sec. 3.2.4), the VSL programming abstraction is designed as middleware (Sec. 3.4).

Since the implementation of pervasive computing scenarios typically requires the processing of context, the VSL is a context-provisioning middleware. Its purpose is providing all kinds of services (Sec. 8.3) in the *service domain* (Fig. 5.1) with the context they need to reach their goals (Sec. 3.6.3).

The VSL is designed as fully distributed system of peers that are called Knowledge Agents (KAs) (Sec. 5.6.1, Fig. 5.4). A Knowledge Agent (KA) is shown in the center of Fig. 5.1. The function of a KA is managing context that belongs to services. Managing context includes providing context repositories for services, and retrieving context from other KA peers.

The arrows show the interaction between the VSL KAs and services. The VSL offers a fixed Application Programming Interface (API) to access context that allows to synchronously access context in the context repository via `get` and `set`, and to asynchronously get notified on context changes that were `subscribe`d before (Sec. 5.3.1).

A special functionality of the VSL is so-called *Virtual Context* that is shown on the right (*get/ set Virtual Node*). *Virtual Context* is described in detail in Sec. 5.4.

Abstraction

Fig. 5.1 can be read from a functional point of view and from a domain point of view as described in Sec. 3.1.2.

From a functional point of view the level of abstraction raises from the bottom to top layer in Fig. 5.1[19]. Smart Devices typically expect concrete orders such as "state[7]=1" while people are likely to give their software-orchestrated Smart Spaces instructions on a high degree of abstraction such as "I want to feel good".

In the domain perspective, the highest degree of abstraction is reached in the context repository of the VSL in the center of the figure as it has the greatest distance to the physical world in the outer layer.

Therefore the "Level of Abstraction" arrow ends in the middle of Fig. 5.1. In the shown length it represents the domain view. The reader may mentally elongate it to read the first interpretation.

5.1.2 Novelty of the Approach

The VSL programming abstraction introduces four new concepts:

1. A *hybrid meta model* that is simple, efficient to parse, and expressive.
2. A *collaborative self-managing ontology.*
3. The *dual use of context models*, first as concept to structure context representations, and second as abstract interfaces to services.

[19]As written in Sec. 3.1.2, the layered view becomes visible when removing the circle connections on the rights of the figure.

4. *Virtual Context* as concept that allows to use context models as abstract service interfaces.

None of the four concepts is implemented in any of the assessed middleware designs. As context management is a task that is not common to middleware in general but especially important for context-provisioning middlewares [KKR+13], it is likely that the four mechanisms and their described combination are novel for middleware in general.

The *hybrid meta model* combines properties from key-value-pairs, markup, object orientation, and ontology to implement a simple-to-use and highly expressive context model. VSL context models can be extended with semantics at run time.

A hybrid meta model and the semantic extensibility at run time are typically not provided by state of the art context modeling techniques [BBH+10]. Probably both aspects are novel as they are implemented by combining static and dynamic elements (instances & instantiation, context models & Virtual Context) which is not a typical proceeding in the context modeling community.

Providing state management (context management) as basic functionality of the VSL programming abstraction enables a full separation of service logic from service state. State can be stored in the VSL instead of using data structures within service implementations. This work shows advantages of such a separation. The structure of service state in the VSL context repositories is defined via context models. Using the context repositories for storing service state enforces the creation of context models as structure of the state space that can be used by services.

The VSL programming abstraction requires VSL context models to be shared over a global Context Model Repository (CMR). The presented mechanisms manage the integrity and the distribution of context models. A collaborative standardization mechanism is introduced. It is extended in Sec. 7.4.3. The combination of the presented mechanisms implements a self-managed ontology. As the CMR is open for submissions of new context models, it implements a collaborative ontology.

The collaborative creation of ontologies is an open resource topic in 2014 [LST13, KA06, HJ02]. Similar approaches to the presented self-managing ontologies could not be found in literature.

The VSL introduces a *dual use of context models* as unified interface to static context, and to dynamic services. Using one interface for both reduces the complexity of context access and inter-service communication to the invocation of 13 fixed methods. In addition to the simplification for developers, the unified interface result in inherent service composability. With the 13 methods any two VSL services can be coupled.

Via the described dual use, the definition of context models is identical to the definition of abstract service interfaces. VSL context models are shared over the global CMR. Thereby the abstract interfaces of services are shared resulting in a service interface directory. In combination with the fixed methods to access service functionality this fosters reuse. Though the use and sharing of abstract service interfaces is known from the Service Oriented Architectures (SOAs) paradigm, the combination with context models and the implemented enforcement of a global sharing of abstract service interfaces are novel.

The term *Virtual Context* describes context that is created on-demand in a transparent way for services accessing it. From an API perspective, Virtual Context can be compared to dynamic websites that get generated dynamically by programs based on Representational State Transfer (REST) requests. Applying such a technology to application context is novel. It enables the use of context models as abstract service interfaces for direct and indirect coupling of services. This lowers the complexity of a solution for Smart Space Orchestration significantly as it consolidates the requirements on the development of context models and abstract service interfaces.

The power of the four novel concepts is shown in Sec. 7, and Sec. 8. The usability study (Sec. 9.6) supports the assumption that the presented design implements a simple-to-use and powerful programming abstraction (<O.0>).

5.2 The Hybrid VSL Meta Model

The longer you look at an object, the more abstract it becomes, and, ironically, the more real.

Lucian Freud, British painter, (1922 - 2011).

The dynamically extensible VSL meta model is the core of the VSL. Fig. 3.12 illustrates the correlation between meta model, context model, and virtual object. In the remainder of this document, the term context will be used instead of virtual object as it is shorter, and as the presented design is situated in the virtual world where context is always represented as virtual object.

Table 3.3 summarizes strengths and weaknesses of different meta models as result of their analysis in Sec. 3.9.

Major advantages of the different approaches are:

- *Key-value pairs* are simple to use, to understand, and to process.
- *Markup schemes* provide structure, add types, metadata, and the possibility for validation.
- *Object oriented models* support inheritance which facilitates the creation of context models (Sec. 3.9), and they automatically represent dependencies via inheritance relationships.
- *Logic based models* allow to express more relations than the previous, they allow to validate semantics, and they allow the inference of new context via applying logic rules.

From the top to the bottom of the list the expressiveness increases which is desired, and the complexity increases which is not desired.

Expressiveness means here that diverse relationships of a context information such as functional or spatial relationships can be expressed explicitly when creating context models using the respective meta modeling approach.

Complexity affects the usability (<O.4>) and the computational complexity that is needed to process context models that follow a specific meta modeling approach.

Expressiveness and complexity have been assessed for existing meta models in Sec. 3.9, and for existing pervasive computing middleware in Sec. 4.5. Detailed differences between the meta models can be seen in table 3.3.

Following, the VSL meta model is introduced. Its design goal is combining the positive aspects of the different meta modeling approachs that were presented in Sec. 3.9 while:

- keeping the complexity so low that user-based development becomes possible,
- enabling *collaborative creation* of context models,
- supporting *timeliness* for making historic contexts accessible,
- introducing *dynamic extensibility* not only for supporting the adding of new context models at run time but also by supporting to add new semantics,
- providing *high expressiveness* for supporting a reasoning process that infers new context from existing context,
- allowing *distributed processing*,
- providing a *syntax that can be validated*,
- supporting *semantic validation*, and
- enabling *efficient parsing.*

For a more detailed discussion of the properties and an assessment of existing meta modeling approachs see Sec. 3.9.

The methodology that is applied in the following is the creation of a hybrid meta model by combining existing meta modeling techniques.

5.2.1 Tuples

The VSL meta model consists of tuples that comprise a value and the following meta-data (see Fig. 5.2):

- *Type information* that defines the type of the tuple value and its function.
- *Reader Identifiers* that are used for read access control.
- *Writer Identifiers* that are used for write access control.
- *Subscriber Identifiers* that are used for handling notifications.
- A version number that allows to access historic values.
- A time stamp that gives information when the tuple value was set.

The tuple can be represented as $\langle address, typeIDs, readerIDs, writerIDs,$ $subscriberIDs, versionNr, timestamp\rangle$.

5.2.2 Hierarchical Addressing

The VSL uses a hierarchical addressing scheme to identify the tuples in the context repositories (Sec. 5.2.1). The addressing scheme is comparable to typical file system addressing:

```
/grandParent/parent/child
```

The VSL addressing puts the tuples into a *hierarchical structure*. Humans are familiar with the concept of hierarchical structuring context (Sec. 3.5). This familiarity facilitates the use of the proposed concepts (<R.29>). *<R.29>*

A logical tree structure of VSL tuples emerges. The tree structure allows to express *containment*. An address `smartDevice/lightSensor/irradiance` expresses that a Smart Device contains a sensor that provides a lightValue. See Fig. 5.2.

The *hierarchical addressing* implements *composition*. Context gets aggregated under a common prefix/ namespace (Sec. 5.2.3)/ subtree (Sec. 5.2.7). Hierarchical addressing gives *composition* semantics to the VSL context models (Sec. 5.2.18).

The described aggregation has the intended side-effect that it makes *different levels of abstraction* accessible from the most general on the left (`smartDevice`) to more specific entities being identified later in the address with the last entry being the most specific (`irradiance`), e.g. `/smartSpace/smartDevice/module/sensor` (<R.25>). *<R.25>*

5.2.3 Namespaces

The hierarchical addressing (Sec. 5.2.2) results in a logical tree structure of context nodes. This automatically structures the *namespace*.

The addressing scheme is applied for context nodes in context repositories and context model, and for the identification of the context model in the shared CMR.

If two context node subtrees have identical subnode addresses but the parent nodes' addresses are different, each node in a subtree can be identified by a unique address. Each subtree implements its own *namespace*.

Using the example context model from listing 5.1, an instance of the model with the address `/device7/lamp23` can be distinguished from another instance of the same model at address `/device42/lamp23` as both are in different *namespaces* (`device7`, `device42`).

VSL context nodes are always addressed via the Identifier (ID) of the KA they are stored at followed by their relative address in this KA's context repository. Both are site-locally unique. Each VSL Smart Space has a unique ID (Sec. 5.5). The hierarchical combination of the three elements leads to a globally unique address of each context node that is stored in the VSL. An example is `/mySmartSpaceID/myAgentID/device42/lamp23`.

On the CMR (Sec. 5.2.8), the address scheme separates the address space into disjoint sub branches. This organizes the context models in the repository and simplifies finding (<R.1>) them which is relevant for reuse (Sec. 5.4.1). *<R.1>*

<R.29> The hierarchical addressing scheme helps *structuring* context nodes (<R.29>), and context models in the CMR.

5.2.4 Key-Value Pair Properties

The VSL addressing introduces only a logical hierarchy. The underlying data does not necessarily have a hierarchical structure with dependencies such as tree structure.

Using a hierarchical data structure would require special access (e.g. tree traversal) introducing computational complexity. Using only a logical hierarchy results in fast processing all context data ($\langle address, value \rangle$) is self-contained. In contrast to an ontology (Sec. 3.9.1), no dependencies have to be resolved for accessing context. This makes the processing of context fast (<R.50>). *<R.50>*

Instances of VSL context nodes can be accessed like key-value pairs. The key is the address and the returned value is the value of the tuple. This results in *low complexity* for using context in services (<R.29>). *<R.29>*

5.2.5 Markup Scheme Properties

The VSL context node trees are represented via markup. In the current implementation of the VSL (Ch. 6), Extensible Markup Language (XML) is used. Therefore the following examples are in XML for illustration of the usability.

All other markup languages could be used if they support *naming entities*, and *associating properties*. The additional properties represent tuple metadata (Sec. 5.2.1) and a fixed set of restrictions that are be described in Sec. 5.2.10.

Listing 5.1 shows a context model for an entity that has two child nodes. The three dots identify that the type is usually part of a longer *namespace* (Sec. 5.2.3) that is not printed as it is irrelevant for the example.

```xml
<deviceA>
  <lamp23 type=".../lamp"></lamp23>
  <lamp42 type=".../lamp"></lamp42>
</deviceA>
```

Listing 5.1: XML representation of the VSL context model of a `deviceA`

Listing 5.1 shows that the XML tags are used for *naming* (see listing 5.1) and not for *typing*. If they would be used for *typing*, the markup would be:

```xml
<lamp name="lamp23"></lamp>
```

Using tags for naming and not for typing is done for two reasons. First it is closer to the VSL addressing scheme (Sec. 5.2.2) which is expected to *facilitate the understanding* of the meta model (<R.29>, <R.41>). Second, the VSL uses multi-inheritance (Sec. 5.2.11) which results in multiple data types per node. Using multiple types as tag name is impractical. *<R.29>* *<R.41>*

The use of a markup scheme allows to represent *containment* (Sec. 5.2.2), and *typing* (Sec. 5.2.9) with low complexity. As the attributes (e.g. `type`) of VSL nodes are fixed (Sec. 5.2.1, Sec. 5.2.10) *syntax validation* (<R.44>) is enabled (Sec. 5.2.14). This reduces the risk of errors which enhances the *dependability* (<R.30>) and the *security* (<R.49>) that the VSL meta model provides. *<R.44>* *<R.30>* *<R.49>*

5.2.6 Context Model

Context models are templates for representing properties of the physical world in the virtual world (Sec. 3.9). See Fig. 3.1 and Fig. 3.12.

The VSL uses trees of VSL nodes as context model. See Fig. 5.2. Represented in markup, listing 5.1 could be a VSL context model for instance.

<R.24> *<R.29>* *<R.4>* *<R.48>*

Trees as descriptive concept to describe properties are a simple concept that is expected to be simple-to-understand by humans (<R.24>, <R.29>). This *increases the usability* of the VSL context abstraction enabling *user participation* (<R.4>), and *crowdsourcing* (<R.48>) of the creation of context models by supporting different levels of expertise at the user side including non-experts.

5.2.7 Knowledge, Trees, and Repositories

The terms *knowledge*, *knowledge tree*, *knowledge base*, and *Knowledge Object Register* are used in the remainder of this work. Therefore their intended meaning is introduced.

Definition ***Knowledge***

The term *knowledge* is used to describe instances of context nodes in this work.

Definition ***Knowledge Tree***

All instances of context models that are stored in a VSL context repository are called *knowledge tree* in this work.

Definition ***Knowledge Base***

The term knowledge base can be found in literature. It describes the conjunction of all knowledge trees of a VSL instance.

Definition ***Knowledge (Object) Repository***

The VSL context repositories that contain the instances of VSL context nodes are also-called *Knowledge Object Repository (KOR)* or shorter *knowledge repository* in this thesis.

5.2.8 Context Model Repository

The VSL uses a Context Model Repository (CMR) to *share context models* (see Fig. 5.2). All VSL context models that should be usable in multiple Smart Spaces must be uploaded to the global CMR. All context models that are instantiated in a local context repository are loaded from the CMR (Fig. A.1).

<R.43>

The CMR is *open to the public* for read and write. Sec. 5.7.1 describes the CMR implementation in detail. Everyone can submit context models that can directly be instantiated in all VSL Smart Spaces (Sec. 5.7.1, <R.43>). The CMR ensures that each submitted context model has a unique name that is called *ModelID*.

The ModelIDs that identify the context models use the VSL addressing schemes for naming (Sec. 5.2.2). The hierarchical naming introduces namespaces (Sec. 5.2.3). The use of *namespaces* structures the CMR which facilitates *reuse*, *collaboration* (<R.42>, <R.48>), and *standardization* (<R.10>). *Standardization* is facilitated as the use of namespaces helps finding existing context models and reusing them (e.g. `lamp`). *<R.42>* *<R.48>*

Context models are *automatically versioned* by enforcing the use of unique names. Each version of a context model must have a new identifier. A numeric suffix can be added to the model identifier for instance: `/light/lamp`, `/light/lamp2`,`/light/lamp3`, etc. *<R.10>*

In combination with the service management that is presented as part of the Distributed Smart Space Orchestration System (DS2OS) framework in Sec. 7.2, a *crowdsourced standardization* (<R.10>) of context models is implemented via crowdsourced convergence (Sec. 7.4.3). *<R.10>*

Having a central global repository ensures that the VSL instances of all Smart Spaces use identical context models. The availability of context models in the global CMR enables developers to identify and reuse context models (Sec. 5.2.18).

5.2.9 Dual Role of Types

Types have a dual use in the VSL.

- Identification of the *data type* of the value of a context node (e.g. `/basic/number`, Sec. 5.2.10).
- Identification of the *functionality* of a context node (naming) by pointing to the ModelID of a context model in the CMR that is associated with a certain functionality (e.g. `lamp`).

The *data type* adds semantics that can be automatically processed by the computer. It allows the validation of values via the VSL, e.g. by automatically validating if a value is within the given restrictions of a context node's data type (Sec. 5.2.14). The *functionality identifier* adds semantics for human developers.

Sec. 5.2.18 introduces the functionality used for typing in the VSL.

5.2.10 Basic Data Types

The VSL meta model knows three basic data types: *text*, *number*, and *list*. The three data types have each different fixed restrictions that are used by the VSL to *automatically validate* (<R.44>, <R.49>) the value of a tuple (Sec. 5.2.14). See table 5.1 for the restrictions. *<R.44>* *<R.49>*

The basic data type *text* represents any text and has a *regular expression* as restriction for its value. The XML representation of an instance of the VSL basic data type *text* with the default restrictions is as follows:

```
<myText type="/basic/text" regularExpression="">
  string value
</text>
```

Data type	restrictions	description
text	regular expression	The data type text can be used to represent texts.
number	upper bound lower bound	The data type number can be used to represent integers.
list	minimum entries maximum entries allowed ModelIDs	The data type list can be used to represent lists of elements. It is the only type that allows to extend a context model with additional context nodes within the *restrictions* of the `list`.

Table 5.1: The three basic data types of the VSL meta model.

The basic data type *number* represents integers and has a *lower bound* and an *upper bound* as restrictions for its value. The XML representation of an instance of the VSL basic data type *number* with the default restrictions is as follows:

```
<myNumber type="/basic/number" lowerBound="" upperBound="">
  23
</myNumber>
```

A number contains an integer value out of ℤ in the VSL. Other number types can be composed from `/basic/number` (Sec. 5.2.11), e.g.

```
<fraction>
  <numerator type="/basic/number"></numerator>
  <denominator type="/basic/number"></denominator>
</fraction>
```

The basic data type *list* represents lists. While with the other basic data types only the values change at run time, the *list* enables changes on the context structure at run time. A service can add and remove list entries within the limits of the restrictions. A *list* has an *allowed types* restriction that contains a comma separated list of ModelIDs, a *minimumEntries* restriction that contains a positive integer that restricts the minimum amount of entries, and a *maximumEntries* restriction that contains a positive integer that restricts the maximum amount of list entries.

At instantiation time at least *minimumEntries* entries of the *list* have to be given, or it cannot be initialized. Entities can only be deleted from a *list* when at least *minimumEntries* entries remain. Entities can only be added to a *list* when it does not contain *maximumEntries* entries yet. *Lists* are controlled via Virtual Context nodes for accessing and deleting entries (Sec. 5.3.2).

The XML representation of an instance of the VSL basic data type *list* with the default restrictions and two list nodes is shown in listing 5.2:

```
<myList type="/basic/list" allowedTypes=""
                           minimumEntries="0"
                           maximumEntries="">
  <elements>
    element1/element2
    <element1 type="/basic/text">
```

```
      foo
    </element1>
    <element2 type="/basic/number">
      42
    </element2>
  </elements>
</myList>
```

Listing 5.2: An instance of the basic data type *list*

As described in Sec. 5.4, context models are instantiated only at the start of services for different reasons. For context nodes of type *text* or *number* only the value can be changed at run time. The *list* is a generative type that can be used to dynamically extend a context model according to the *list* restrictions at run time. To store context that changes dynamically (e.g. a list of missed calls), nodes in the context model can be typed as *list*. See Sec. 5.3.2.

5.2.11 Object Oriented Properties

Object oriented software design is close to the functionality that can be observed of the human mind (Sec. 3.5). As shown next, besides the good understandability for humans, object orientation helps *reducing the complexity* of context modeling (<R.41>, <R.29>). *<R.41>*

<R.29> The VSL supports inheritance. VSL inheritance means that context models can cause the instantiation of other context models by using the other context model's *ModelID* as type identifier in their definition.

As object oriented concepts, the VSL meta model supports *inheritance* for *subtyping* and *composition* during the modeling and the instantiation of models.

As described before (Sec. 5.2.4), the VSL context repositories do not store objects but only tuples as this is more efficient for processing and distributed processing (Sec. 4.5).

Subtyping

VSL context models can *derive* basic data types and existing context models in the CMR.

Subtyping is implemented by narrowing the restriction of an existing basic data type (Sec. 5.2.10) or a derived type that has a context model in the CMR. The following `boolean` is defined as subtype of `/basic/number`:

```
<boolean type="/basic/number" lowerBound="0" upperBound="1">
  0
</boolean>
```

The listing shows that context models can have values.

As described before (Sec. 5.2.9), types are used to give semantics to the value of a tuple for machines and humans. A typical operation for adding semantics for humans is *naming*. It happens by giving an existing type a new name:

```
<isOn type="/derived/boolean">0</isOn>
```

The resulting derived type `isOn` could be used as subnode of a "`lamp`" context model for instance (listing 5.1, Fig. 5.2):

```
<lamp>
  <isOn type="../isOn">0</isOn>
</lamp>
```

The relevance of *naming* becomes more clear in the context of the locator-id split that is implemented by the VSL (Sec. 5.2.13).

Composition

New context models can be *composed* from existing context models in the CMR. Listing 5.1 and Fig. 5.2 show an example where the new type with the ModelID `deviceA` contains a node of type `lamp`. VSL context models are typically *composed* of multiple nodes of other types.

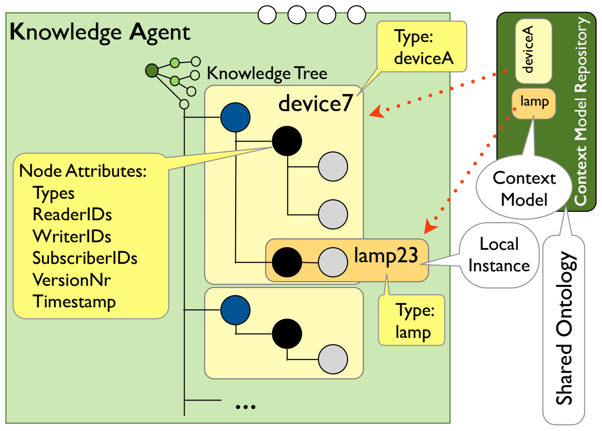


Figure 5.2: The VSL meta model with the global CMR and a local KA that contains a knowledge tree. The arrows show the instantiation of a context model with the ModelID `deviceA` that contains a node of a context model with the ModelID `lamp23` that automatically gets instantiated with the instantiation of `deviceA`.

On instantiation of a context model from the CMR to a VSL instance the name of the parent node of the context model is changed to an instance name in the local knowledge repository (e.g. `device7` in Fig. 5.2).

The VSL ensures that the instance name of the root node (`device7`) of a context model that gets instantiated is unique. As each knowledge repository has a unique identifier and as each VSL operated Smart Space has a unique identifier, the full address of each VSL context node is world wide unique. See Sec. 5.2.2.

Keeping the names of the subnodes that are contained inside a context model unchanged is necessary to enable services to use relative addressing inside an instance of a context model once they have identified the root node of the context model instance. See Sec. 5.2.13. Having identified the root node `myRoot` of an instance of the type `deviceA` for instance, it is clear from the context model that is shown in listing 5.1 that the lamps can be reached under the addresses `myRoot/lamp23`, and `myRoot/lamp42`.

Multi-Inheritance

As types identify functionality in the VSL context models, *multi-inheritance* is necessary for representing entities that provide different functionality. *Multi-inheritance* describes that a context node in a context model can inherit from multiple exiting types (context models) that are identified by their ModelIDs.

A context model to a Smart Device that contains a light that can be dimmed, and that can blink could be defined as:

```
<fancyLamp type="../dimmableLight,.../blinkingLight,.../lamp">
</fancyLamp>
```

The VSL inherits all context models that are associated with the ModelIDs starting with the context model identified by the rightmost modelID. Values and restrictions are overridden when new values are given in the derivation chain. In listing 5.2.11, ". . . /lamp" is initialized first, overridden by ". . . /blinkingLight" properties, again overridden by ". . . /dimmableLight" properties. In addition possible additional nodes that are present in the later initialized context models are added in the model instance.

The example from the listing above is already the entire context model definition of the model `fancyLamp` as all properties are inherited. This gives an impression how inheritance simplifies the creation of context models (<R.29>, <R.4>). *<R.29>*

The given multiple inherited ModelIDs make it possible to identify all different functionality that is associated with the different ModelIDs. This property is called *subtype polymorphism*. For a service that accesses the `fancyLamp` as a `.../lamp` it is transparent that the `fancyLamp` has more context to offer. See Sec. 5.2.13. *<R.4>*

Functional Diversity

multi-inheritance allows services to access the `fancyLamp` as `.../dimmableLight`, `.../blinkingLight`, `.../lamp`, and `.../fancyLamp`. Different functionality provides different levels of abstraction, e.g. a `lamp` can be switched on and off, a `dimmableLamp` adds a finer granularity by additionally allowing dimming. This allows to implement transparent access to context on different levels of abstraction (<R.25>), e.g. *<R.25>*

```
type="veryAbstract,moreConcrete,rawData"
```

Via *multi-inheritance*, compatibility of Smart Devices can be reached. The adaptation service that gives access to the *fancy lamp* via the context model `fancyLamp` makes the device access possible for all services that can interact with functionality of at least one of the inherited types.

This implements *portability* (<R.8>) as a service that operates lamps can now run in an environment that only contains *fancy lamps*. Via the *multi-inheritance* the difference is transparent to the service. *<R.8>*

Data Type Diversity

The VSL does not use *interrelated objects* for representing context but *self-contained tuples*. It implements object oriented functionality during the instantiation of context models. All inheritance dependencies are resolved at instantiation time, resulting in self-contained tuples as context model instances.

Using self-contained context model instances that have all dependencies resolved at instantiation time results in *fast processing* of VSL context models (<R.50>) as no *<R.50>*

dependencies have to be resolved when accessing data (Sec. 5.2.14). Self-containment is implemented by making the inheritance chain of each context node explicit during instantiation.

Allowing *multiple types* for context nodes is needed to identify the *inheritance chain* of a type back to a basic data type (see *subtyping*). The definition of the `boolean` type above shows the explicit derivation chain that goes back to the basic data type `/basic/number` (`type="/basic/number"`). After instantiation the type field contains `"/derived/boolean,/basic/number"`.

The `isOn` type introduces another derivation step:

```
<isOnInstance type=".../isOn,/derived/boolean,/basic/number"
              lowerBound="0" upperBound="1">1</isOnInstance>
```

All VSL context nodes that can carry values must have a basic data type *text* or *number* at the end of their *inheritance chain* (Sec. 5.2.10). Inner nodes of a context model do not necessarily carry values as they can represent *aggregation* only.

Rooting all data types for values in only two basic data types enables services that can interpret the two basic data types to interpret the basic semantics of all VSL context nodes. A service that can interpret `/basic/number` can interact with the *isOn* type above as a *number.* A service that can interpret `/derived/boolean` can interact with the *isOn* type above as a *boolean.* A service that can interpret `isOn` can interact with the *isOn* type above as a value describing the on-state of a device.

From a data type-perspective, the most abstract data type is given on the left of the inheritance chain. To the right the types in the chain get more specific ending at a basic data type (Sec. 5.2.10).

From a functional point of view it is the other way round with the richest semantic type on the left, and the most generic semantic type (no specific semantics) on the right with the basic data type. Therefore services that want to interpret the semantics of a VSL context node should parse the derivation chain from left to right until they find the first data-type they understand.

<R.25> The explanation shows how the use of the explicit *inheritance chain* enables the transparent interaction with VSL context nodes on different levels of abstraction (<R.25>).

As shown in Sec. 8.3.5, the described properties of the inheritance chain mechanism allow to implement generic services that can display all values in a space.

Rooting each VSL context node that can have a value to a basic data type is also relevant for validating context values before storing them in the VSL as described in Sec. 5.2.14.

Inheritance Example

To illustrate how inheritance structures and facilitates the creation of context models, some examples from the perspective of a developer that uses the VSL are given.

Following a context model "`ipv4Address`" is shown:

```
<ipv4Address>
  <a type="/basic/number" lowerBound="0" upperBound="255">0</a>
  <b type="/basic/number" lowerBound="0" upperBound="255">0</b>
  <c type="/basic/number" lowerBound="0" upperBound="255">0</c>
  <d type="/basic/number" lowerBound="0" upperBound="255">0</d>
</ipv4Address>
```

This model could now be reused in another context model, e.g. as `ipv4Address` of the network interface of a Personal Computer (PC):

```
<networkInterface>
  <myIpv4 type=".../ipv4Address"></myIpv4>
  ...
</networkInterface>
```

Via inheritance, context models can assign values to the properties of the inherited context model as shown in the following example where the before defined type "`networkInterface`" is inherited:

```
<myNIC type=".../networkInterface">
  <myIpv4>
    <a>10</a>
    <b>23</b>
    <c>42</c>
    <d>65</d>
  </myIpv4>
  ...
</myNIC>
```

An instance of the context model "`myNIC`" in a knowledge repository under the name *myNIC* would contain the following data:

```
<myNIC type=".../networkInterface">
  <myIpv4 type=".../ipv4Address">
    <a type="/basic/number" lowerBound="0" upperBound="255">10</a>
    <b type="/basic/number" lowerBound="0" upperBound="255">23</b>
    <c type="/basic/number" lowerBound="0" upperBound="255">42</c>
    <d type="/basic/number" lowerBound="0" upperBound="255">65</d>
  </myIpv4>
  ...
</myNIC>
```

The following listing shows a real world example for a Smart Device that was built during the user study (Sec. 9.6) by participants:

```
<smartDevice>
    <temperature type="/ilab/temperature"></temperature>
    <button type="/ilab/button"></button>
    <ledRed type="/ilab/led"></ledRed>
    <ledYellow type="/ilab/led"></ledYellow>
    <ledGreen type="/ilab/led"></ledGreen>
    <ledOnBoard type="/ilab/led"></ledOnBoard>
    <lightSensor type="/ilab/lightSensor">
    </lightSensor>
    <timer0 type="/ilab/triggerTimer"></timer0>
    <timer1 type="/ilab/triggerTimer"></timer1>
    <vTimer0 type="/ilab/vTimer"></vTimer0>
    <vTimer1 type="/ilab/vTimer"></vTimer1>
</smartDevice>
```

Though no explanation of the functionality of the Smart Device is given, the example shows how self-descriptive VSL context models can be.

In addition, the three last listings show how *inheritance reduces the complexity* of VSL context models (<R.29>) as the complexity is added automatically by the VSL *<R.29>*

on context model instantiation via resolving the inherited types. All composed and subtyped context models are loaded from the CMR and initialized with the given values from right to left.

This is the instantiated version of the "`smartDevice`" context model shown above:

```
<mySmartDevice type="/ilab/smartDevice" version="0" timeStamp
    ="2014-02-10 19:16:20.223" access="rw">
<mySmartDevice type="/ilab/smartDevice" version="0" timeStamp
    ="2014-02-10 19:16:20.223" access="rw">
<button type="/ilab/button,/basic/number" version="0" timeStamp
    ="2014-02-10 19:16:20.277" access="rw">
</button>
<ledGreen type="/ilab/led,/ilab/isOn,/derived/boolean,/basic/number"
     version="0" timeStamp="2014-02-10 19:16:20.439" access="rw">
0
<desired type="/ilab/isOn,/derived/boolean,/basic/number" version
    ="0" timeStamp="2014-02-10 19:16:20.472" access="rw">
0
</desired>
</ledGreen>
<ledOnBoard type="/ilab/led,/ilab/isOn,/derived/boolean,/basic/
    number" version="0" timeStamp="2014-02-10 19:16:20.5" access="rw
    ">
0
<desired type="/ilab/isOn,/derived/boolean,/basic/number" version
    ="0" timeStamp="2014-02-10 19:16:20.532" access="rw">
0
</desired>
</ledOnBoard>
<ledRed type="/ilab/led,/ilab/isOn,/derived/boolean,/basic/number"
    version="0" timeStamp="2014-02-10 19:16:20.3" access="rw">
0
<desired type="/ilab/isOn,/derived/boolean,/basic/number" version
    ="0" timeStamp="2014-02-10 19:16:20.337" access="rw">
0
</desired>
</ledRed>
<ledYellow type="/ilab/led,/ilab/isOn,/derived/boolean,/basic/number
    " version="0" timeStamp="2014-02-10 19:16:20.366" access="rw">
0
<desired type="/ilab/isOn,/derived/boolean,/basic/number" version
    ="0" timeStamp="2014-02-10 19:16:20.402" access="rw">
0
</desired>
</ledYellow>
<lightSensor type="/ilab/lightSensor,/basic/number" version="0"
    timeStamp="2014-02-10 19:16:20.559" access="rw">
</lightSensor>
<temperature type="/ilab/temperature,/basic/number" version="0"
    timeStamp="2014-02-10 19:16:20.238" access="rw">
</temperature>
<timer0 type="/ilab/triggerTimer,/ilab/timer,/basic/number" version
    ="0" timeStamp="2014-02-10 19:16:20.577" access="rw">
<desired type="/ilab/timer,/basic/number" version="0" timeStamp
    ="2014-02-10 19:16:20.599" access="rw">
</desired>
</timer0>
<timer1 type="/ilab/triggerTimer,/ilab/timer,/basic/number" version
    ="0" timeStamp="2014-02-10 19:16:20.647" access="rw">
<desired type="/ilab/timer,/basic/number" version="0" timeStamp
    ="2014-02-10 19:16:20.677" access="rw">
```

```
</desired>
</timer1>
<vTimer0 type="/ilab/vTimer,/basic/number" version="0" timeStamp
    ="2014-02-10 19:16:20.71" access="rw">
</vTimer0>
<vTimer1 type="/ilab/vTimer,/basic/number" version="0" timeStamp
    ="2014-02-10 19:16:20.736" access="rw">
</vTimer1>
</mySmartDevice>
```

5.2.12 Types as Primary Context

The term *primary context* describes the context that is used as key to select context [BBH+10] (Sec. 5.2.4). The primary context of the VSL are the *types* that are represented by the ModelIDs in the CMR. Therefore, the VSL meta model supports types as explicit metadata.

The VSL offers functionality to search for instance addresses based on the ModelID of a context model. See Sec. 5.2.13 and Sec. 5.7.2.

Via the extensibility of the VSL meta model, additional primary contexts such as *location* can be added at run time. See Sec. 5.7.3, Sec. 6.6.4.

5.2.13 Locator-Id Split

The name of a resource indicates what we seek, an address indicates where it is, and a route tells us how to get there.

John F. Shoch (1950-) [Sho78], American computer scientist and venture capitalist, contributor to PARC Universal Protocol (PUP), a predecessor of Transmission Control Protocol (TCP)/ Internet Protocol (IP).

The use of VSL types as *primary context* (Sec. 5.2.12) results in a *locator-id split* [JST+09, BFK+11]. The *locator* of a VSL context node is its VSL address. It is globally unique as described in Sec. 5.2.2. A VSL context node is identified by each of its *types*.

Locator-id split means that locating instances of context models in the knowledge base of a VSL Smart Space happens independent of the site-specific instance identifiers.

As seen in the examples above and illustrated in Fig. 5.2, instances of context models in context repositories can have any address as instance identifier. The possibility to search for types *enables portability* (<R.8>). It allows to develop services that identify service functionality, offered by context nodes, based on the ModelIDs of the context models (identifier, ID) independent of concrete Smart Space node instance addresses (locator). *<R.8>*

The lower red dotted arrow in Fig. 5.2 shows the instantiation of a context model `deviceA` in a local VSL context repository. While the instance gets an instance specific name (`device7`), its type remains `deviceA`.

Services that want to access context of type `deviceA` can retrieve the instance address `device7` via a VSL search for the type `deviceA` (Sec. 5.2.12, Sec. 5.7.2). This splits the *locator* `device7` from the identifier `deviceA`, making it possible to identify context nodes independent of their instance names (locator).

The use of the CMR for sharing context models (Sec. 5.2.8), and the dual use of types, for identifying *functionality* and *data type* of a context node (Sec. 5.2.9), enable service portability. They allow service developers to identify functionality they want to use for implementing pervasive computing scenarios independently of concrete devices or Smart Space instances.

5.2.14 Context Validation

The VSL meta model is designed to foster validation of syntax and semantics.

As described in Sec. 5.2.5 and Sec. 3.9, syntax validation is typically enabled by the use of a *markup scheme* for defining context models. The use of XML in the prototype allows such *validation* (<R.44>). *<R.44>*

Context Model Validation

Via the described properties of the VSL meta model with multi-inheritance, and the rooting of all data types that contain values in two basic data types (*text*, *number*), all context models that are submitted to the CMR can be formally validated for being well defined applying the methods described next.

<R.49> To *ensure the integrity of all context models* (<R.49>) that are stored in the CMR, only well defined context models that derive and compose existing context models correctly are stored in the CMR and can be used in local VSL instances.

The described mechanisms ensure the integrity of all context models in the CMR. Automatic context model validation is the basis for opening the CMR for dynamic extension with new context models (Sec. 5.2.18).

Derived Data Type Validation

The validation of a VSL type requires the execution of code that does the validation. Such code is called *validator*. The KAs that implement the VSL contain *basic data type validators* for the three basic data types (*text*, *number*, *list*). They can validate the described restrictions (Sec. 5.2.10).

With the *basic data type validators*, all subtypes of a basic data type can be validated as they can only narrow the defined restrictions (see table 5.1). All nodes in VSL context models that can store values must be derived from a basic data type by definition. Therefore all context values that are stored in the VSL can be validated according to the restrictions given for the basic data types (<R.49>). *<R.49>*

To validate if a VSL type *derives* another VSL type correctly, it has to be checked, if the restrictions are narrowed and not widened. The restrictions for a number type can be checked for validity by validating that the new restrictions are valid with the current restrictions of the type that gets derived. The validation, if a regular expression narrows another regular expression for a text type, can typically not be done as the problem has *non-polynomial complexity* [MS72].

Semantic Validation of Composed Types

Typically context nodes in a context model will not be *isolated* but *composed* to represent context together. The correlation of context nodes cannot be validated with the *basic data type validators* as they only validate isolated nodes.

The entire *inheritance chain* is contained in each VSL context node including purely aggregating nodes that do not have a value themselves (e.g. `deviceA` in listing 5.1). Type-specific validators can be selected based on the ModelIDs in the type chain.

Type-specific validators are programs that can either be dynamically linked to a KA at run time or be added via the μ-middleware property of the VSL (Sec. 6.2).

An example for correlated context nodes is an IP address with its corresponding subnet identifier. The validity of the IP address is defined by the subnet identifier. Via the *basic data type validator* for *number* the validity of the IP address and that of the subnet identifier can be checked independently. But as both are correlated it could be desired to provide a validator for the composed type.

Having the *inheritance chain* of each VSL node, such *type-specific validator* can be identified by traversing the derivation chain of a context node from left (most functionality specific) to right (least functionality specific). Following this order, the first identified existing *type-specific validator* is applied. The described process implements dynamically extensible semantic validation (<R.23>). The end of each derivation is a basic data type. Therefore, a validator is found for each context node. *<R.23>*

Though the VSL meta model allows such validation, it was not implemented and tested so far. Depending on the implementation it may not slow down the processing of the VSL context models instances significantly. At the same time it would add additional security for VSL context operations since context changes to invalid values would automatically be rejected (<R.49>). *<R.49>*

Context Value Validation

All instances of VSL nodes that contain data must be derived from the basic data types *text* or *number* (Sec. 5.2.10, Sec. 5.2.11).

As described before, the CMR is designed to contain valid context models only. This facilitates the instantiation of context nodes. When loading the context models in the node's *inheritance chain*, the models can simply be instantiated one after another from right to left, overriding all values and restrictions. No –possibly computation-intense– validation is required as it was done when checking-in the context model at the CMR (<R.50>). *<R.50>*

The result of the context model instantiation via the VSL is a self-contained knowledge tree. All (inherited) properties are present in the instance. This reduces the effort for the validation of any VSL context node to applying the *basic data type validators* with the given restrictions of the instance.

The validation of values using the *basic data type validators* scales as the basic data type is always the last type in the *inheritance chain* of a node, and as the *restrictions* are fixed and directly available at a node (Sec. 5.2.10, <R.50>). *<R.50>*

Each value that is stored in the VSL is *checked for validity* as described and rejected when the validity is not given. As the validation happens automatically it is simple-to-use (<R.29>). As it cannot be circumvented it introduces security-by-design (<R.49>). This *ensures the integrity* of all values in VSL knowledge base according to the restrictions given by the basic data types (<R.49>). *<R.29>* *<R.49>* *<R.49>*

5.2.15 Expressiveness

With the given basic data types *all data* that can be expressed on a digital processing unit can be represented by putting its binary representation in a text node such as

```
<blob type="/basic/text" regularExpression="[01]*"></blob>
```

<R.28> The above definition of a VSL data type `blob` (binary large object) proofs the statement (<R.28>). However, such a representation of data seems not useful as it renders the purpose of context models, to introduce semantics, void.

<R.25> The *hierarchical context reasoning* (Sec. 5.6.6, Sec. 8.3.2) demonstrates how the VSL programming abstraction allows *different levels of abstraction* (<R.25>) being represented and linked automatically via advanced reasoning services.

When using a cascade of services that automatically create context on different levels of abstractions (Sec. 5.4.3), it makes sense to store raw data as described above. Services that know the semantics of the raw data can access them via the VSL, avoiding information loss that could be introduced by transforming the data into another representation. Other services can access the context representation on the abstraction level they need (Sec. 5.2.11, see Fig. 8.4, Sec. 8.3.2).

5.2.16 Dynamic Semantic Extension of the VSL Meta Model

The VSL meta model can be dynamically extended with semantics at run time.

Understanding the implementation of this feature requires understanding of the VSL service interface. Thus, this feature is explained in Sec. 5.7.3.

5.2.17 Virtual Context

The VSL provides a novel mechanism that is called Virtual Context. It extends the VSL meta model with a fourth type that is called Virtual Node. It allows direct coupling of services using context nodes.

Understanding the implementation of this feature requires understanding of the VSL service interface. Thus, this feature is explained in Sec. 5.3.4.

5.2.18 A Self-Managing Dynamically Extensible Collaborative Ontology

The instantiation of context models in a VSL context repository happens by instantiating context models that are stored in the CMR[20]. As described in Sec. 5.6.5, the VSL programming abstraction fosters the use of one context model per service.

[20]Via the caching that is described in Sec. 5.7.1 context models can also be deployed locally for testing without uploading them to the CMR. For sharing context models, the CMR should be used as it manages the distribution of context models.

The instantiation over the CMR forces developers to *publish their context models* if they want to share their services. Publishing context models is requirement for reusing them which is fostered by the VSL (Sec. 7.4.3).

As context models must be uploaded to the CMR for instantiation in Smart Spaces, the content of the CMR defines what can be represented in a VSL Smart Space. The VSL types (Sec. 5.2.9) represent *inheritance* relationships ("*is-a*"). The *hierarchical addressing* (Sec. 5.2.2) represents *composition* relationships ("*has-a*"). Additional dependency semantics can be added (Sec. 5.7.3). The described properties make the CMR an *ontology* for Smart Spaces.

Collaboration is implemented by accepting submissions from everyone, and by fostering reuse of existing context models to create new ones (<R.42>). *<R.42>*

New context models in the CMR can immediately be instantiated in VSL Smart Spaces as described in Sec. 5.7.1. This implements *dynamic extensibility* (<R.43>). *<R.43>*

The automated validation of context models submissions at the CMR (Sec. 5.2.14) ensures the integrity of the CMR. If non-existent context models are referenced, or restrictions are violated, a context model is rejected and not disseminated over the CMR.

The collaborative standardization mechanisms that are described in Sec. 5.6.4, and Sec. 7.4.3 work autonomously. They result in a self-managing convergence of context models.

The deployment of context models into Smart Spaces works autonomously.

As a result, the CMR implements a *self-managing ontology* (<R.3>). *<R.3>*

5.2.19 Assessment of the new Hybrid Meta Model

Sec. 5.2 gave a brief summary about the advantages of different meta modeling approaches that were assessed in Sec. 3.9. Table 3.3 summarizes the approaches more in detail. The last column of the table is the assessment of the VSL meta model.

Following the ratings in the table are discussed according to the criteria given in Sec. 3.9. After each criterion the rating from table 3.3 is given, e.g. (++). They follow directly from the design of the VSL context model as *hybrid* of the existing approaches.

Human Related

The *comprehensibility* (++) of the VSL meta model is as high as typical logic based meta models. It can express *subtyping* and *composition* relationships that can easily be understood by humans (Sec. 5.2.11).

The *complexity* (++) is low as the VSL meta model uses a simple markup (Sec. 5.2.5) that is close to the hierarchical addressing schemes (Sec. 5.2.2). At the same time it supports object oriented design (Sec. 5.2.11).

Collaboration (++) is supported via the CMR for sharing (Sec. 5.2.8), the use of namespaces for structuring (Sec. 5.2.3), and the possibility for crowdsourced convergence (Sec. 5.6.4).

Diversity Support

The VSL meta model represents *dependencies* (++). Subtyping ("is-a") and composition ("has-a") dependencies between contexts can be expressed directly with the meta model (Sec. 5.2.11). In addition, semantic relations can be added dynamically at run time (Sec. 5.7.3).

Timeliness (++) is supported via automated context versioning (Sec. 5.2.1, Sec. 5.3.2).

Dynamic extensibility (++) of the VSL ontology is implemented by adding context models to the CMR (Sec. 5.2.8), and extending instantiated context models via the *list* nodes (Sec. 5.2.10). *Semantic extension* can be implemented via additional primary contexts (Sec. 5.7.3).

Machine Related

Reasoning support (++) is given by the implemented ontology for Smart Spaces, and the sharing of context models (Sec. 5.2.18). Via the semantic extensibility with additional search providers, building blocks for reasoning can be added at run time (Sec. 5.7.3).

Distributed processing (++) is implemented by the self-containment of the VSL context nodes (Sec. 5.2.11), and the access control metadata (Sec. 5.2.1). It is supported by the distribution transparency that is implemented via the programming abstraction (Sec. 5.6.1).

Syntax validation (++) is implemented via the use of markup for representing context models which implies an underlying structure that allows validation (Sec. 5.2.5).

Semantic validation (++) is implemented for data types via rooting all VSL types of context nodes that can carry values in the basic data types *text* and *list*. Via the ontology, additional semantics can be validated by applying logic (Sec. 5.2.14).

The self-containment of each context node as tuple with direct access over its instance address implements *efficient parsing* (++) (Sec. 5.2.11).

Conclusion

The VSL meta model fulfills the requirements for being a *suitable meta model* according to the criteria from Sec. 3.9 (<R.40>). *<R.40>*

5.3 The VSL Context Interface

58. Fools ignore complexity. Pragmatists suffer it. Some can avoid it, geniuses remove it.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

The VSL programming abstraction is implementation agnostic. The implementation is hidden behind the functional VSL interface that is presented in this section. As described in Sec. 3.2.4, the hardware that runs the software that provides the VSL programming abstraction for services is expected to be distributed. Therefore the programming abstraction is designed to be usable as distributed system though having a single computing node is sufficient.

The VSL is implemented as Peer-to-Peer (P2P) system as presented in Ch. 6. In this context, the VSL implements a context overlay on top of distributed computing nodes.

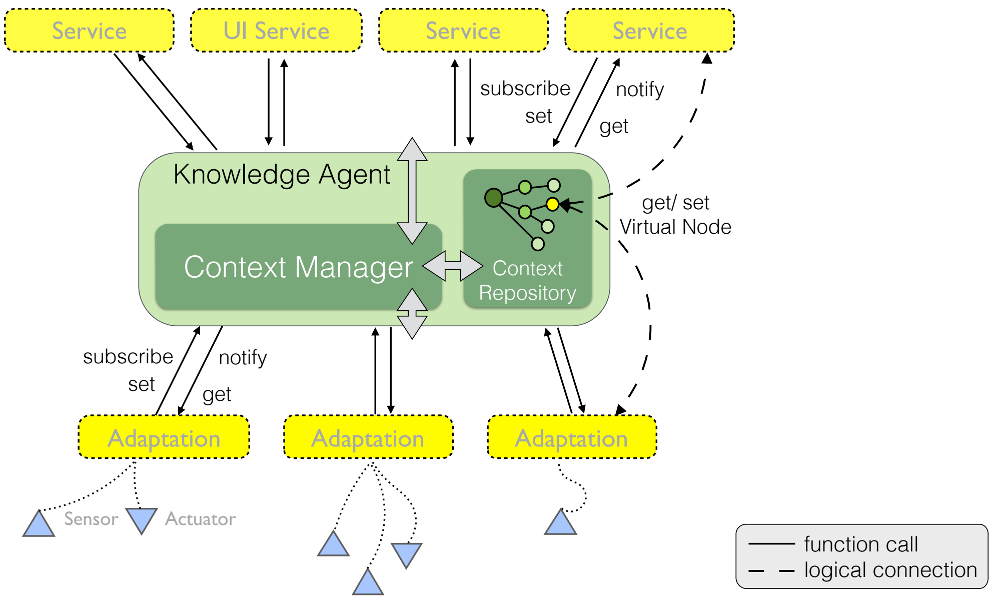


Figure 5.3: Context Management Architecture of the VSL.

The entities that implement the VSL are called *Knowledge Agents (KAs)*. Fig. 5.3 shows a KA and its main components. From the perspective of a service developer, the VSL is a context provider that provides a context repository and context routing (see quote in Sec. 5.2.13) for services.

The functional entities of a KA that are shown in Fig. 5.3 are the *context manager* and the *context repository*. The *context manager* provides the functional interface that is introduced in this section. The *context repository* stores context for services.

5.3.1 VSL Application Programming Interface

Simplicity is the ultimate sophistication.

Anselm, The Recognitions (1955) p. 457, William Gaddis, American novelist, (1922-1998).

The API of a KA consists of *13 fixed methods* that can be grouped into the functional domains *context access*, *access control*, and *Virtual Context*. The presented interface is used for accessing context that is stored on a KA, and for inter-service communication. It is the only interface that is used by services to access functionality that is provided over the VSL in a Smart Space.

<R.29> Using a *fixed* interface simplifies context interaction (<R.29>). Using a *unified* interface removes the artificial separation between orchestration services and adaptation services that is introduced by the state of the art pervasive computing middleware (Sec. 4.5, <R.6>). It supports the *portability of services* as all services use the same functional interface (<R.8>), and it enforces *service composability* as all services are inherently interface compatible (<R.36>).

<R.6>

<R.8>

<R.36>

Context Access

The following API functions are used by services to *access* context via a KA. The functionality is described in detail in Sec. 5.3.2.

- **get** *[nodeAddress]* returns the context at the specified address.
- **set** *[nodeAddress] [value]* changes the context at the specified address.
- **subscribe** *[nodeAddress] [callBack]* subscribes the context subtree that starts at the specified node address. On changes of a context node within the context subtree all subscribers are notified via their specified callback functions.
- **unsubscribe** *[nodeAddress]* unsubscribes an existing subscription on the context subtree that starts at the specified address.
- **lockSubtree** *[parentAddress] [callBack]* locks a VSL subtree for exclusive use. The callback is called when a lock times out before being unlocked.
- **unlockSubtree** *[parentAddress]* unlocks a VSL subtree from exclusive use and commits all changes in the unlocked subtree since the locking.
- **revertSubtree** *[parentAddress]* unlocks a VSL subtree from exclusive use and withdraws all changes in the unlocked subtree since the locking.

The `get` and `set` operations are used to retrieve and change context in the VSL (Sec. 5.3.2).

The functions `subscribe` and `unsubscribe` are used to subscribe for notifications via callback on the changes of a context node or the subtree the context node is the root of (Sec. 5.3.2).

The functions `lockSubtree`, `unlockSubtree`, and `revertSubtree` are used for implementing transactions (Sec. 5.3.2).

Access Control

To access the VSL, services must authenticate at a KA via certificates. See Sec. 5.3.3. The following API functions are used by services to *authenticate* at a KA:

- **registerService** *[localName] [serviceCertificate]* associates a service with a KA enabling context access via the API.
- **unregisterService** uncouples a service from a KA.
- **addCertificate** *[shortName] [certificate]* adds another identity passed via the *certificate* to a service using the identifier *shortName*.
- **removeCertificate** *[shortName]* removes the additional identity that is identified by the specified *shortName* from a service.

A service `register`s at a KA with its unique certificate (Sec. 5.5). Services can dynamically register during the run time of a KA (<R.23>). After successful authentication via a service certificate the context model that is associated with a service (Sec. 5.4.1) is instantiated in the context repository of the authenticating KA in the service's subtree (Sec. 5.2.3). The *localName* is combined with the *service name* (Sec. 5.5.1) for creating a service instance specific namespace. *<R.23>*

The `addCertificate` and `removeCertificate` functions are used to delegate the rights of another certificate, e.g. that of a user, temporarily to a service (Sec. 5.3.3). The *shortName* is used as identifier for removing the certificate later again, e.g. when a user logs off. This allows to create services with *multi-user support* (<R.2>). *<R.2>*

Virtual Context

The following API functions are used by services to *register Virtual Context* at a KA. See Sec. 5.3.4.

- **registerVirtualNode** *[nodeAddress] [callBack]* registers a Virtual Node callback on the VSL node that is specified by the address.
- **unregisterVirtualNode** *[nodeAddress]* removes the link to a callback-function from the VSL node that is specified by the address.

`registerVirtualNode` and `unregisterVirtualNode` are used to register Virtual Node callbacks in the VSL (Sec. 5.3.4).

5.3.2 Context Access

The VSL provides context management for services (Sec. 5.6.5, <R.5>). This section details the API that is offered by the VSL programming abstraction to access context. *<R.5>*

Addressing

Context nodes in the context repositories of the KAs that build the VSL can be accessed via hierarchical addresses as described in Sec. 5.2.2. Each KA has a unique ID inside a VSL operated Smart Space. Each service has a unique namespace (Sec. 5.2.3) in the context repository of the KA it authenticates at.

If a service has the ID `serviceId` in its `serviceCertificate` (Sec. 5.3.3), the call `registerService myFirstInstance <serviceCertificate>` on a KA with the ID `agentId` assigns the following unique address space on the KA's context repository to the service:

```
/agentId/serviceId/myFirstInstance
```

Get/ Set

<R.29> Context is exchanged as String with a KA. Entire context subtrees can be retrieved and set using the presented markup representation (Sec. 5.2.5). Using the same representation is done for consistency to simplify[21] the use of the VSL (<R.29>).

Using the described addresses, context nodes with meta data can be queried as in the following example:

```
system@agent_mop: / % get /agent_mop/mySmartDevice/ledRed

<ledRed type="/ilab/led,/ilab/isOn,/derived/boolean,/basic/number"
    version="7" timeStamp="2014-02-12 17:45:16.516" access="rw">
<![CDATA[0]]>
<desired type="/ilab/isOn,/derived/boolean,/basic/number" version
    ="0" timeStamp="2014-02-12 17:44:31.992" access="rw">
</desired>
</ledRed>
```

To obtain only the *value* of a context node, the functional suffix `/value` can be added to an address:

```
system@agent_mop: / % get /agent_mop/mySmartDevice/ledRed/value

0
```

To obtain the entire subtree that originates at a given address, the functional suffix `/*` can be added:

```
system@agent_mop: / % get /agent_mop/mySmartDevice/ledRed/*

<ledRed type="/ilab/led,/ilab/isOn,/derived/boolean,/basic/number"
    version="7" timeStamp="2014-02-12 17:45:16.516" access="rw">
<![CDATA[0]]>
<desired type="/ilab/isOn,/derived/boolean,/basic/number" version
    ="0" timeStamp="2014-02-12 17:44:31.992" access="rw">
<![CDATA[0]]>
</desired>
</ledRed>
```

<R.40> Historic versions of context nodes can be addressed by adding the version number as functional suffix at the end of an address. *Timeliness support* (<R.40>) is implemented

[21]Via *native connectors* the returned context nodes can be transformed into suitable entities of the programming language such as Java objects for further facilitating the use of the VSL. See Sec. 6.3.3.

by versioning of context nodes. which is important for creating pervasive computing scenarios (Sec. 3.9, Sec. 4.5). The following listing shows a context instance with version information (`version` and `timeStamp` properties of the opening tags).

```
system@agent_mop: / % get /agent_mop/mySmartDevice/ledRed/3
<ledRed type="/ilab/led,/ilab/isOn,/derived/boolean,/basic/number"
    version="3" timeStamp="2014-02-12 17:44:59.102" access="rw">
<![CDATA[1]]>
<desired type="/ilab/isOn,/derived/boolean,/basic/number" version
    ="0" timeStamp="2014-02-12 17:44:31.992" access="rw">
</desired>
</ledRed>
```

The functional suffix is only evaluated when no node with the same address exists. If a node with name `myNode/value` exists, its value can be queried via the address `myNode/value/value`. This allows to use all names that are valid in the used markup scheme as node names, such as "`myNode/value`". (Sec. 5.2.5).

Subscriptions

The VSL supports subscriptions on its context nodes. Authenticated services can subscribe a context node if their *serviceID* has read access or write accesses to the node (Sec. 5.3.3). To remain *self-contained*, the *serviceIDs* subscribed services are stored in the *subscriberID* metadata of a context node (Sec. 5.2.1).

If an accessible (*serviceID*) context node or one of its subnodes changes, a service has access to, the KA calls the registered callback in the corresponding service. The callback is registered via the `subscribe` API function. Using the Java connector (Sec. 6.3.3), the abstract interface shown in listing 5.3 has to be implemented.

```
public interface ISubscriber {
    void notificationCallback(String address);
}
```

Listing 5.3: The `ISubscriptionHandler` interface that is used to implement notification callbacks in the prototype.

As can be seen in listing 5.3 the address is passed to the callback and not the new value. Not passing the value is done for three reasons:

- The operations in a service that follow a notification may require more context than just the value that would not have been disseminated in a value-based notification broadcast, and that can be queried more efficiently via the API.
- The interest by the subscribers in retrieving the actual value might be sparse, and sending node values that are bigger than node addresses leads to unwanted overhead (<R.50>). *<R.50>*
- Explicitly sending notifications gives control over the dissemination (<R.49>). *<R.49>*

Passing the address is necessary as subscribers of a parent node should be informed which child changed.

As described in Sec. 5.2.11, the hierarchy of the VSL information model reflects different degrees of abstraction (Sec. 5.2.2). Subscribing on a specific depth in the address hierarchy makes a subscriber automatically keep track of changes on all subordinate levels.

Via the described mechanism the granularity of subscriptions can be controlled by subscribing on nodes of different hierarchy levels that result from the hierarchical addressing (Sec. 5.2.3). A service that is interested in a single leave value could subscribe the leaf. A service that is interested to be notified of changes in an aggregate can subscribe on an inner node of the VSL (<R.25>). *<R.25>*

The passing of the address that changed to the callback function in a service allows to register the same callback for multiple context nodes. By evaluating the passed address, the handler can change its behavior. As an example, the same notification handler that prints an alert on a node value change can be registered to all nodes where this functionality should apply. Though the same function is registered as callback for all nodes then, it can behave differently based on the address the KA passes. This turned out to reduce the implementation complexity of services in practical use (<R.1>). *<R.1>*

Transactions

As described in Sec. 5.2, context models typically contain nodes with semantically correlated data. The correlation can require *atomic* change of nodes in one or multiple subtrees.

As an example the IP address and its subnet from Sec. 5.2.14 are used again. An adaptation service (Sec. 8.3.1) that controls the network interface of a host subscribes the parent node of the different addresses that specify the network interface configuration. If it starts reconfiguring the interface at the first change notification it receives, the network interface might become unreachable as the combination of IP and subnet mask may be invalid before all changes are done. The example is especially problematic if the interface of an unattended Smart Device is managed by the adaptation interface and the misconfiguration makes the entire device unreachable.

As solution to the described problems, the VSL supports *transactions* via the *locking* of subtrees. Services that have access to a node can lock the entire subtree below the node for a certain time. While holding a lock, a service can obtain other locks to implement a transaction over multiple subtrees.

Notifications about changes are only sent out to other *serviceIDs* than the locker's *serviceID* when a lock is released. This enables a service to change multiple context nodes before other services get notified or can access context nodes in a locked subtree.

To prevent starvation, locks time out. To be notified of a lock time-out, a locking service has to provide a callback that is called when the lock times out before being released via an `unlockSubtree` call. A timeout leads to a rollback of the locked subtree of the context repository. If a service wants to abort the changes, it can call the `revertSubtree` method that sets the entire subtree to the state it had when the lock was acquired.

The lock mechanism implements a form of distributed transaction. Transactions are typically characterized by the Atomic Consistent Isolated Durable (ACID) properties [TVS06, FHA99]:

- *Atomicity* means that a transaction happens indivisibly. A VSL lock prevents all but the locking service to access a locked subtree. Notifications to all processes but the locking one are only sent when the lock is released. The locking process gets its notifications immediately. If a lock is not committed all actions are rolled back. The described features implement indivisibility.

- *Consistency* means that a transaction does not violate a system's invariants. As VSL transactions consist of calls to regular VSL commands this requirement is fulfilled.

- *Isolation* means that concurrent transactions do not interfere with each other. This is guaranteed as subtrees can only be locked by one service at a time. As subtrees are based on the hierarchical address space, different subtrees cannot overlap. The context repositories is divided into disjoint subtrees (Sec. 5.2.2). Different locks are on different subtrees and do not interfere with each other. Only the current locker can lock and commit subtrees of a locked tree. This is useful to refine / narrow locks. When a parent transaction is committed, possible child transactions remain open and even be withdrawn later. In such a case only the notifications of the context nodes that are now outside the subtree that remains locked are disseminated.

- *Durability* means that changes are permanent on commit. VSL operations are write through. For rollback a log with rollback actions is created that is executed when a transaction does not get committed before a timeout. The lock is released only after the rollback in this case. When a KA crashes before a transaction is done, it automatically rollbacks the entire transaction when coming up again. This is possible as the rollback log is always written before doing a change. This enhances durability even on KA failures.

As the ACID properties are fulfilled, the lock mechanism of the VSL implements transactions.

Orchestration services may require multiple nodes in distributed context repositories to be changed together. As the VSL provides location transparency via its API this can be done by transparently acquiring locks on different KAs. If a service implementation ensures that none of the locks expired via callback before the last lock is released (e.g. by timely renewing of locks), *distributed ACID* is implemented.

List Access

The *list* is the only generative type that allows services to add context nodes to an instance of their corresponding context model (Sec. 5.6.5) at run time. It allows dynamic extensibility of context model instances (<R.23>). As the API in Sec. 5.3.1 shows, there are no function calls to manage lists. Instead, the VSL uses its own Virtual Context mechanism (Sec. 5.3.4) to implement interaction with the basic data type *list*. *<R.23>*

An instance of *list* is shown in listing 5.2. The context model is shown in listing 5.4.

```
<list allowedTypes="" minimumEntries="0" maximumEntries="">
  <!-- [listAddress/addNodeAt/[position (set 0 for beginning, size+1
       for end)]
  [modelID] --> <addNodeAt type="/system/list/addNodeAt"></addNodeAt
      >
  <!-- [listAddress/deleteNodeAt [listPosition] -->
  <deleteNodeAt type="/system/list/deleteNodeAt"></deleteNodeAt>
  <elements type="/system/list/elements"></elements>
</list>
```

Listing 5.4: Context model of the basic data type *list*

As the comments show, context nodes can be added to a VSL *list* by calling the address:

```
[listAddress]/addNodeAt/[position (set 0 for beginning, size+1 for
    end)] [modelID]
```

An example call is `set myList/addNodeAt/0 /basic/text` which adds a context node of type `/basic/text` at the beginning of the list.

Nodes can be removed from a list position via:

```
[listAddress]/deleteNodeAt [listPosition]
```

An example call is `set myList/deleteNodeAt 1` which deletes the first context node in the *list* that was added at the previous example. On deletion of a node, all subsequent nodes are automatically consolidated. If a node with the current number 6 gets deleted, all following nodes are decreased in their number. As the number locators of the nodes are discovered using the size attribute of the *list*, or the locator-id- split, the shift in the numbering is not a problem. Prior to deletion the entire list is locked (see transactions) to prevent concurrent access.

Nodes can be accessed by calling `get` or `set` on the position of the desired node:

```
set [listAddress]/1 [value]
get [listAddress]/1
```

An example call is:

```
system@agent_mop: /agent_mop % set listAddress/1 "hello world!"

system@agent_mop: /agent_mop % get listAddress/1
<efpr2u5elkr4o7uph5i07r25ijb type="/basic/text" version="1"
    timeStamp="2014-02-12 20:26:20.116" access="rw">
<![CDATA[hello world!]]>
</efpr2u5elkr4o7uph5i07r25ijb>

system@agent_mop: /agent_mop % get listAddress/1/value
hello world!

system@agent_mop: /agent_mop % get listAddress/size/value
1
```

The listing shows that the setting of a value creates a new context node with a random name. The logic of the VSL list provides the number of available nodes (last line of the listing) and allows to access all list elements via their serial number.

5.3.3 Access Control

Services authenticate at the VSL when establishing a connection to their KA by mutually exchanging certificates. Sec. 5.5 describes the security architecture of the

VSL in detail. A *service certificate* contains multiple *accessIDs* that are delegated to the service by a user on installing the service on her Smart Space (Sec. 7.3).

The VSL provides access control by comparing *accessIDs* from *service certificates* with the IDs that are stored in the metadata of each context node (Sec. 5.2.1). The *readerIDs* are evaluated for `get` requests, the *WriterIDs* for `set` requests. A context node can be accessed by a requesting service if at least one of its *accessIDs* from the service certificate (see below) is available in the corresponding ID list of the node.

Each context node stores its access rights making it fully independent of other nodes. The self-containment happens in accordance with the *key-value properties* of the meta model described in Sec. 5.2.4. Having all data that belong to a context node in its metadata allows fast processing (<R.50>), and it makes the access control independent of external entities which enhances the *dependability* of the VSL (<R.30>), e.g. in case of partitioning of the VSL (Sec. 5.6.1).

<R.50>

<R.30>

The ID "*" is used to define public access. If it is present in the corresponding access ID list of a context node, any ID can be used to access the node.

The access rights are stored in the context models in the CMR and copied into the context repository of the KA a service registers to when the corresponding context model (Sec. 5.4.1) gets instantiated. The access rights cannot be changed at run time for security reasons (Sec. 5.5).

The following listing shows the model of the *lamp* from Sec. 5.2.11 with public read access and public write access to the *desired node* that is used to inform the autonomous control cycle of an adaptation service of a desired state change (Sec. 8.3.1).

```
<lamp reader="*">
  <isOn type="../isOn" reader="*">
    0
    <desired type="../isOn" reader="*" writer="*">
    0
    </desired>
  </isOn>
</lamp>
```

The *accessIDs* of a service are stored in its *service certificate* (see Sec. 5.5). For implementing diverse pervasive computing scenarios, it is required to change the access rights of a service dynamically at run time, e.g. for *multi-user support* (<R.2>)

<R.2>

Via the `addCertificate` and the `removeCertificate` methods it is possible to delegate additional *accessIDs* to a service by presenting additional certificates such as *user certificates.* An example for such rights delegation is shown with the generic user interface service (Sec. 8.3.5) which has only basic access rights in the service certificate that get extended via user certificates when users with certain rights log on.

The described access control mechanism is transparent for services. This makes it *simple-to-use* (<R.29>, <R.1>). In addition it prevents service developers from not using it (<R.49>).

<R.29>

<R.1>

The role of the described *access control* in the security architecture of the VSL becomes more clear in combination with the explanations in Sec. 5.5, Sec. 6.5, and Sec. 7.3.

<R.49>

5.3.4 Virtual Context

Virtual Context is the central functionality of the VSL that makes it fundamentally different from the assessed state of the art (Sec. 4.5). Virtual Context combines the mostly static structure of the VSL context models with the dynamic functionality of services.

VSL context nodes can have four types as described in Sec. 5.2:

- *Text* nodes store textual values.
- *Number* nodes store numerical values.
- *List* nodes store elements of other types and are the only nodes that allow to change the structure of a context model at run time.
- *Composite* nodes contain other nodes but no value.

Via multi-inheritance different node types such as *composite* and *number* can be combined to create an inner VSL node that can store a value.

The listed node types are called *regular nodes* as they are regularly served by the VSL context repositories via the KAs as described in Sec. 5.3.2.

Definition ***Regular Node***

The term regular node is used for all context nodes that are stored in, and served by the context repositories of the KAs that span the VSL.

Regular nodes are similar to a distributed database. In the prototype implementation, a database is used as back-end for regular nodes on each KA which provides *scalability* (<R.50>) and persistence which enhances the *dependability* (<R.30>) on failure as the context remains available when a KA is restarted (Sec. 6.3.5). *<R.50>* *<R.30>*

A difference to existing distributed databases is the connection of the database back-end with the VSL meta model. Different to typical databases, the VSL supports hierarchical structuring, inheritance, validation, and the dynamic extension and use of data types at run time as described in Sec. 5.2.

<R.31> Regular nodes implement asynchronous communication and loose coupling (<R.31>) between services via stored context and subscriptions. The VSL manages regular nodes autonomously including security and context retrieval (Sec. 6.3).

Virtual Nodes

Virtual Nodes are the fifth and last predefined node type of the VSL.

Definition ***Virtual Node***

The term Virtual Node is used for VSL nodes that provide context via a function callback into a service.

Virtual Nodes can be compared to dynamic web pages that are not permanently stored as static pages but get dynamically generated on request based on the parameters that are given in the URL[22].

Virtual Nodes do not store values in the VSL context repository. Instead they register a callback on a VSL node that gets called by the KA when the node is accessed. See Fig. 5.3 on the right. As a result, the availability of a Virtual Node value depends on the availability of the corresponding service that serves the value via the registered callback.

Virtual Nodes implement synchronous communication. The VSL manages the context routing (<R.1>) and provides basic access control (Sec. 5.3.3, <R.49>) to Virtual Nodes by invoking the callback only when the ID of the accessing entity is allowed to read (`get` request) or write (`set` request). *<R.1>* *<R.49>*

Accessing Virtual Nodes is transparent for requesting services as the regular 13 VSL API methods (Sec. 5.3.1) are used to access *regular nodes* and *Virtual Nodes*. Fig. 5.3 shows the access to Virtual Nodes via the regular VSL API on the right. Each of the shown accesses from services could be to a *regular node* or a *Virtual Node* as it is transparent for services. On the top right and the bottom left, the major API functionality for services is shown.

Virtual Context implements *access transparency* (<R.12>) and *persistence transparency* (<R.18>). Via the type `/system/virtual` in their inheritance chain, Virtual Nodes can be identified if required. *<R.12>* *<R.18>*

For implementing transparent access, the callbacks of Virtual Nodes must provide implementations for the methods that can be called on a regular node. In the prototype implementation, the abstract Java interface of a Virtual Node callback is as follows:

```java
public interface IVirtualNodeHandler {
    String get(final String address, final String readerIDs);

    void set(final String address, final String value, final String
        writerIDs);

    void subscribe(final String address, final String subscriberID,
                   String subscriptionCallbackAddress);

    void unsubscribe(final String address, final String
        subscriberIDs);

    void lockSubtree(final String address, final String lockerID,
                     final String abortCallbackAddress);

    void unlockSubtree(final String address, final String lockerID);

    void revertSubtree(final String address, final String lockerID);
}
```

Listing 5.5: The `IVirtualNodeHandler` interface that is used to implement Virtual Nodes in the prototype

The callback function is registered by a service at its KA with the API function `registerVirtualNode [nodeAddress] [callback]`. The KA routes all requests to a node that is declared virtual to the callback. The same happens for requests on

[22] e.g. http://pahl.de/?site=_contact&subject=I+like+your+thesis

subnodes of a Virtual Node that are not specified in the context model as described next. The answer to the callback is routed back through the VSL to the caller. This makes it transparent to the receiver if the request was handled by the KA or by a service via callback.

Virtual Subtrees

As can be seen in listing 5.5, all method signatures contain the VSL address that was requested. It is passed to the callback as the VSL does not only forward requests to the VSL address of the Virtual Node (Sec. 5.2.2) but also requests on addresses of subnodes of a Virtual Node that are not specified in the context model. This extends the Virtual Node concept as it implements *Virtual Subtrees*.

Specifying all context nodes in the context model has the advantage that the available subnodes are specified, and can be looked up by developers over the CMR. This fosters the reuse of a service.

Not specifying all subnodes and using the described Virtual Subtree mechanism gives more flexibility as described next. Requests to subnodes of a Virtual Node that are not specified in the context model are routed to the registered callback. This enables passing parameters to the callback function, e.g. by using the addressing schemes of the VSL. More details about *Virtual Subtrees* are introduced in Sec. 5.4.2 after the introduction of the abstract service interfaces.

The power of this novel mechanism is demonstrated in the remainder of this thesis with the *CMR* (5.7.1), the *type search* (5.7.2), and the location search provider (Sec. 6.6.4) as an example of *meta model extensibility* (Sec. 5.7.3).

5.4 The VSL Service Interface

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Eric Raymond, American Open-Source Advocate, (*1957); adaptation of a quote from Fred Brooks, American Computer Scientist, (*1931), Turing Award 1999, The Mythical Man-Month, Chapter 9.

The VSL uses the API described in Sec. 5.3.1 for *inter-service communication.* All services access the VSL via the 13 fixed methods presented in Sec. 5.3.1. All services use the VSL for *inter-service communication.* Therefore the *service interface* was already presented in Sec. 5.3. This section explains its use as *inter-service communication* in detail.

Via its *regular nodes*, the VSL provides context storage for services, supporting context-awareness (<R.5>). The VSL context repositories can be used for exchanging information between services. Via the *subscriptions* (Sec. 5.3.2), coupling between services can be implemented over context. *<R.5>*

Via the novel Virtual Context mechanism, context that is available as *procedural knowledge* –generated by functions– becomes accessible as *descriptive knowledge* over the Virtual Nodes. The availability of *descriptive knowledge* facilitates the use of the context abstraction (Sec. 3.5, <R.24>). *<R.24>*

5.4.1 Context Models as Abstract Service Interfaces

Using the VSL, services can be coupled over *regular nodes* and *Virtual Nodes.* Services can be coupled over *regular nodes* using *subscriptions* (Sec. 5.3.2). The novel coupling mechanism over *Virtual Nodes* uses context models to directly couple services (Sec. 5.3.4).

The VSL context that is used for *inter-service communication* is structured by context models. **The VSL uses *context models* as *abstract service interfaces.***

The use of a single mechanism, and a single representation for context models, and abstract service interfaces reduces the complexity of Smart Space orchestration significantly. It consolidates the problem domains *context modeling* and *abstract interface design* that are typically separated in the state of the art (Sec. 4.5) to one problem domain (Sec. 3.12).

The creation of context models is rooted in the reality as formal ontologies are a formalization of parts of the physical reality. The design of abstract service interfaces instead is rooted in the creation of programs. The latter is typically further away from the reality of non-programmers than the former. The step of unifying the two tasks facilitates the creation of abstract service interfaces as they automatically emerge with describing (<R.24>) the reality in context models. *<R.24>*

The creation of context models is supported by the VSL meta model (Sec. 5.2). Via the described dual use of context models the meta model support applies to the design of abstract service interfaces facilitating their creation, and providing validation support (<R.39>). In addition the inheritance features of the VSL meta model become usable *<R.39>*

when designing *abstract service interfaces*, making it a *modular* task (<R.26>). *<R.26>*

The described functionality of the CMR (Sec. 5.2.8) fosters *crowdsourcing* as it enables the sharing of abstract service interfaces (<R.48>). Shared interfaces, and fixed methods to access services foster composition (<R.36>) and reuse (<R.35>) of VSL services. *<R.48>* *<R.36>* *<R.35>*

Combined with the separation of service logic from service state, and the requirement to declare the layout of the service state as context model in the CMR, the creation of abstract service interfaces is enforced (<R.32>) via the dual use of context models. In addition, full encapsulation of service functionality behind the abstract interfaces is implemented (<R.34>). The data access happens over the decoupled (regular node) or coupled (Virtual Context) declarative interface. *<R.32>* *<R.34>*

The context discovery mechanism of the VSL automatically provides a service discovery mechanism over the VSL (<R.38>) by using *types* to implement a *locator-id split* (Sec. 5.2.13). A service instance can be discovered searching for the type of its context model. *<R.38>*

Generating Service Stubs

Context models originate in the real world as described above. This makes their creation often simpler for developers than the creation of an abstract interface [Dét02]. A purpose of abstract service interfaces is structuring a program. Though the abstract service interfaces of the VSL can be originated in the real world they automatically provide structure to the service that implements the interface.

The context model of a service can be interpreted as an Interface Definition Language (IDL) as it defines the VSL interface to the service. Interface Definition Language (IDL) definitions in Common Object Request Broker (CORBA) can be used to generate implementation stubs. The same can be done with VSL context model. Generating methods for the management of each context node in the model automatically structures a VSL service. By providing the logic behind the context nodes, a developer can implement a service. See Sec. 8.2.

The brief explanation above shows how the use of context models as abstract service interfaces facilitates the creation of services as the previously unconnected steps of context modeling and abstract interface design are combined. This fosters crowdsourced development (<R.48>). *<R.48>*

5.4.2 Virtual Context Revisited

The meaning of Virtual Context for the VSL programming abstraction becomes more clear knowing about the dual use of VSL context model as representation scheme for parts of the real world, and as abstract service interface.

The previously described inter-service communication over *regular VSL nodes* implements *asynchronous coupling* between services. Virtual Nodes implement *synchronous coupling* between services as requesting them results in direct callbacks into services (Sec. 5.3.4).

Virtual Subtrees can be used to implement *synchronous coupling* between services and *passing parameters*. Assuming a Virtual Node is registered at the address `/myAgent/myService/myVirtualNode` and has no children in its context model, parameters can be passed as follows:

```
get /myAgent/myService/myVirtualNode/parameter1/parameter2/
    parameter3/...
```

Using the regular VSL address separator "/" makes it transparent to a caller that the semantic of the address parts are parameters for a service[23].

Examples for the use of the described parameter passing technique are the services that are introduced in Sec. 5.7, which provide basic VSL functionality such as the context search.

5.4.3 Communication Modes

The VSL *meta model* implements a hybrid of existing context modeling techniques (Sec. 5.2). The described VSL *inter-service communication* implements a hybrid of different coordination modes. See Sec. 3.4.1.

The diversity of pervasive computing use cases requires the VSL to offer all communication modes shown in table 3.2.

Table 5.2 maps the different coordination modes that were introduced in Sec. 3.4.1 to requirements of pervasive computing workflows.

		Temporal	
		coupled	uncoupled
Referential	coupled	time & security critical *video / audio transmission*	security critical *sensor value dissemination to a defined group*
	uncoupled	time critical *clock signal dissemination*	time & security uncritical *ambient information*

Table 5.2: Match of Smart Space requirements to the different coordination modes of table 3.2.

The table shows that referential coupling matches with requirements on security ("who gets the information?") and with the selection ("who is interested in the information?"). Temporal coupling matches with the requirement of short response time.

An example that needs *temporal and referential coupling* is the *surveillance of a space* via video or audio streams.

An example for *temporally uncoupled* and *referentially coupled* communication is *sensor value dissemination to a defined group* of other services.

[23]It is also possible to use arbitrary character in an address such as `get "/myAgent/myService/myVirtualNode/hello world$23?parameter7=42"` but such an address is not transparent for other services.

An example that requires *temporal coupling but no referential coupling* is the dissemination of security uncritical information such as a *clock signal* that should be published to services of a Smart Space.

No temporal and referential coupling is required for the exchange of security uncritical information. An (Sec. 8.3.2) that provides the time of the day out of the set {morning, noon, afternoon, evening, night} could be an example for such a service.

Via *regular nodes temporally uncoupled* (asynchronous) coordination is implemented (<R.20>, <R.22>). Via *Virtual Nodes temporally coupled* (synchronous) coordination is implemented (<R.19>, <R.21>). Via the VSL access control (Sec. 5.3.3), by setting *accessIDs referential coupling*, and by setting the *accessIDs* to "*" *referentially uncoupled* coordination can be implemented. *<R.19> <R.20>* *<R.21> <R.22>*

When using regular VSL nodes, the access control of the VSL nodes is entirely handled by the VSL. It is transparent for a service. It simply changes a context node. The VSL provides access to the *accessIDs* that are permitted, and it sends notification to the subset of those IDs that subscribed the context node or one of its parents (Sec. 5.3.2).

The *subscriptions* implement *temporal coupling* but they are *asynchronous* from the perspective of a *context providing service* as it only sets a context value in the VSL. From the perspective of a *subscribing service*, *Subscriptions* implement *temporal coupling* but as the notifications are not used for disseminating the value, it is less strict than the Virtual Context coupling. The evaluation in Sec. 9.3 shows and evaluates differences between the coupling via *notifications* and *Virtual Nodes*.

5.4.4 Service Oriented Architecture

Via the described design, each VSL service that allows reuse needs a context model. Context models are associated with service over the ModelID that is stored in the certificate of a service (Sec. 5.5). When a service registers at the VSL (Sec. 5.3.1), its corresponding context model is automatically instantiated in the context repository of the KA it registered at. As described above, this makes the service discoverable in the VSL. Services that do not provide any context use an empty context model.

<R.31> As the criteria for *loose-coupling* of services are fulfilled, and as the CMR acts as directory for abstract service interfaces, the VSL implements a SOA (<R.31>):

- *<R.32>* With the context models as abstract service interfaces, *service contracts* are implemented (<R.32>).
- *<R.33>* *Service autonomy* (<R.33>) is implemented as each service runs independently from other services. Services can have dependencies. When all dependencies to other services are fulfilled, a service can run. Using context models as abstract service interfaces, resolving a dependency is equivalent to checking the availability of a data type in a VSL instance. If some dependencies are not fulfilled, running a VSL service is possible. However, a service can typically not serve its purpose in such a case as necessar input is missing (e.g. cascaded reasoning in Sec. 5.4.3).
- *<R.34>* *Service abstraction* (<R.34>) is automatically implemented via the context models.

<R.35>

- *Service reusability* (<R.35>) is enabled via the sharing, the abstract service interfaces over the CMR, and the sharing of services over the Smart Space Store (S2Store) (Sec. 7.2.6).
- *Service composability* (<R.36>) is given via the use of the fixed API. *<R.36>*
- *Service statelessness* (<R.37>) is implemented via the separation of service logic from service state (Sec. 5.6.5) *<R.37>*
- *Service discoverability* (<R.38>) is implemented via the *locator-id split* of the VSL meta model (Sec. 5.2.13) that allows to identify context model instances and thereby services based on their ModelID. *<R.38>*

In addition, comprehensive *abstract interface validation* (<R.39>) is implemented via the validation of the context models (Sec. 5.2.14). *<R.39>*

5.5 Security-by-Design

This section introduces the security architectures of the VSL (Sec. 5.5). Additional elements of the DS2OS security architecture that is presented in Sec. 7.3 complement the presented features for real world deployment with crowdsourced development.

Smart Spaces that are orchestrated by *software* (Sec. 2.4.1), e.g. via the VSL, implement Cyber-Physical Systems (CPSs) that allow software to interact with the physical environment of humans (Sec. 2.6.2). Smart Spaces need to be secured [BCG12, DDMS12, CD12, Lan01, TPR^{+}12]. Major reasons are *security*, *safety*, and *privacy protection.* This section highlights how the VSL programming abstraction implements security-by-design for services.

Via Virtual Context (Sec. 5.3.4) all coupling between software entities in Smart Spaces can be handled over the VSL middleware. The position of the VSL middleware in the middle of each context access and inter-service communication automatically enforces the security policies of the VSL as it must be used for communication.

<R.49> The term *security-by-design* <R.49> expresses that the described VSL security mechanisms are automatically applied to services without service developers having to add special code.

5.5.1 Identification

The VSL is designed to be fully distributed. To reflect the distribution (Sec. 5.6.1) and the autonomy (5.6.2) of the computing nodes that run the KAs that span the VSL, *certificates* are used as secure containers for data.

Each VSL site acts as Certification Authority (CA) that can issue locally valid certificates. To do so, each VSL site has a public-private key pair ($K_{pub}(VSLsite)$/ $K_{priv}(VSLsite)$). It has a Site Local Certificate Authority (SLCA). Connecting the Site Local Certificate Authoritys (SLCAs) hierarchically via a global CA could be done to implement trust between VSL sites but as described in Sec. 8.3.4, connectivity based on explicit trust is proposed in this work.

Each software entity that communicates with the VSL needs a locally signed certificate to connect to a KA (Sec. 5.3.1). A VSL certificate for a software entity A contains:

- A unique name.
- A public key of A, $K_{pub}(A)$.
- The *readerIDs* and *writerIDs* that A has (Sec. 5.3.1).
- A *validity period* that defines the beginning and the end of the validity of the certificate.
- The *ModelID* that identifies the context model that belongs to A.
- A *cryptographic hash* over the *context model.*
- The public key of the VSL site, $K_{pub}(VSLsite)$.

Each certificate is signed with the private key of the VSL site, $K_{priv}(VSLsite)$. As the certificate of each entity contains the $K_{pub}(VSLsite)$, each entity can verify the integrity of the information in a foreign certificate locally using the public key from its *own* certificate without accessing a central entity.

The unique name is used for the namespace of the entity in the context repository (Sec. 5.3.1, Sec. 6.4.1).

$K_{pub}(A)$ can be used to encrypt data for A. This is used for disseminating new keys when re-keying in the VSL μ-middleware implementation for instance (Sec. 6.5.6).

The *readerIDs* and *writerIDs* are used for access control as described in Sec. 5.3.2. For referring to both the term *accessID* is used in the remainder of this document. By storing them in the certificate, a service cannot change them without the SLCA. Sec. 7.3 describes how the certificates can be used to implement user aware access control in future Smart Spaces.

The *validity period* is relevant for *distributed revocation*. Revocation is necessary to remove unwanted certificates (e.g. service removed, certificate compromised). It is required that the clocks of all KAs are synchronized to make certificates expire when their validity period is over. The clock synchronization is implemented by the VSL implementation via the regular alive pings (Sec. 6.5.1). As invalid certificates are not renewed by the SLCA, this implements a distributed revocation. Sec. 7.3 describes a possible automated certificate renewal process.

The *ModelID* identifies the context model that gets instantiated when a service gets started. The context model defines not only the layout of the context storage a service can use and its abstract interface (Sec. 5.2.9) but it also defines the access rights to the context nodes of a service (Sec. 5.3.3). Having the *ModelID* and the *cryptographic hash* over the *context model* in the certificate validates the correspondence to a certain context model and its integrity.

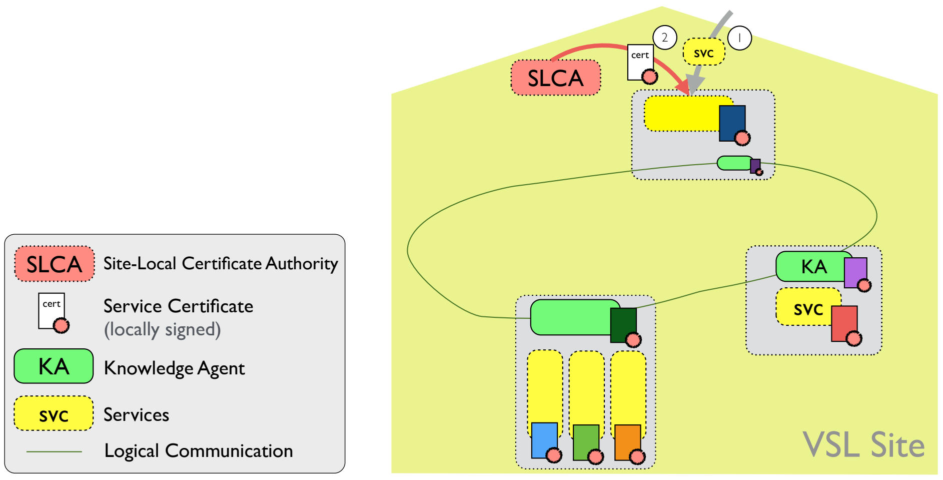

SLCA
Site-Local Certificate Authority
cert
Service Certificate (locally signed)
KA
Knowledge Agent
SVC
Services
Logical Communication
2
1
VSL Site

Figure 5.4: All software entities in a VSL site have a certificate that is signed by the site and used for distributed authentication.

Fig. 5.4 shows that a new service has to be equipped with a certificate before it can register at a KA (1, 2). The figure shows that all entities have a certificate that is issued by the SLCA. The colors indicate that the content of each certificate is different. The red circle illustrates that all are signed with the same key from the SLCA.

Fig. 7.3 shows the described security mechanism in the interplay with the other components of DS2OS. It shows how the presented mechanism allows the implementation of a trust chain from the developer of a service to the computing host in a VSL site that runs the service, and how the user can set the access rights explicitly with the described scheme.

5.5.2 Authentication

The described certificates are used for authentication of entities that interact with the VSL.

Knowledge Agents

Each KA has a certificate. To implement the VSL as a protected domain, KAs authenticate mutually when establishing a connection. The communication between KAs is encrypted. See Fig. 5.4.

Services

<R.4> *<R.48>*

Each VSL service has a certificate as it is required for registering to a KA. Such identification anchor is needed to allow secure use of services that are deployed from outside and shared by distributed developers (<R.4>, <R.48>). Services authenticate mutually with a KA when establishing a connection. The communication between services and KAs can be encrypted. As both are typically running on the same physical machine this may not be necessary if this machine is trusted. Unencrypted communication typically introduces a security risk. See Fig. 5.4.

Other Entities

Users or other entities can have certificates. Such certificates can be used to temporally add access rights to services (Sec. 5.3.3). See Fig. 5.4.

5.5.3 Authorization

The access rights are stored as part of a context model in the CMR as described in Sec. 5.3.3. Defining “`writer="cameraAccess"`” restricts the right access to services that have the *accessID* `cameraAccess` in the *writerIDs* of their certificate. “`reader="*"`” gives public read access to a context node.

The authorization happens based on the *accessIDs* (*readerIDs*, *writerIDs*) of the certificates as described in Sec. 5.3.3. The access control scheme is applied to context access and inter-service communication.

The VSL access control mechanisms apply for service coupling when context change notifications are used (Sec. 5.3.2), and when Virtual Nodes are used (Sec. 5.3.4). In case of Virtual Context, services can also handle their access control based on the accessIDs that are passed over the callback[24] (Sec. 5.3.4).

[24]In such case, the VSL access rights would be set to “*”.

5.5.4 Rights Delegation

To implement multi-user support (<R.2>) services can temporarily be delegated additional rights. Sec. 5.3.1 describes the API functionality. Sec. 8.3.5 gives an example for the use of the functionality to delegate additional rights to an Graphical User Interface (GUI) service via user certificates. The service has only few rights to access context. When a user logs on, it inherits the user rights and can display the user all context, it has access to. *<R.2>*

5.5.5 Confidentiality

To ensure the privacy of data on the physical storage, the use of encryption in the tuple store of the context repository in each KA is recommended e.g. as proposed in [App12].

5.5.6 Service Dependability

As Smart Spaces are Cyber-Physical System (CPS) (Sec. 2.6.2), it is required that they provide safety and security [BCG12, DDMS12, CD12, Lan01, TPR$^+$12, Kub68, Asi50, Cla93]. To implement pervasive computing scenarios, Smart Space Orchestration (Sec. 2.4.1) with user developed services (<R.4>, <R.48>) is enabled via the presented VSL programming abstraction.

Though the VSL structures and facilitates the development and interplay of services in future Smart Spaces, a VSL operated Smart Space with multiple autonomously interacting services will be a complex system. Emulators (5.6.7) enable the testing of functionality in simulations. Testing can help increasing the dependability of services.

The validation mechanisms of the VSL meta model (Sec. 5.2.14) apply to the application state of services that is stored and routed by the VSL (5.4.1). This enhances the dependability of services.

As third point, the use of state (context) for interaction between services fosters applying formal methods to model VSL Smart Spaces. Formal models of VSL services and their interplay can be used to apply methods for formal validation, to check if an intended behavior occurs, and formal verification, to check if an observed behavior is within a specification, to parts of the emerging software system. Though out of scope of this thesis, following this path may allow certify service functionality in future VSL Smart Spaces (Sec. 10.4).

To illustrate how the use of explicit state for communication between Smart Space services fosters formal modeling, some starting points are given next.

Hoare-Floyd logic

Using context as service interface makes the input and output of service procedures explicit as it must be represented as descriptive knowledge (Sec. 5.4).

Applying Hoare-Floyd logic is a method to validate program functionality by adding and checking invariants for sub functionality [Hoa69, Flo67]. Using the VSL context models facilitates the formulation and the checking of such invariants. The invariants can simply be formulated as expected values or value ranges for context nodes in the current VSL state.

Hoare-Floyd logic is a possibility to validate service functionality [Hoa69, Hoa78]. Applying it to VSL services is simplified by the central use of context models.

Automata

As written in Sec. 3.6, automata are a suitable formalism for developing Artificial Intelligence (AI). The described use of explicit context for exchange between VSL services fosters the modeling of services as formal automatons.

Services can be expressed as labeled transition systems. The states of the state transition system are the possible configurations a service interface / context model instance can have. For labeled transition systems and especially their more restricted form Deterministic Finite Automaton (DFA) formal verification methods exist [CES83].

A more concrete example is given next.

Modeling a Software Orchestrated Smart Space

VSL Smart Spaces run multiple services that follow different goals (Sec. 3.6.3). Each service can be seen as a so-called agent that aims to fulfill its goals. As a result, formal modeling techniques for multi-agent-systems apply. The following formalism is adapted from [Woo09].

The state of a VSL Smart Space at a given time is defined by the union of all context repositories, the knowledge base. The knowledge base can be formalized as set E of context nodes e:

$$E = \{e, e', \ldots\} \tag{5.1}$$

The functionality of service can be expressed as set of transitions (S_{Tran}) between states of context nodes:

$$S_{Tran} = \{\alpha, \alpha', \ldots\} \tag{5.2}$$

A run, r, of a service can be expressed as:

$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} \quad \cdots \quad \xrightarrow{\alpha_{n-2}} e_{n-1} \xrightarrow{\alpha_{n-1}} e_n \tag{5.3}$$

Services in a space can transform the environmental state independent of each other. Let $R(E, S_{Tran})$ be the set of all finite runs r. $R^{S_{Tran}} \subset R$ be the subset of R that ends with an action and $R^E \subset R$ be the subset of R that ends with an environmental state. Ending with an environmental state means that a service finishes and set all values it intended to change in the VSL.

The effects of a service can be expressed as subset of the power set of E [25] ($\wp E$) as shown in the following *state transforming* function $R^{S_{Tran}}$:

$$\tau : R^{S_{Tran}} \rightarrow X, X \subseteq \wp E \tag{5.4}$$

If $\tau(r) = \emptyset, r \in R^{S_T ran}$ the system has ended as there are no successor states defined.

[25]The power set consists of tuples of all possible combinations of allowed states in E

With the introduced formal descriptions a VSL Smart Space can be described as:

$$Env = \langle E, e_0, \tau \rangle \tag{5.5}$$

e_0 is the starting state of the environment. In case of the VSL, this state is reached when the context models of all services are instantiated but no service was running yet.

Services can be defined as entities that transform between environmental states, resulting in a specific run over states in E (R^E):

$$Svc : R^E \rightarrow Tran \tag{5.6}$$

The run of a service (R^E) can now be defined as sequence $(e_0, \alpha_0, e_1, \alpha_1, e_2 \ldots)$ if:

1. e_0 is the initial state of Env.
2. $\alpha_0 = Svc(e_0)$.
3. $\forall u > 0, e_u \in \tau((e_0, \alpha_0, \ldots, \alpha_{u-1}))$ and $\alpha_u = Svc((e_0, \alpha_0, \ldots, e_u))$.

A purely *reactive service* (Sec. 3.6.4) can be modeled as:

$$Svc : E \rightarrow Tran \tag{5.7}$$

The difference to the general service definition 5.6 is that the reactive service is not considering the history of the environment but only acting based on the current state.

An example for a reactive service is a heater control:

$$Svc(e) = \begin{cases} \text{heater on} & \text{if } e = \text{temperature too low} \\ \text{heater off} & \text{otherwise} \end{cases} \tag{5.8}$$

For more details see [Woo09]. Difference to the modeling of multi-agent systems that is shown in the book, the VSL programming abstraction does not require the creation of a model in addition to the services. Instead with the use of context models as abstract service interfaces the current state of all services in the VSL (eq. 5.1) is explicitly available and can directly be used. This facilitates the modeling.

Via the timeliness support of the VSL (Sec. 5.2.1, Sec. 5.3.2) there is no differentiation between historic context and current context as the historic context remains available. As a result, reactive services can transparently use historic context for querying older values of a context node.

The unified descriptive VSL interface may facilitate the validation of service functionality. Analyzing and using the emerging possibilities is future work.

Conclusion

The VSL programming abstraction bases services on the formal structure of their context models. This enables the application of additional formal methods to increase the dependability (<R.30>) of VSL operated Smart Spaces. *<R.30>*

5.6 Resulting Properties

This section describes how the presented VSL programming abstraction implements *distribution transparency* (Sec. 5.6.1), *autonomy* (Sec. 5.6.2), *portability* (Sec. 5.6.3), and *collaborative convergence* (Sec. 5.6.4).

The section closes with a discussion how the VSL programming abstraction structures the creation of Smart Space services (Sec. 5.6.5).

5.6.1 Distribution Transparency

<R.11> The VSL is designed as *distributed system* (<R.11>). This design is chosen to provide *scalability* (Sec. 9.4, Sec. 4.5), and to make use of the distributed resources of future Smart Spaces (Sec. 3.2.4).

The VSL implements a *unified system view* (Sec. 3.4) on the virtual representation of a Smart Space by providing *transparency* for services:

<R.12>
- *Access transparency* (<R.12>) is provided via the context models (Sec. 5.2) that are also used as abstract service interfaces (Sec. 5.4), the fixed functional interface (Sec. 5.3.1), and the autonomous context management that is implemented via the KA peers (Sec. 6.4).

<R.13>
- *Location transparency* (<R.13>) is implemented as the VSL routes requests (Sec. 5.3.2) transparently between distributed KAs (Sec. 6.4) that handle requests either via their context repository, or via executing a callback into a service (Sec. 5.3.4).

<R.13>
- *Migration transparency* (<R.13>) is implemented via the *locator-id split* (Sec. 5.2.13) that allows to retrieve services independent of their location on a certain KA. Further details are given in Sec. 5.2.13.

<R.15>
- *Relocation transparency* (<R.15>) is supported by the VSL programming abstraction via Virtual Context that can be used to implement transparent forwarding of requests from one context address to another context address (new service location). More details are given in Sec. 7.2.4.

<R.16>
- *Concurrency transparency* (<R.16>) is implemented by the VSL for access to *regular nodes* via the context repositories. Using the VSL to exchange states of a service (e.g. from a Smart Device via an adaptation service, Sec. 8.3.1) automatically decouples the access to the state of the Smart Device from the Smart Device and from its service. In practical use, regular context nodes are suitable for implementing most pervasive computing scenarios.
  In case of Virtual Context (Sec. 5.3.4), the service callback must be programmed to provide *Concurrency transparency.*

<R.17>
- *Failure transparency* (<R.17>) is implemented by decoupling services from their state. The VSL context of a service exists independent of the service. This makes service failures transparent with regard to the context exchange.
  In case of Virtual Context (Sec. 5.3.4), service failure is not transparent. However, when a service manager (Sec. 7.2.3) brings a failed service up again, it can

be reached over its original bindings to Virtual Nodes in the VSL. This implements at least failure tolerance.
Via the persistence functionality of the used database back-end (Sec. 6.3.5) in the implementation, failure tolerance can be implemented for the KAs as well.

- *Persistence transparency* (<R.18>) is provided via the unified API (Sec. 5.3.1) that hides the origin of context values. For a requester it is transparent, if a context value is served from the context repository of a KA, or if is dynamically generated via a service and returned (Virtual Context). *<R.18>*

Fulfilling the described properties results in *distribution support* (<R.11>). Sec. 6.4 describes how the described properties are reached autonomously. *<R.11>*

5.6.2 Autonomy

The infrastructure in a Smart Space Back-End is heterogeneous and distributed (Sec. 3.2.4). This leads to different properties of communication links and resources on computation nodes. It cannot be expected that the underlay of computing nodes, consisting of the actual connections between the nodes, and their hardware, is dependable.

For providing a high level of autonomy, the VSL implements a fully distributed design. Central components are avoided as they are critical for the *scalability* of the system, and for its operation. Concerning the VSL as middleware, *partitioning* of computing nodes that span the VSL via their KAs can make central components unreachable for one or several of the emerging partitions. This makes a fully distributed design even more important.

The VSL is designed to make each computing node autonomously operable (<R.3>). In particular, the following aspects are relevant for the autonomy: *<R.3>*

- The VSL *meta model* implements a data structure of *self-contained tuples* that only have dependencies during instantiation (Sec. 5.2).
- The *inter-service communication* uses the local KA as communication endpoint that forwards requests and can handle partitioning for services (Sec. 5.4).
- The *security architecture* of the VSL is based on certificates that can be verified by each KA locally (Sec. 5.5).
- The *CMR service* (Sec. 5.7.1) implements a decoupling from the central CMR by caching and serving context models from its cache. This allows to instantiate cached context models even when the CMR is not reachable.
  The same principle is applicable to each KA. It can cache all locally instantiated context models. The local caching allows to reinstantiate context models locally without connection to the *CMR service.* This is especially interesting for the basic data type `list`, and for starting new instances of existing services on a host.
  Different caching strategies [Tan01] can be optimal depending on the available resources (e.g. Least Recently Used (LRU)). They determine how independent a VSL site is from the global CMR, and how independent a KA is from the CMR service.

The functionality that is provided by the VSL programming abstraction is fully transparent for services. It is hidden behind the fixed API (Sec. 5.3.1), and the context models (Sec. 5.2.8).

5.6.3 Portability Support

Portability of services is a fundamental challenge for Smart Space Orchestration. Portability enables services to run in diverse Smart Spaces (Sec. 4.5, [KFKC12, DHK11, DMA$^+$12]). *Portability* also means that the same application-executable can run on different machines [Ber96, TVS06].

Services that implement pervasive computing scenarios typically have dependencies. *Portability* has different meanings in this thesis:

1. *In relation to Smart Device instances*, portability refers to the independence of services from concrete Smart Device instances. It can be implemented by adding a layer of abstraction between a service and a device. In an Operating System (OS) this layer is typically called device driver. In the VSL this role is taken by the adaptation services (Sec. 8.3.1). The VSL context models are the abstract service interfaces that implement the decoupling (Sec. 5.4.1).
2. *In relation to other services*, portability describes that a service that depends on other services is independent of concrete implementations of the other service. This concerns the abstract service interfaces in particular. The VSL context models are the abstract service interfaces that implement the decoupling.
3. *In relation to the middleware*, portability means that a service is not pinned to a certain middleware because of the particular support functionality it offers (Sec. 4.5). This point becomes more clear in Sec. 6.2.
4. *In relation to service executables*, Portability refers to the ability to run the executable on diverse hardware architectures with diverse OSs. The need for this aspect of portability is expressed by <R.45>.

The VSL programming abstraction solves the first three challenges. They are referred to as requirement <R.8>.

The first three challenges are mostly relevant for supporting diverse Smart Devices (<O.1>) and for supporting the creation of diverse orchestration services that implement pervasive computing scenarios (<O.2>, <O.3>).

Service portability in relation to Smart Devices and other services follows directly from the programming abstraction as it is centered around abstract interfaces (Sec. 5.4.1), and as it provides a locator-id-split (Sec. 5.2.13).

Part of this portability is that the abstract interfaces are converged to implement standardization. Sec. 5.6.4 describes such a convergence can be implemented. Sec. 7.4.3 extends the approach from Sec. 5.6.4 for real world usability.

The third aspect, the independence from middleware-specific support functionality becomes more clear in Sec. 6.2.

<R.8> Summarizing the *interface portability* (<R.8>) is provided by the VSL as follows:

1. The VSL implements a type-based locator-id-split (Sec. 5.2.13).
2. As each service needs a VSL context models, each service automatically provides an abstract interfaces (Sec. 5.4.1).
3. The context models are published over the global CMR (Sec. 5.7.1). The CMR becomes part of the S2Store in DS2OS (Sec. 7.2.6).
4. DS2OS provides crowdsourced mechanisms for convergence of contexts in the CMR (Sec. 5.6.4, Sec. 7.4.3).

The combination of the described mechanisms fosters *real world interface portability* of services (<R.8>, Sec. 6.2). *<R.8>*

The fourth aspect of portability, portability in relation to the executables, is implementation specific and not of conceptual nature. It is therefore solved in Sec. 6.3.2 and Sec. 7.2.1.

5.6.4 Support for Collaborative Convergence

As described in Sec. 4.5, to provide service portability in real world deployments, standardization of abstract interfaces is required. If the abstract service interfaces are not standardized, the heterogeneity of the Smart Devices is lifted to a higher level of abstraction only. Service interoperability is not provided.

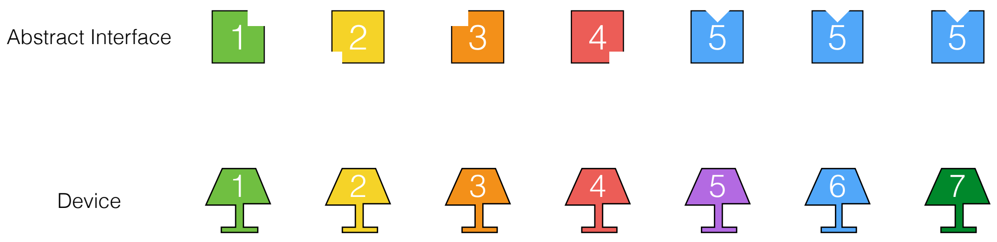


Figure 5.5: Without standardization heterogeneity is only lifted to a higher level of abstraction.

Fig. 5.5 illustrates the problem. Each lamp 1-4 has a different abstract interface. Services that support the lamp 3 cannot use the same abstract interface (labeled “3”) to address lamp 2 for instance. Instead they have to use the interface labeled “2”. As result, a service that is compatible with the abstract interface 3 cannot interact in a Smart Space that only contains lamps of the types 1, 2, 4, 5, 6, or 7.

As discussed in Sec. 3.3.2, classical standardization approaches are not suitable for the convergence of abstract interfaces in Smart Spaces. Reasons include the high amount of interfaces to be standardized and the time it takes to go through a classical standardization process (Fig. 3.2).

The coupling between context models and abstract service interfaces in the VSL programming abstractions provides a solution (<R.10>). To make use of portability, services must be shared between Smart Spaces. To do so, a repository is needed that provides similar functionality for Smart Spaces like the App store for smartphones (<R.46>). The Smart Space equivalent to the App store that is presented in this work as part of the DS2OS framework in Sec. 7.2.6 is called Smart Space Store (S2Store). *<R.10>*

Having the S2Store it becomes possible to correlate the used ModelIDs from the service certificates (Sec. 5.5) with the available context models in the CMR (Sec. 5.2.8). Statistics on how many services use a specific context model can be provided. The design of the S2Store in Sec. 7.2.6 uses the described and several additional metrics to create such a statistic.

Having a statistic of the popularity of a model, it is likely that service developers in a crowdsourced development scenario (<R.48>) use this popularity to decide, which context models they want to support in their service implementation. Most popular context models are likely to be supported in most Smart Spaces in the world. This results in a higher reach for a service as it can run in more Smart Spaces.

In the example in Fig. 5.5, lamp 5-7 use the most popular context model 5. As described above, the assumption for the proposed *collaborative standardization* is that the market will push context model 5 for adaptation services and orchestration services. A service implementation that interacts with the state of a lamp represented in the structure of context model 5 can run in Smart Spaces with lamps of the types 5, 6, or 7.

See Sec. 7.2.6 for an extended *collaborative standardization* strategy that requires more components of DS2OS.

5.6.5 Separation of Logic and State

Memory and imagination are but two words for the same thing.

Leviathan, Thomas Hobbes, British Philosopher, (1588 - 1679).

<R.1> The VSL is a persistent versioning storage for service state. Its use allows to *separate service logic from service state.* This facilitates the creation of services (<R.1>) as developers can focus on the logic of their services using the VSL over its unified API for keeping state [GDH01, GDLM04, DAS01].

The VSL allows *separating service logic from service state.* Service developers do not have to implement functionality to store state. They can create stateless services that use the VSL for managing their own and remote state of other services. The VSL allows *transparent sharing* of state with other services by setting the access rights
<R.1> (Sec. 5.3.3) accordingly (<R.1>). Besides freeing developers from providing functionality for *sharing context*, using the VSL for this purpose implements transparent
<R.49> access control (Sec. 5.3.3) contributing to *security-by-design* (<R.49>).

Externalizing the service state from a service makes its context *independent of the service.* The context that is used by other services is entirely managed by the context repository of the KA. This enables providing service context even when a service is not running while guaranteeing the specified access conditions (Sec. 5.5). Historic context remains available even after a service is shut down. Such functionality is typically not provided by the state of the art solutions where context is coupled to the lifetime of services (Sec. 4.5).

The experience with the VSL prototype shows that the resulting programs are simpler and shorter than programs that have to implement state management (Sec. 5.8). The code complexity and the amount of code are reduced. This lowers the risk for acciden-
<R.49> tal *implementation errors.* It increases the *security* (<R.49>) and the *dependability*

<*R.30*> (<R.30>) of VSL services.

With the described separation of service logic from service state, VSL service implementations become *stateless* as they store all state as context in the VSL (<R.37>). <*R.37*> A stateless service design simplifies service management significantly as described in Sec. 7.2.

5.6.6 Modularization Support

The described four modes for coupling services (Sec. 5.4.3) foster the *modularization of services* (<R.26>). Via *notifications* and *Virtual Nodes* services can be cascaded as shown in Fig. 5.6. <*R.26*>

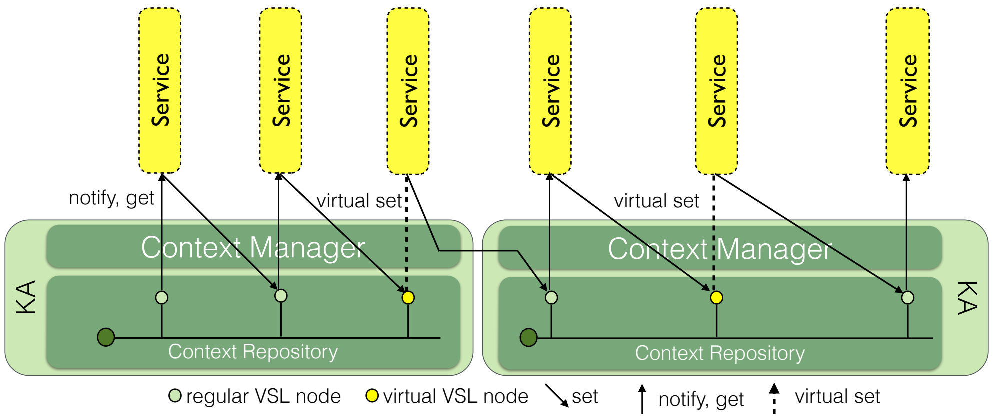


Figure 5.6: Different services are cascaded via subscriptions on regular VSL nodes, and via Virtual Nodes.

The coupling points between the shown services are explicit via the use of the context models. When each service changes the level of abstraction between its input and output, the example service cascade from Fig. 5.6 leads to the availability of *different levels of abstraction* in the VSL context repository.

Via the coupling, the different levels of context abstraction are not only provided but also coupled. Using a bidirectional coupling, changes on any level of abstraction are automatically reflected to the other levels of abstraction via the other services. Each service only has to contain logic to deduce knowledge (context) from one level of abstraction to another.

This is similar to the natural structuring of tasks to solve problems that humans do (Sec. 3.5, <R.25>). The principle is also similar to a protocol stack as it is used in computer networks (Fig. 3.3). For each of the services it is transparent that other services preprocessed the context before. <*R.25*>

Modularization facilitates the creation of complex functionality by creating or reusing services that act as building block for complex functionality. The inherent interface compatibility of all services (<R.6>) enables the *composability of services* (<R.36>). The *sharing of context models* fosters it. See Sec. 5.6.3. <*R.6*> <*R.36*>

As the VSL manages context autonomously (Sec. 5.6.2), it is fully transparent for services that the cascade shown in Fig. 5.6 spans multiple distributed KAs. Sec. 8.3.2 provides a concrete example for service cascading via so-called Advanced Reasoning Services (ARSs).

5.6.7 Emulator Support

Almost all creativity involves purposeful play.

Abraham Harold Maslow (1908 - 1970), American psychologist.

The VSL uses context models as abstract service interface (Sec. 5.4.1). Smart Devices are connected via adaptation services to the VSL (Fig. 5.1, Sec. 8.3.1). This leads to service abstraction, hiding all service details behind the interface (context model, Sec. 5.4.4).

The previously described properties facilitate the development of emulators. It allows to use the original context models / abstract service interfaces for emulating functionality such as Smart Devices (<R.47>). An example is the *office emulator* that is introduced in Sec. 8.3.3. *<R.47>*

As described in the quote above, and in Sec. 3.10, the availability of emulators is relevant for real world software development for Smart Spaces. Trying things out and experiencing the effects of actions are important methods that help humans to learn [PPB$^+$04, Pap05].

In the future, it could become good practice for vendors of Smart Devices to offer an emulator for their devices to enable software developers to create and test software for the device without owning it [EBDN02]. See Sec. 10.4.

5.7 Basic VSL Services

The VSL uses Virtual Context (Sec. 5.3.4) to provide its own functionality.

This section introduces two basic services. The *CMR service* provides the interface of a VSL site to the global CMR (Sec. 5.7.1). The *type search* is used for discovering *context node locators* based on their *type identifier* (Sec. 5.7.2).

The design of the two basic services shows how the VSL meta model can be extended with new semantics dynamically at run time. This is further described in Sec. 5.7.3.

5.7.1 CMR Service

The Context Model Repository (CMR) is the global repository for context models (Sec. 5.2.8). Each service has a context model (Sec. 5.4.1). For instantiating context models, they have to be loaded from the CMR.

The VSL instantiates the context model of a service when the service registers (Sec. 5.3.1). As described in Sec. 5.2.11, the context models are only needed for creating new nodes as VSL context is self-contained (Sec. 5.6.2). The only type that needs to instantiate new nodes at run time is the `list` (Sec. 5.3.2).

To retrieve context models, the *CMR service* is used. It implements a caching proxy for CMR access. The *CMR service* uses the following context model:

```
<model type="/basic/text" reader="*"></model>
```

Listing 5.6: The context model `/system/cmr`.

The proxy registers the root node of the context model as Virtual Node. As the VSL implements distribution transparency (Sec. 5.6.1), all KAs can search for the location of the *CMR service* by searching for a node of type `/system/cmr` (Sec. 5.4.4, Sec. 5.7.2).

To obtain a model, the Virtual Subtree property of Virtual Context is used. It allows to address any node in the subtree after a Virtual Node (Sec. 5.3.4). To load the context model of type `/basic/text`, the following command is executed on a KA:

```
get /agent_mop/cmr/basic/text
```

All address parts after the Virtual Node (`/agent_mop/cmr`) are interpreted as *ModelID* by the *CMR service*. In the above request the parameter is `/basic/text` and the content of the context model is retrieved from the global CMR and returned as value of the Virtual Node.

As the ModelIDs of the context models in the CMR are unique (Sec. 5.2.3), context models can be cached by the *CMR service* and by any KA resulting in desired independence from external services that becomes relevant when a VSL overlay becomes partitioned. See Sec. 5.6.2.

See Sec. A.2 for an Unified Modeling Language (UML) diagram of the described functionality.

5.7.2 Type Search

The *type search* implements the *locator-id split* of the VSL context (Sec. 5.2.13, <R.8>). The *type search* is used for identifying instance *locators* of context nodes to a given type *identifier*. *<R.8>*

The *type search* uses Virtual Context in the same way, the *CMR service* does (Sec. 5.7.1). The only difference is that the *type search* gets mounted in the *knowledge tree* in a special, well known, subtree `/search` under the ID `type`.

The *type search* uses a special context model that is generic for all search provider service. It is shown in listing 5.7. It provides a text that contains a list of instance locators. By intention, it is identical to the `/system/cmr` context model shown in listing 5.6.

VSL types have the dual purpose of identifying the corresponding basic data type and of typing the functional semantics (Sec. 5.2.9). The two types (context models) shown in listing 5.6 and listing 5.7 fulfill the second purpose.

```
<model type="/basic/text" reader="*"></get>
</model>
```

Listing 5.7: The context model `/system/searchProvider`.

To search for the *CMR service* the following command can be issued:

```
get /search/type/?/system/cmr
```

Everything after the Virtual Node `/search/type` is interpreted as parameter. The question mark is used as separator. Before the "?" a *subtree* address can be passed to restrict the search to a certain subtree such as "`/agent5/secrets`". The data after the question mark is interpreted as VSL type identifier. The result of the search for `/system/cmr` is the instance address (`/agent_mop/cmr` in the example from Sec. 5.7.1).

5.7.3 Dynamic Semantic Extension of the VSL Meta Model

Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.

Ludwig Josef Johann Wittgenstein
Austrian-British philosopher, (1889 - 1951)

The VSL meta model that was introduced in Sec. 5.2 allows to express subtyping ("is-a") and composition ("has-a") of *types* in the context model via *multi-inheritance* and *hierarchical addressing*.

The basis of the VSL meta model are key-value pairs (Sec. 5.2.4). In combination with the composition mechanism (Sec. 5.2.11), it is possible to add additional context –which includes additional metadata– to context models. Such additional metadata can express additional semantics, such as the *location* of a context node:

```
<binaryWithGeoLocation type="/derived/binary">
  <myGeoLocation type="/derived/geolocation"></myGeoLocation>
</binaryWithGeoLocation>
```

Via the *multi-inheritance* (Sec. 5.2.11), such additional metadata can be added to all nodes by inheriting from another model such as *hasGeoLocation*:

```
<hasGeoLocation>
  <myGeoLocation type="/derived/geolocation"></myGeoLocation>
</hasGeoLocation>
```

This adds the *myGeoLocation* context to all nodes that inherit from the context model `hasGeoLocation`, e.g.

```
<binaryWithGeoLocation type="hasGeoLocation,/derived/binary">
</binaryWithGeoLocation>
```

Though this way a *geolocation* can be expressed using the VSL meta model, the *geolocation* has no semantics support from the VSL. It cannot be used as *primary context* in semantic queries to identify context nodes (Sec. 5.2.12).

Each service that wants to make use of the *geolocation context* can make a query `get /search/type/derived/geolocation` that results in all *locators* of nodes that have a *geolocation context* in their context model. Then it could select relevant nodes based on the functional type by running a second query for a type and intersecting both results for instance. Though possible, this design introduces complexity in services that the VSL programming abstraction aims to prevent (<O.0>).

The *type search* (Sec. 5.7.2) shows how semantic query functionality can be provided to services by offering a *search provider*. By creating a new search provider `typeAtLocation`, the VSL can be extended with another primary context, location.

Such a search provider could for instance return all node instances of a given type in a certain spatial area, e.g. `get /search/typeAtLocation/livingRoom/?/basic/number` would return all nodes of `/basic/number` in the area that is defined as `livingRoom` in the search provider[26].

As back-end of the location search provider, a cache of the location information from the distributed nodes could be used to make the search faster. The abstract implementation would be as described for the `typeSearch` in Sec. 5.7.2. Details of a concrete implementation are given in Sec. 6.6.4.

Like the *location search provider* different semantics can be added to the VSL making it semantically extensible at run time (<R.28>). *<R.28>*

Implicit vs. Explicit Semantic Extension

Though adding additional context nodes to context models is possible as described, this is impractical for real use extensions of the VSL semantics. It requires changes of the models in the CMR. The versioning of context models in the CMR (Sec. 5.2.8), and the anchoring of context model in service certificates (Sec. 5.5) make this approach difficult to use in real world environments. It would require changing all services that use a context model as only the new version of the model will contain the additional information.

Adding an additional *search provider* without changing the context models is sufficient for extending the VSL semantics implicitly. This is practically feasible in real world deployments as it does not require changes on services. Using the *location search*

[26] `/?/` and `/!/` are used in the example as separators to allow multiple locations or multiple types to be passed, e.g. `get /search/typeAtLocation/livingRoom/!/bathRoom/?/basic/number/!/basic/text`.

provider as example, it is sufficient that it knows locations, e.g. by maintaining a database. The location information does not have to be explicitly present in the context models of entities (services) that have a location.

The *explicit semantic extension* by adding additional context to context models suits better to the self-contained design of the VSL (Sec. 5.6.2). The *implicit semantic extension* suits better to the transparency and the dynamic extensibility. See Sec. 6.6.4.

5.8 VSL Service Elegance

You recognize elegant code when you print it on a T-shirt and it looks good!

Answer to the question what elegant code is, Michael Sperber (*1971), functional programming enthusiast, T-shirt fan, and great teacher, during an introductory lecture to computer science at Universität Tübingen in 2000.

The described service API facilitates the creation of services (<R.1>). Context dissemination is entirely handled in the VSL and does not have to be implemented in services. As practical demonstration of the developer support, the implementation of a VSL service that obtains information from an OS shell, and provides it as context to other services is presented. *<R.1>*

The fixed VSL API was introduced in Sec. 5.3.1. After registering, mainly the `get` and `set` method are needed to implement service logic.

The example shows how the provided *VSL programming abstraction* enables the creation of structured service implementation code that is typically short as the *context management* functionality is provided by the VSL.

A service for reading the amount of currently logged in users from the shell and providing it over the VSL can be implemented as follows:

```java
package org.ds2os.services.loggedInUsers;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.ds2os.connector.Connector;
import org.ds2os.exceptions.VslException;

public class LoggedInUsers {
    private final Connector c;

    public LoggedInUsers() throws VslException, IOException,
        InterruptedException {
        c = new Connector();
        c.registerService("myLoggedInUsersService", "
            loggedInUsersService.crt");
        Pattern pattern = Pattern
                .compile("([:0-9]+)\\s+up\\s+(.*?),\\s+([0-9]+)
                    users?,\\s+load averages?: ([0-9]+\\.[0-9][0-9])
                    ,?\\s+([0-9]+\\.[0-9][0-9]),?\\s
                    +([0-9]+\\.[0-9][0-9])");
        Process p;
        while (true) {
            p = Runtime.getRuntime().exec("uptime");
            p.waitFor();
            BufferedReader in = new BufferedReader(new
                InputStreamReader(p.getInputStream()));
            String line = in.readLine();
            Matcher matcher = pattern.matcher(line);
            matcher.find();
            String users = matcher.group(3);
            c.set(c.getKORSubtree() + "/usersLoggedIn", users);
            Thread.sleep(5000);
29            }
30        }
31
32        public static void main(String[] args) throws VslException,
              IOException, InterruptedException {
33            LoggedInUsers myService = new LoggedInUsers();
34        }
35    }
```

Listing 5.8: Minimalistic implementation of a VSL service that obtains the amount of currently logged in users from the shell and provides it in the VSL (listing of the entire source code).

The corresponding VSL context model that defines the *structure of the context storage* of the service and the *abstract interface to the service* is as follows:

```
<loggedInUsersService>
  <usersLoggedIn reader ="*" type="/derived/usersLoggedIn"</
      usersLoggedIn>
</loggedInUsersService>
```

The star indicates that the value is publicly readable.

The definition of the inherited context model (Sec. 5.2.6) is as follows:

```
<usersLoggedIn type="/basic/number"></usersLoggedIn>
```

The resulting instance of the context in the VSL is:

```
system@agent_mop: /agent_mop/myLoggedInUsersService % get
    usersLoggedIn/*
<usersLoggedIn type="/basic/number" version="75" timeStamp
    ="2014-01-22 20:15:22.741" access="rw">
<![CDATA[3]]>
</usersLoggedIn>
system@agent_mop: /agent_mop/myLoggedInUsersService %
```

The version number 75 indicates, that the value was updated 75 times. A look at the code shows that the implementation updates it every 5 seconds.

The shown code example gives an impression how elegant it is implementing Smart Space services using the presented VSL programming abstraction. Following the initial quote, elegance is used to express that the VSL programming abstraction helps to reduce the lines of code significantly by providing a level of abstraction that simplifies the implementation of pervasive computing services while not introducing limiting constraints.

The example also shows the simplicity of the VSL meta model, the *context access* *<R.29>* *API* (<R.29>), and –as they are the same– the simplicity of the VSL *service API* *<R.1>* (<R.1>). The interaction with the VSL –including the initialization of the connector (Sec. 6.3.3)– needs only four lines of code (14,15,27). The rest of the program is application logic.

The used connector provides the VSL methods as Java methods. Sec. 6.3.3 shows how support for different programming languages can be provided by the VSL implementation.

Ch. 6 describes the back-end functionality of the KAs that enables the coding simplification shown in the listings. Sec. 6.6 and Sec. 8 show more examples for services including services that use Virtual Context.

5.9 Chapter Conclusion

The VSL programming model acts as enabler for software-orchestrated Smart Spaces. It allows the required shift from implementing pervasive computing scenarios to implementing pervasive computing scenarios (<R.1>, Sec. 2.5, [Abo12]).

The *VSL meta model* (Sec. 5.2) combines the expressiveness of logic-based context modeling approachs with the verifiability of markup-based modeling, and the processing efficiency of key-value pairs. In addition it enhances the dependability of services in VSL Smart Spaces as it includes and enables several ways of syntactic and semantic context validation.

Using context models as abstract service interfaces (Sec. 5.4) reduces the complexity of Smart Space Orchestration while increasing the dependability. The complexity is decreased as the descriptive interfaces enforce full encapsulation of services, structure services, and implement inherent interface compatibility. The novel concept Virtual Context (Sec. 5.3.4) enables the use of context models as abstract service interface for all four coordination modes of inter-service communication (see table 5.2).

Separating the API (context access) from the offered functionality of services (context models) makes all services inherently compatible in the function call API. Separating the service logic from the service state (Sec. 5.6.5) facilitates VSL service development by structuring services.

Using the Context Model Repository (CMR) (Sec. 5.2.8) as open global central instance for collecting and disseminating context models implements a dynamically extensible collaborative ontology. It is self-managing as it maintains its integrity by versioning context models, structuring them into namespaces, and automatically validating new context models before storing them.

Using types as primary context implements a locator-id split that enables portability of services (Sec. 5.2.13). The sharing of context models over the CMR, and the possibility to emulate functionality foster the creation of portable services.

The implementation of the VSL programming abstraction as middleware, and its use for all kinds of service interaction enable the implementation of security features as security-by-design (Sec. 5.5). Security-by-design does not rely on the implementation of security features on the service site but implement basic security including authentication and authorization inherently via the programming abstraction.

Together the described parts foster modularization and reuse of services. The examples of the basic VSL services *search* and *CMR service* shows how the introduced VSL programming abstraction enables transparent extensibility at run time with diverse functionality such as additional semantics.

Though most of the presented aspects are new in their interplay, the major novel concepts that were introduced in this chapter are the hybrid meta model, the collaborative self-managing ontology, the dual use of context models as element to structure context and as abstract service interfaces, and Virtual Context as enabler for the dynamic extensibility of the VSL at run time.

As indicated on the overview page at the beginning of the chapter, the VSL provides solutions to almost all of the requirements that were identified in Sec. 3.12. The only open requirements are *portability of the VSL implementation* (<R.7>), *support* *for multiple programming languages* to implement services (<R.27>), *portability of service implementations* (<R.8>), and the *availability of an App store* (<R.46>).

The former three requirements are implementation specific and therefore introduced in Ch. 6. The App store is not directly related to the VSL but to service management that is introduced with DS2OS in chapter 7.

The requirements emerged from the objectives of the thesis. A mapping which requirement supports which objective most is shown in Fig. 3.15.

The *real-world support* (<O.4>) is mostly supported by the *simplicity* of the programming abstraction, its distribution that implements *scalability*, its autonomy that reduces the *complexity*, and the diverse security features that supports *dependability*.

The *diversity of use cases* is mostly supported by the *openness* and the *dynamic extensibility* of the *programming abstraction*.

Not separating between orchestration services (<O.2>) and adaptation services (<O.1>) make the structuring and facilitating elements of the programming abstraction apply to both. The major structuring element of the VSL are the context models that define the abstract service interfaces, and the structure of the state storage of services. Major facilitating elements are the consistent use of concepts at multiple points in the design (e.g. the hierarchical structuring), and the meta model. The autonomous features of the VSL support developers and users in their software-orchestrated Smart Spaces.

All together the overall goal of *designing a programming abstraction for Smart Spaces that structures and facilitates the development of pervasive computing applications at large, and that can be used in the real world* (<O.0>) is fulfilled.

The following two chapters –especially chapter 7– illustrate how the presented programming abstraction can be used to add extending functionality so that all functionality offered by the assessed middleware (Sec. 4.5) is supported such as service management.

6. The Virtual State Layer μ-Middleware

The paper claims that the VSL/DS2OS approach provides ”similar or better support for scenarios other middleware was especially designed for.” This is a bold claim that is not at all substantiated. Were it true, this would be revolutionary.

Anonymous reviewer, level “expert”, 2013, to a paper handed in for the conference PerCom2014.

Short Summary The chapter introduces the μ-middleware concept.
An implementation of the Virtual State Layer (VSL) programming abstraction as distributed, self-managing Peer-to-Peer (P2P) system is presented.
Programming language specific connectors are introduced as concept for supporting developers.
Implementation details of the basic services that were introduced in Sec. 5.7 are given for illustrating the advantages of the μ-middleware design.

Key Results The VSL programming abstraction implements a novel type of pervasive computing middleware that is called μ-middleware. The VSL implementation is fully distributed and autonomous. The combination of distribution and autonomy leads to scalability. The autonomy enhances the dependability and the security of the VSL implementation. High interoperability of the VSL is reached by using standardized technology for implementing the communication and security features. The usability of the implementation is enhanced by providing programming language specific connectors.

Key Contributions The concept *μ-middleware* is introduced. It enables the seamless integration of functionality into a middleware, resulting in dynamic extensibility at run time.

Challenges Addressed <R.3> *Self-management*, <R.4> *User participation*, <R.8> *Logical portability of services*, <R.7> *Portability VSL implementation*, <R.11> *Distribution support*, <R.23> *Dynamic extensibility*, <R.27> *Support of multiple programming languages*, <R.30> *Dependability*, <R.45> *Portable service executables*, <R.48> *Crowdsourced development*, <R.49> *Security-by-design*, <R.50> *Scalability*

Summary The chapter starts with the introduction of the μ-middleware concept that emerges from the VSL programming abstraction. The term μ-middleware is used to describe that only fundamental, non domain-specific functionality is provided in a middleware core that can be transparently extended via services at run time.
The VSL μ-middleware is implemented in Java as system of distributed Knowledge Agent (KA) peers. Using Java, context models as abstract service interface, and implementing a μ-middleware, results in portability. It is supported by the openness of the implementation resulting from the use of standardized communication methods.
The VSL implementation is designed modular, which allows exchanging components for supporting different transport mechanisms, or storage mechanisms. It applies the separation of service logic from service state that was proposed in Sec. 5.6.5.
The VSL is implemented as fully autonomous system. It manages the connections to services, and provides full context management including context routing and context discovery between distributed KA peers.
Using X.509 certificates, encryption, and the access control from Sec. 5.3.3, the VSL implementation provides diverse security-by-design elements.
The implementation details of the basic services from Sec. 5.7 illustrate the μ-middleware properties by demonstrating dynamic transparent extension at run time. In addition they show how the VSL programming abstraction shortens, structures, and facilitates the implementation of VSL services.

6.1 Introduction

Ch. 5 introduced the Virtual State Layer (VSL) programming abstraction. Though examples from the following Java-based prototype implementation are used to illustrate the concepts, the VSL is designed independent of a specific implementation. The VSL programming abstraction provides basic context management. Additional functionality can be added at run time. The VSL can be implemented and used as a distributed system or as a centralized system.

This chapter introduces the prototype implementation of the VSL as self-managing Peer-to-Peer (P2P) system. The implementation shows the feasibility of the concept. It is used for the evaluation of the suitability of the VSL as programming abstraction for Smart Spaces in Ch. 9.

Following, an implementation for the VSL concepts from Ch. 5 is introduced. As the VSL programming abstraction only provides basic functionality, the resulting VSL middleware provides less functionality than several of the assessed middleware solutions from Sec. 4.5. However, diverse functionality including the features provided by the assessed middleware can be dynamically added at run time via Virtual Context as described in Sec. 5.3.4. Sec. 7 describes such add-ons. To describe the resulting concept of an extensible core the term *μ-middleware* is introduced.

6.2 μ-Middleware

La perfection est atteinte non quand il ne reste rien à ajouter, mais quand il ne reste rien à enlever.

Antoine-Marie-Roger de Saint-Exupery (1900-1944), French aristocrat, writer, poet, and pioneering aviator.

As discussed in Sec. 4.5 and summarized in Sec. 4.5.18 and table 4.1, a fundamental problem of existing pervasive computing middleware is that it offers only a fixed functionality.

Some of the assessed middleware (Sec. 4.5) provides support functionality for specific workflows. Such support functionality can be used as building block by developers to implement services in the corresponding domain. Examples for such functionality are the ACE middleware add-on (Sec. 4.5.14) or the Event-Condition-Action (ECA)-rule-based development of services of OPEN (Sec. 4.5.12). Workflow-specific support functionality is important to facilitate the development of services (<O.4>).

It is problematic to provide support functionality within a middleware. While supporting the workflows it was designed for, specific functionality can impede the implementation of other workflows. The least impediment is imposing preferences to developers by facilitating certain implementations. This can influence the way, developers implement functionality, restricting the creative process of software development since the diversity of pervasive computing requires diverse solutions (Sec. 2.5, [DM12, Hin99]). In the worst case, a given middleware functionality prevents the implementation of certain pervasive computing scenarios as their required functionality cannot be implemented with the given middleware base. Typically existing pervasive computing middleware complicates the implementation of workflows, it was not designed for, as designated mechanisms of the middleware have to be circumvented.

Other assessed middleware does not provide supportive building blocks (e.g. BOSS, Sec. 4.5.7). Leaving the implementation of workflow-specific functionality entirely to orchestration service developers (Sec. 8.3) complicates the service development. In addition, it leads to other problems such as redundancy since all services that need a certain functionality have to implement it (Sec. 3.3).

Pervasive computing middleware provides an abstraction on top of the Smart Device hardware in Smart Spaces. It implements interface portability of services by making them independent of concrete hardware instances (<R.8>). An operating system provides an abstraction on top of the peripheral hardware of a computer (<R.8>) and a run time environment (<R.45>).

Typical pervasive computing middleware in 2014 can be compared to early operating systems that were monolithic. As described in Sec. 3.4.4, a monolithic operating system has all functionality statically compiled into the kernel. It cannot be changed or extended at run time. Only the fixed kernel functionality can be used to implement services.

In the early 1980s μ-kernels emerged. They provide a minimum amount of functionality in the Operating System (OS) core, the kernel, and can be extended by services outside the kernel [RR81, ABB^{+}86]. Among other advantages, the μ-kernel approach allows dynamic reconfiguration of an OS (e.g. device drivers, new file systems) via modular extensibility according to the needs of an application [GG92].

The VSL design enables the transformation of pervasive computing middleware from a monolithic core architecture to a modular system. In reminiscence to the development of operating systems, following the idea that pervasive computing is the next generation of computing (Sec. 2.5), the term *μ-middleware* is introduced to describe the new architecture.

Definition ***μ-middleware***

> A μ-middleware is a middleware that provides only fundamental, non domain-specific functionality, and that supports transparent extension with functionality at run time via regular services.

As described in Sec. 2.6.4, a fundamental task when implementing pervasive computing services is acquiring and processing context. Managing context is done by context-provisioning middleware. Therefore the VSL is a context provisioning μ-middleware.

To enable dynamic extension at run time, the VSL uses *Virtual Context* (Sec. 5.3.4). Virtual Context can be accessed transparently like regular context but it is served dynamically via callbacks into services and not via context repositories.

Virtual Context is accessed transparently by other services. Services that expose Virtual Context can be connected to the VSL at run time. Both properties together implement the transparency and dynamic extensibility of the VSL at run time which makes it a *μ-middleware* according to the definition given above.

The VSL's described *μ-middleware* properties enable adding support functionality, which is statically provided in other pervasive computing middleware designs, dynamically at run time. The VSL can be extended by adding services that implement the desired functionality. By using unified interfaces to all services, the VSL integrates such extensions seamlessly, independent if they provide "core functionality" or not (Sec. 5.7.3). The basic services (Sec. 6.6) confirm this. They implement "core functionality" such as the *type search*, and they are implemented as VSL services.

Different to the existing middleware with statically built-in support functionality, support functionality from different application domains can dynamically be added to the VSL. This makes the VSL a middleware that can be used in different domains which leads to function compatibility between the domains.

Sec. 5.6.4 discussed the problem of lifting heterogeneity from the device layer to a higher level of abstraction based on abstract service interfaces. A solution is the introduction of crowdsourced convergence (Sec. 5.6.4).

Portability has multiple dimensions as described in Sec. 5.6.3. The above described dimension of portability decouples services logically from device-specific interfaces (<R.8>). When using a middleware this first aspect of portability includes offering a middleware interface that is identical and usable in heterogeneous environments. A second aspect is enabling the execution of service executables on heterogeneous hardware (<R.45>, Sec. 6.3.1). Existing pervasive computing middleware typically solves the first aspect in its scope. The second aspect can be solved with existing technology such as Java (Sec. 6.3.1, Sec. 3.10).

Having the challenges of implementing portability solved in the dimensions described before, another aspect of portability emerges. The heterogeneity problem occurs again on the next level of abstraction: between different middleware for different use cases. Different pervasive computing middleware is typically incompatible (Sec. 4.5). So-called *middleware silos* exist.

Many existing pervasive computing middleware is designed for a specific use case (Sec. 4.5.2). Services that run on one middleware are not portable to another middleware. The different interfaces, and the missing support functionality prevent services from running on other middleware, preventing interaction with services that run on other middleware. Such interaction between scenarios is required to foster the implementation of divers pervasive computing scenarios (<O.3>).

μ-middleware resolves the problems that were identified and described by the term middleware silo. A μ-middleware design comprises a common core that can dynamically be extended with domain-specific functionality. Sharing the same μ-middleware core, enables interoperability of different pervasive computing domains and scenarios. Services from all scenarios can communicate over the common μ-middleware.

6.3 System Architecture

In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.

Grace Hopper, American computer scientist, (1906-1992)

The VSL is implemented as Peer-to-Peer (P2P) system. It consists of distributed peers that are called *Knowledge Agents (KAs)*. The Knowledge Agents (KAs) peers autonomously provide and manage the context repositories and the other VSL mechanisms that were introduced in the last chapter. Fig. 6.1 maps the concepts from Fig. 5.3 to a system architecture.

The figure shows four computing hosts in a VSL site. Each host runs a KA (1). The design as distributed system of peers results in scalability of the VSL implementation (<R.50>). The performance of the system can be enhanced by adding more computing nodes that host KA peers. As described in Sec. 6.4.4, by running services on different hosts, the VSL context is automatically distributed among the peers. If different services access different context on different KAs, the load gets automatically balanced between the hosts . *<R.50>*

Different services (yellow) are running as clients logically on top of the KAs (green). A classification of different service categories is given in Sec. 8. A special service is the CMR service (2) that provides the connectivity of the VSL site with the Context Model Repository (CMR) (Sec. 5.7.1, Sec. 6.6.1).

The *Context Model Repository (CMR)* (3, Sec. 5.2.8) is provided on a central server outside the VSL site. It stores the context models and implements the ontology of VSL Smart Spaces (Sec. 5.2.18, Sec. 6.6.1).

The following implementation uses the Internet Protocol (IP) as communication protocol between the KAs. This choice fits to the described development of Smart Devices towards standard Personal Computer (PC) designs ("all IP", Sec. 3.2.4).

6.3.1 Portability of VSL and Services

Smart Spaces consist of heterogeneous computing environments (Sec. 3.2.4). Portability of the middleware is required (<R.7>) to allow context exchange between heterogeneous hosts that run instances of the middleware. Portability includes two aspects here. First it means that the implementation of the VSL is running on heterogeneous computing nodes with different OSs as described here. Second it means that the VSL implementation uses standard interfaces so that it can be exchanged without having to change services (Sec. 6.3.2).

Portability of services is required if services should run on different hosts (<R.45>, Sec 5.6.3). Computing hosts typically run an OS that provides an abstract interface to the basic hardware such as Central Processing Unit (CPU) or memory [Tan01]. It is the run time environment for application software. Though it is possible to use services that run only in a specific environment (e.g. Unix OS) for orchestrating Smart Spaces, sharing services between different hosting environments requires services in a format that allows to run them on different platforms including embedded systems with diverse hardware architectures and OSs.

The Java programming language [GM95] provides an environment that allows to run the same service executable on heterogeneous hardware. Java programs are compiled into Java bytecode that is interpreted for execution. The use of an intermediate platform-independent format for executables results in portability. System-specific run time environments, called Java Virtual Machine (JVM), are available for diverse platforms that execute the bytecode.

The goals of the Java programming language fit to the goal to foster user-based collaborative, crowdsourced, development of services for Smart Spaces. Relevant properties are the following ones [GM95]:

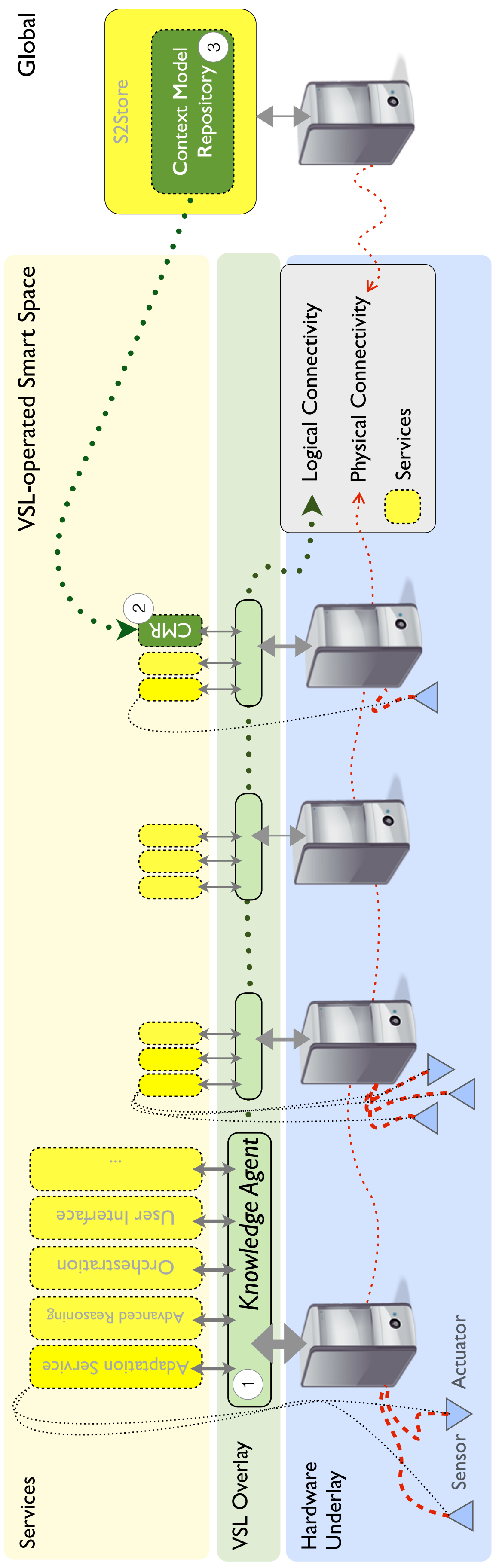


Figure 6.1: System view on the Virtual State Layer.

- *Simple, Object-Oriented, and Familiar*: Java is designed to support programming beginners (<R.4>). *<R.4>*
- *Robust and Secure*: Java validates software at compile and run time. It provides automated memory management. This lowers the error potential during development, and it protects the hosts, Java applications are executed on, at run time (<R.49>). *<R.49>*
- *Architecture Neutral and Portable*: Java is designed to make programs executable with the same bytecode on different hardware platforms (<R.7>, <R.45>). *<R.7>* *<R.45>*
- *High Performance*: Though it is interpreting code to run it, Java aims to provide high speed. For further optimization it is possible to translate Java bytecode into native code [HCJG97, HGH96]. By using Java, the described feature allows optimizing VSL service performance only by recompiling them. This makes the use of Java for implementing VSL services flexible and performant (<R.50>). *<R.50>*
- *Interpreted, Threaded, and Dynamic*: The interpretation allows fast development and debugging, multi-threading support allows to make use of available hardware resources (e.g. CPU, RAM), and the dynamic binding of libraries supports flexible extensions (<R.3> <R.23>). *<R.3>* *<R.23>*

In addition, existing libraries for many applications such as interfacing the protocol stacks of Building Automation System (BAS) are available[27]. This facilitates the development of VSL adaptation services.

For the discussed reasons Java is used for implementing the VSL and for implementing VSL services in the prototype implementation described in this thesis.

Though Java is chosen for the prototype, the programming language neutral design of the VSL interface (Sec. 6.3.2) enables the interfacing of the KAs by services via different programming languages (Sec. 6.3.3).

6.3.2 KA Service Interface

The interface of the KAs is implemented via Extensible Markup Language (XML)-Remote Procedure Call (RPC). XML-RPC uses XML over Hyper Text Transfer Protocol (HTTP) for communication. The resulting interface is OS and programming language independent. It can be used to exchange information between KA and service implementations in different programming languages.

Calling the *get* method (Sec. 5.3.1) of a KA via XML-RPC can be done as follows:

```xml
<?xml version="1.0"?>
<methodCall>
  <methodName>get</methodName>
  <params>
    <param>
        <value>
          <string>/agent_mop/mySmartDevice/ledRed/value</string>
        </value>
    </param>
  </params>
</methodCall>
```

[27]An example for a library that interfaces a network management protocol (Sec. 4.4) is http://www.snmp4j.org/ that interfaces the Simple Network Management Protocol (SNMP).

Following the example from Sec. 5.3.2 a possible answer is:

```
<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
        <value><string>0</string></value>
    </param>
  </params>
</methodResponse>
```

By using the described programming language independent interface, VSL services can be implemented in different programming languages such as Java or Python (<R.27>). This is relevant for supporting different existing programming communities for enabling users of one programming language to use it with the VSL. In addition, different programming languages suit better to solve certain problems. *<R.27>*

The use of XML-RPC in combination with the fixed API (Sec. 5.3) and the sharing of context models over the CMR make the VSL μ-middleware an open system, combining interoperability, portability, and the use of open communication protocols. Openness supports portability of services (<R.8>, <R.45>). *<R.8>*

The VSL interface consists of 13 fixed functions (Sec. 5.3.1). XML-RPC is used as standardized mechanism. Both enable the automatic generation of connectors from the VSL interface definition (e.g. in an Interface Definition Language (IDL), Sec. 5.4.1, Sec. 3.4). This enhances the usability significantly, and it reduces coding errors in the implementation of connectors as it is done automatically. *<R.45>*

The implementation of the described automatic generation of connector code is future work.

6.3.3 Programming Language-Specific VSL Connectors

Though the use of XML-RPC (Sec. 6.3.2) allows to communicate with a KA from different programming languages, using native interfaces in a programming language facilitates the development. Providing native interfaces to the KAs in different programming languages is done by so-called *VSL connectors*.

A *VSL connector* provides a native interface to a programming language and communicates with the KA using the described XML-RPC mechanism (Sec. 6.3.2). By offering VSL connectors to different programming languages programmers can flexibly choose their programming language to implement VSL services.

Programming is an activity that is close to human thinking [JL93] (Sec. 3.5). It is easy for humans to learn programming when using the right tools [MHP00, Ahl76, Can76, Gol76, KG77, Cho56, Kay96, Pap05]. The chosen approach, to use connectors, makes it possible to offer native support for diverse programming languages including simple to learn languages such as visual programming languages [WHI97, Kay90, Kay96] (<R.27>, <R.4>). Only the connectors not the KAs have to be adapted to the programming language, a service developer wants to use. *<R.27>* *<R.4>*

A wide choice of programming languages is also relevant for advanced programmers to support the diversity of pervasive computing scenarios to be implemented. Different programming languages fit better for solving different problems [MHP00, KG77, Abo12] (<O.3>).

The prototypical implementation provides a Java connector that offers the following abstract interface for service programmers:

```
public interface IConnector {
    String get(String address);
    void set(String address, String value);

    void subscribe(String address, ISubscriber subscriber);
    void unsubscribe(String address, ISubscriber subscriber);

    void startTransaction(String address, ITransactionAbortHandler);
    void commitTransaction(String address);
    void abortTransaction(String address);

    String registerService(String localName, String
        serviceCertificate);
    void unregisterService();
    String addCertificate(String shortName, String
        additionalCertificate);
    void removeCertificate(String shortName);


    String registerVirtualNode(String address, IVirtualNodeHandler
        virtualNodeHandler);
    String unregisterVirtualNode(String address);

    String getKORSubtree();
    String getModel(String modelID);
}
```

Listing 6.1: Abstract interface of the Java connector. All methods throw exceptions that are not shown for better readability.

Listing 6.1 shows a direct mapping of the methods from the VSL Application Programming Interface (API) (Sec. 5.3.1) to Java methods. The commands `getKORSubtree()` and `getModel()` are additional support methods ("syntactic sugar") of the connector that result in the previously described VSL calls for implementing the functionality. The `getKORSubtree()` method returns the subtree that is assigned to the service on registration. The `getModel()` method can be used to retrieve context models from the CMR (Sec. 6.6.1) [28].

Via the connectors also other protocols can be interfaced. e.g. connections via XMPP [SA11] to KAs can be implemented. Though the fixed methods of the VSL API facilitate such interfacing of other protocols it was not done yet and is future work. For real world use security of such protocols must be taken into account.

6.3.4 Transports

The VSL implementation needs two classes of messages to be exchanged between the KAs to implement a VSL overlay. *Multicast messages* are used to synchronize the distributed KAs (Sec. 6.4.9). *Unicast messages* are used to exchange context between specific KAs (Sec. 6.4.6).

The prototype contains a *transport* class that handles the message exchange. This makes the transport exchangeable to use different underlying protocols (<R.7>). The *<R.7>*

[28]This can be interesting for services that load additional parameters via other context models than their own by using the CMR as central context storage for instance.

current implementation uses plain XML messages that are sent via Transmission Control Protocol (TCP) stream for the unicast communication and via User Datagram Protocol (UDP) IP multicast for the messages that should reach all KAs.

The use of standardized communication protocols provides compatibility between different VSL implementations as another aspect of portability (Sec. 5.6.3, <R.7>). *<R.7>*

6.3.5 Database Back-End

As described in Sec. 5.2.1, VSL context consists of tuples with fixed metadata. Each tuple in a context repository on a KA is self-contained as relationships to other context nodes are only logically introduced via the hierarchical addressing (Sec. 5.2.2), and via the VSL types (Sec. 5.2.9). All dependencies are resolved during instantiation (Sec. 5.2.11).

To implement persistence, VSL tuples are stored in a database. The tuple address is used as key to access the database. The KA connects to the database via the abstract Java Database Connectivity (JDBC) interface. The use of an abstract interface allows to exchange the database for supporting heterogeneous systems (<R.7>). *<R.7>* The prototype uses HyperSQL DataBase[29] (HSQLDB) as database implementation. Depending on the resources that are to be used on a computing host different implementations for storing the context tuples can be optimal.

Using a database as back-end automatically provides *scalability*, *Atomic Consistent Isolated Durable (ACID)* (Sec. 5.3.2), *concurrency transparency*, and multi-platform support for persisting VSL tuples [BD99].

Each KA uses an isolated database as storage back-end. The VSL implements a distributed database via the distributed KAs. A difference to existing distributed databases is the connection of the database back-end with the VSL meta model. Different to a database, the VSL supports hierarchical structuring, inheritance, validation, and the dynamic extension and use of data types at run time as described in Sec. 5.2. Virtual Context is a concept that is typically not found in existing database implementations.

6.3.6 Separation of Logic and State

Sec. 5.6.5 proposed a separation of service logic from service state for structuring the service development. Such separation is applied in the implementation of the KA peers. State of node-local services is stored directly in the local context repository.

The experience with the design and the implementation of the KA identified the following (subjective) advantages in developing software (<R.1>): *<R.1>*

- Using the local context repository for storing state unifies the access to state as the fixed VSL API is used. This facilitates the context access.
- Using a context repository outside a service implementation facilitates the sharing of context between different service implementation modules for modular services. In addition, it allows to couple distributed services in a structured way.

[29]http://hsqldb.org/

- The state of a service is made explicit via using a context model. This structures the software design as it makes the process of creating data structures explicit.
- Having most state at a central place (context repository) that can be accessed from outside the running application allows to monitor and debug an application.

All four aspects are desired effects of applying the VSL programming abstraction. They facilitate the development of services using the VSL programming abstraction *<R.1>* significantly (<R.1>). The resulting support for developers that is implemented by the VSL programing abstraction fosters user participation in the development process *<R.4>* (<R.4>).

The described positive effects are confirmed by the outcome of the user study in Sec. 9.6.

6.4 Self-Organization

<R.3> The VSL provides its functionality fully autonomously (<R.3>). This is necessary as

- Smart Devices in Smart Spaces are typically *unattended* (Sec. 2.4, Sec. 3.2) [Pos11],
- typical Smart Spaces will *not* be maintained by *experts* [BLM$^+$11, MH12, Dou03, TPR$^+$12], and
- the *complexity* of Smart Spaces is difficult to manage by humans [Kru09, Pos11, CD12, Abo12].

This section introduces different mechanisms of the KAs that implement autonomy.

6.4.1 Service Registration

Services must register at a KA to access the VSL via the API (Sec. 5.3.1, Sec. 6.3.3).

The service registration consists of the following steps based on the service certificate (Sec. 5.5.1, Sec. 6.5) that a service presents:

1. The service certificate is validated using the public key of the VSL site that is part of the KA's own certificate.
2. The *serviceID* is generated by concatenating the *name* from the certificate and the *local name* that is passed with the registration method.
3. The *registration binding record* is created as described below.
4. If the local context repository contains context under the *serviceID* it is used. Connecting a service to its existing context implements persistent state over the run time of a service. If no context exists the context model is instantiated as follows:
   (a) The context model that is identified by the ModelID which is part of the service certificate is loaded from the CMR (Sec. 5.2.8) via the CMR service (Sec. 5.7.1).

(b) The integrity of the context model is validated using the hash from the certificate. This is important as the context model contains the access rights for external services.

(c) If valid, the model is locally instantiated under the address `/agentID/serviceID`.

Registration Binding

If the service certificate that a registering service presents is valid, the KA stores a mapping as shown in Fig. 6.1.

Service ID	readerIDs	writerIDs	Transport binding	public key	expiration time

Table 6.1: The binding record of a service in a KA.

The data from the mapping is taken from the certificate (Sec. 5.5.1):

- The *serviceID* is the concatenation of the *service name* with the *local name*.
- The *readerIDs* and *writerIDs* are taken from the *certificate*.
- The *transport binding* contains the IP address, and the port number that are used to communicate with the service.
- The *public key* is the one from the certificate.
- The *expiration time* is taken from the certificate.

The *readerIDs* are used for `get` requests and the *writerIDs* for `set` requests. See Sec. 6.5 and Sec. 5.5.

The KA checks periodically if service binding records are expired and closes the connections of those services that have expired bindings. As the service certificate is typically invalid at that time, a service cannot re-register until it has a new valid certificate (Sec. 7.2).

6.4.2 Context Storage

The KAs use a tuple storage in form of a database as persistent storage back-end (Sec. 6.3.5). The primary key to access the storage is the context node address (Sec. 5.2.2).

Each KA uses its own storage back-end. As the use of the JDBC interface allows to exchange the storage back-end by using the same interface, it is possible to run KAs with different storage back-ends within the same VSL overlay.

The logical structure of context is independent of its storage since it follows the VSL meta model (Sec. 5.2). Its self-containment allows to use different storage back-ends such as plain files in directories.

The implemented locator-id split (Sec. 5.2.13) uses type identifiers for identifying context tuples. This requires having a fast search for types. In the presented VSL prototype the context database is used for executing the type search.

If the used storage back-end is slow, it makes sense to use a second data structure (e.g. a hash map) only for the type search for speed up. This was not done in the prototype as using a single data structure for storing all VSL tuple-related information reduces the complexity of the KA implementation. Low complexity is desired.

The context access happens via the methods of the VSL API (Sec. 5.3.1). See Fig. 6.2.

6.4.3 Virtual Context Node Handling

Requests to Virtual Context nodes result in callbacks into services (Sec. 5.3.4). Virtual Nodes can be registered from a service using the API (Sec. 5.3.1, Sec. 6.3.3).

As described in Sec. 6.3.6, the KAs use their context repositories for storing their own state. The mapping between a service and its Virtual Nodes is maintained via adding the following context to a VSL node:

```
<serviceId type="/system/serviceId,/basic/text" version="1"
    timeStamp="2014-02-19 10:51:18.251" access="rw">
<![CDATA[/agent_mop/client:cmr]]>
</serviceId>
```

Listing 6.2: Virtual Node callback binding in a VSL context node.

The listing shows the mapping of the CMR service (Sec. 5.7.1, Sec. 6.6.1). The mapping is indirect over the local service Identifier (ID). The communication happens via XML-RPC to the TCP port, a service is registered with. It is resolved via the *service binding record* (Fig. 6.1, Sec. 6.4.1).

The indirect mapping is used to decouple the registration binding (TCP port) from the routing entry in the VSL (agent ID). When a service registers again (e.g. after failure), only the registration binding has to be changed (Sec. 6.4.1). The information in the Virtual Nodes remains valid.

The context access happens transparently via the methods of the VSL API (Sec. 5.3.1) as described in Sec. 5.3.4. See Fig. 6.2.

The VSL provides the entire routing for inter-process communication. In addition it can be used to provide basic access control (Sec. 5.5). This simplifies the development of VSL services (<R.1>) as the described functionality does not have to be handled in the service implementation. *<R.1>*

6.4.4 Context Distribution

The context of a service is always stored at the KA it is connected to. The *local context principle* is applied to optimize the resulting context access pattern that emerges from the separation of service state from service logic.

The separation of service state from service logic (Sec. 5.6.5) results in frequent access from stateful services to their state. Such access can typically be handled best locally as the direct connection between a service and its KA implements the shortest latency and the highest bandwidth compared to connections with more hops (e.g. to other KAs, Sec. 9.3). This is in particular the case when a service and its KA run on the same machine which is the intended use case (Sec. 7.2).

The performance of the context access is better on a local KA than on a remote KA. The evaluation shows that an additional hop adds about 30ms additional latency in the test setup (Sec. 9.3.4). In heterogeneous and slow networks the negative impact will be stronger (Sec. 3.2.4). In such cases distributing the same context on multiple KAs would have positive impact on the performance of the context access and the dependability of the VSL. But it would have negative impact on the complexity of the system as it would require synchronization of distributed instances of the same context nodes.

The assumption that the backbone of future Smart Spaces will be an IP domain (Sec. 6.3) in combination with the distributed context discovery (Sec. 6.4.6) leads to a maximum distance of one KA for accessing any context inside a VSL site. Requested context is either stored directly on the KA, a service is registered to, or its KA can directly reach the KA with the context.

The maximum distance to access context is one. As a result, additional latency and possible bandwidth limitations only apply once. When context requests would be routed through multiple KAs, latency and bandwidth limitation would affect the context exchange multiple times. Therefore context caching is not implemented in the described KA prototype. Sec. 6.4.5 discusses context caching.

Context is discovered locally using the shared directories that contain the structure of the entire VSL knowledge base (Sec. 6.4.6). The KA that provides the context is directly contacted. See Fig. 6.2 for an example with two services connected to two KAs.

In case, both services are connected to the same KA, context is available locally and no second KA is involved. The resulting sequence diagram is similar to that shown in Fig. 6.2. Only *agent1* can be removed and the arrow marking the request from *service1* goes directly to *agent2*. The reply arrow goes directly to *service1*.

Virtual Context must always be served via the KA that maintains the mapping to the service callback (Sec. 5.3.4, Sec. 6.3.6). This is shown in the lower part of Fig. 6.2.

6.4.5 Context Caching

Each context node version in the VSL has a Globally Unique IDentifier (GUID). It consists of the elements `/VslSiteId/agentId/serviceId/contextNodeAddress-es/versionNr`. This allows to cache VSL context transparently.

The design of the VSL with self-contained tuples (Sec. 5.2) facilitates caching. All metadata including the access permissions would automatically be synchronized and applied without changes on the existing implementation (<R.49>). *<R.49>*

A KA could cache locally accessed context nodes making their retrieval for local services faster. In addition the availability of context copies could be published regularly as described in Sec. 6.4.6.

Context caching would enhance the resilience of the VSL overlay as context could be replicated and remain available not only on service failure but also on failure of KAs.

The existing VSL prototype implementation allows adding the described caching without major changes. However, this was not done yet for context consistency reasons. Context caching could lead to the situation that services retrieve cached context while

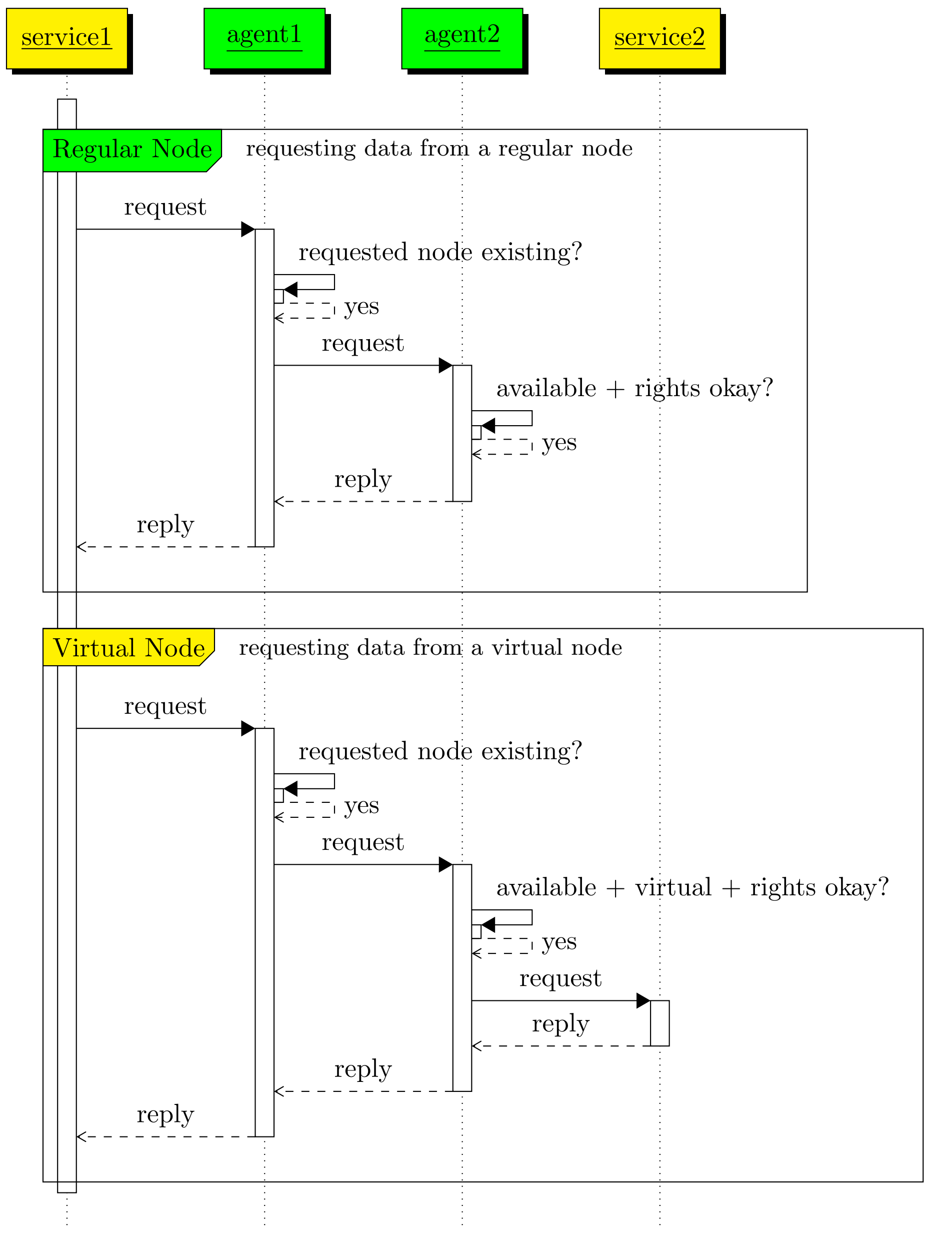


Figure 6.2: Requesting VSL context over regular and virtual nodes.

newer context is available. Such a situation is transparent for a context requester as the cached copy is identical with the original context node and requested with the same method. As a result, the requester cannot know that newer context is available.

As each context node contains a timestamp in its context meta data, a cached copy is valid. However, it possibly contains not the most recent context. The requester cannot know when the last value update on the context happened. It could assume (wrongly) that a retrieved cached context is the newest version. Additional markers for cached copies could be a solution for this problem. Context caching is a topic of future research.

Context caching could enhance the resilience of the VSL further and is a topic for future development.

6.4.6 Context Discovery

Context can be accessed using the hierarchical addresses introduced in Sec. 5.2.2 as locators. For implementing interface portability (<R.8>) the locators can be resolved via the *primary context* (Sec. 5.2.12), the *VSL type* as described in Sec. 5.2.13.

The distribution (context storage) and lookup (context discovery) of data are fundamental challenges of P2P systems [RFH$^+$01, GEBF04, KLL$^+$97].

Distributed Hash Table

A typical solution for data distribution and lookup in a P2P system is a Distributed Hash Table (DHT). A Distributed Hash Table (DHT) distributes data to peers based on a hash that is typically built over the meta data of the data object that should be stored (e.g. name) [RFH$^+$01, GEBF04, KLL$^+$97]. In case of the VSL suitable metadata would be the type as it is used for searching context. Knowing the meta data, peers can apply the hash function to determine the peer that would contain the data. This peer (or multiple peers) can be contacted then to ask for the existence of data and their retrieval.

The DHT principle is most efficient when the connections between the peers have low latency. The computing infrastructure of future Smart Spaces is heterogeneous (Sec. 3.3.1). This includes the computing resources and the network connectivity. As a result, retrieving data from KAs that have a network connection with high delay or high error rate will be slow. In a typical DHT with equal distribution of data between peers such resource-based criteria would not be taken into account.

As described in Sec. 5.3 and Sec. 5.4 the VSL programming model fosters a separation of application logic and application state when implementing services. As a result services are likely to access their state (context) frequently. The use of a DHT would distribute a service's data on distributed KA. This introduces additional latency. Assuming a uniform distribution of context and requests, the KA with the worst connectivity to the VSL impact the expected latency of context access by a service which is undesired as it slows services down.

The combination of a heterogeneous infrastructure that cannot guarantee fast data exchange between peers, and the frequent local access to certain context by services make a DHT unsuitable as distribution and discovery mechanism for the VSL.

Shared Directories

Instead of implementing a DHT, the described VSL implementation uses the *locality principle* of storing all context only directly at the KA a service is connected to. This is described in Sec. 6.4.4 and Sec. 6.4.5.

The discovery of context in a VSL site happens based on the VSL types (Sec. 5.2.13). A possibility for implementing context discovery would be to query all KAs of a site. For the described heterogeneity reasons (see above), this approach is not chosen.

Instead, all KAs synchronize the node structure of their context repositories periodically (Sec. 6.4.7). The context values are not synchronized for the reasons discussed in Sec. 6.4.5, and for scalability reasons as the context values may be large (<R.50>). *<R.50>* The access permissions are synchronized as the search uses them to return only results a service can access with its current accessIDs (Sec. 5.5). As shown in Fig. 6.2 access control is handled at the KA that holds the requested context values. For facilitating the use (<R.1>), *<R.1>* and for securing the information which context is available (e.g. as anti-burglar-service, <R.49>) *<R.49>* the search is designed as described not to reveal context nodes that cannot be accessed.

As all KAs have the structure of the entire VSL site locally via the described synchronization (Sec. 6.4.7), context discovery happens locally via lookup in the local context repository. This makes the frequent operation of context search fast (<R.50>). *<R.50>*

The type search is implemented using Virtual Context as shown in Sec. 6.6.3.

6.4.7 Context Repository Structure Synchronization

For discovering each other and for synchronizing the context repository node structures of the distributed KA peers, periodical *alive pings* are exchanged between KAs using the multicast transport (Sec. 6.3.4).

An *alive ping* contains an ID that identifies the current version of the context repository of the sender. In addition, it contains all updates on the context node structure that were done in the context repository of the sender since the last *alive ping* was sent.

A structure update is adding or removing context nodes. It is not exchanging context values. Context values are only stored at the KA, a service is connected do. The receivers of an *alive ping* store the context node structure of the remote KA. The local copy of the sender's context repository structure is called *remote Knowledge Object Repository (KOR).*

If a node structure update is sent with an *alive ping*, it contains the expected *versionID* of the local copy of the sender's context repository structure. This *versionID* is used by a receiving KA to check, if the incremental update can be applied, or if a full update has to be requested from the KA the *alive ping* originates at.

The expected *versionID* is sent to ensure that the update is applicable and that no update was missed. If the local *versionID* of the *remote KOR* is higher or equal than the expected *versionID* from the update and the communicated *versionID* after applying the update is higher than the local *versionID*, the update is applied and the local *versionID* is set to the *versionID* from the *alive ping.* If the communicated *versionID* after applying the update is lower than the corresponding local *versionID* the *remote KOR update* is discarded as the local version is newer.

In case all KAs receive all *alive pings*, the structures of the *remote KORs* are always synchronized. See upper part of Fig. 6.3. In case no structure changes happen on a KA only the current *versionID* is disseminated. As shown in the figure, this is the typical case as changes on the context repositories only happen when new services are added, services are removed, or *list* entries are changed (Sec. 5.2.10).

The lower part of Fig. 6.3 shows the loss of an *alive ping* that results in a missed context repository structure update. Depending on the current configuration, an agent tolerates *n alive pings* with a *versionID* that does not match the local *versionID* of the *remote KOR*. Then an explicit update is requested.

The answer is sent via the multicast mechanism again as other KAs may have missed the lost *alive ping update* as well. Since duplicate *alive pings* are discarded receiving an older update has no impact on a KA. After successfully applying the structure update on all KAs the context repository structure is synchronized on all KAs again resulting in the right search results.

If a KA misses *alive ping* updates, its *remote KOR* is not synchronized anymore. However, this only affects the search capabilities as the context access happens independent of the described methods. The tolerance of missed updates enhances the dependability of the VSL implementation (<R.30>). *<R.30>*

6.4.8 Agent Discovery

The VSL overlay is spanned by the distributed KA peers. In addition to the synchronization of the *remote KORs* for the search (Sec. 6.4.6) the *alive ping* mechanism (Sec. 6.4.7) is used for agent discovery. When an agent is started it begins sending periodic *alive pings*. All other KAs in the same IP domain receive the *alive ping* (Sec. 6.3).

Like a service each KA has a certificate for authentication. The *agent certificate* contains the ID of a KA. All KAs that are synchronized to a VSL overlay (Sec. 6.4.9) know each other's IDs.

The KA IDs are ordered. When a KA wants to join an existing KA overlay, the KA of the overlay with the next existing KA ID above the one of the new KA acts as authenticator for the new KA. When it receives the *alive ping* of the new KA it sends an *authentication offer* to the new KA. See top part of Fig. 6.4.

Then both KAs authenticate mutually using the public keys from their own certificates to validate the received *agent certificate* (Sec. 5.5). See middle part of Fig. 6.4. Similar to the service authentication (Sec. 6.4.1), a binding record for the new KA is created as shown in Fig. 6.2.

AgentID	versionID	transportBindings	time to live	certificate expiration	public key

Table 6.2: The binding record of another agent in a KA.

The *agentID* is the ID of the new KA. The *versionID* contains the current local *versionID* of the *remote KOR* – the structure of the context repository on the KA. The

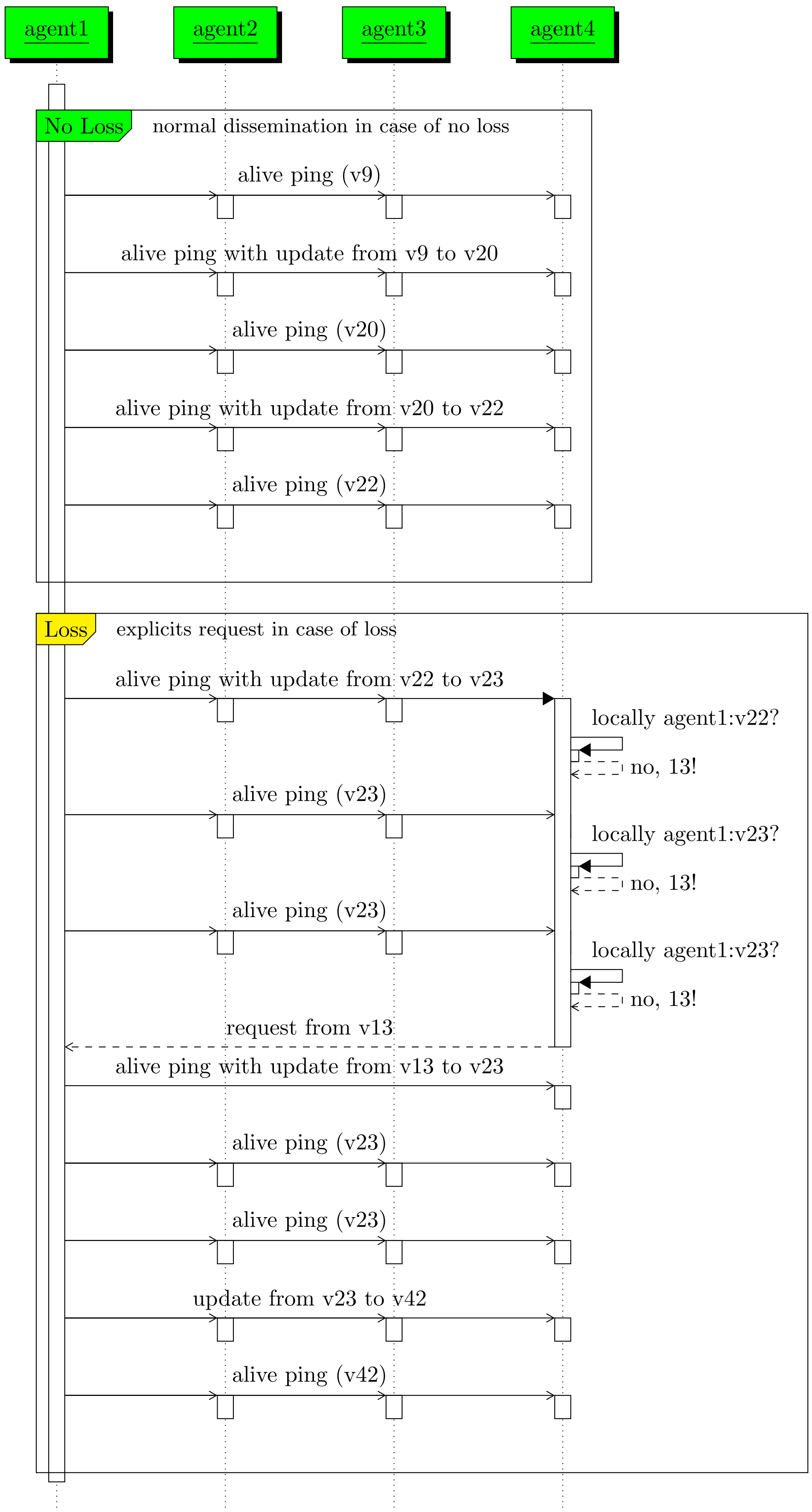


Figure 6.3: Periodical context repository structure updates (alive pings) from agent1.

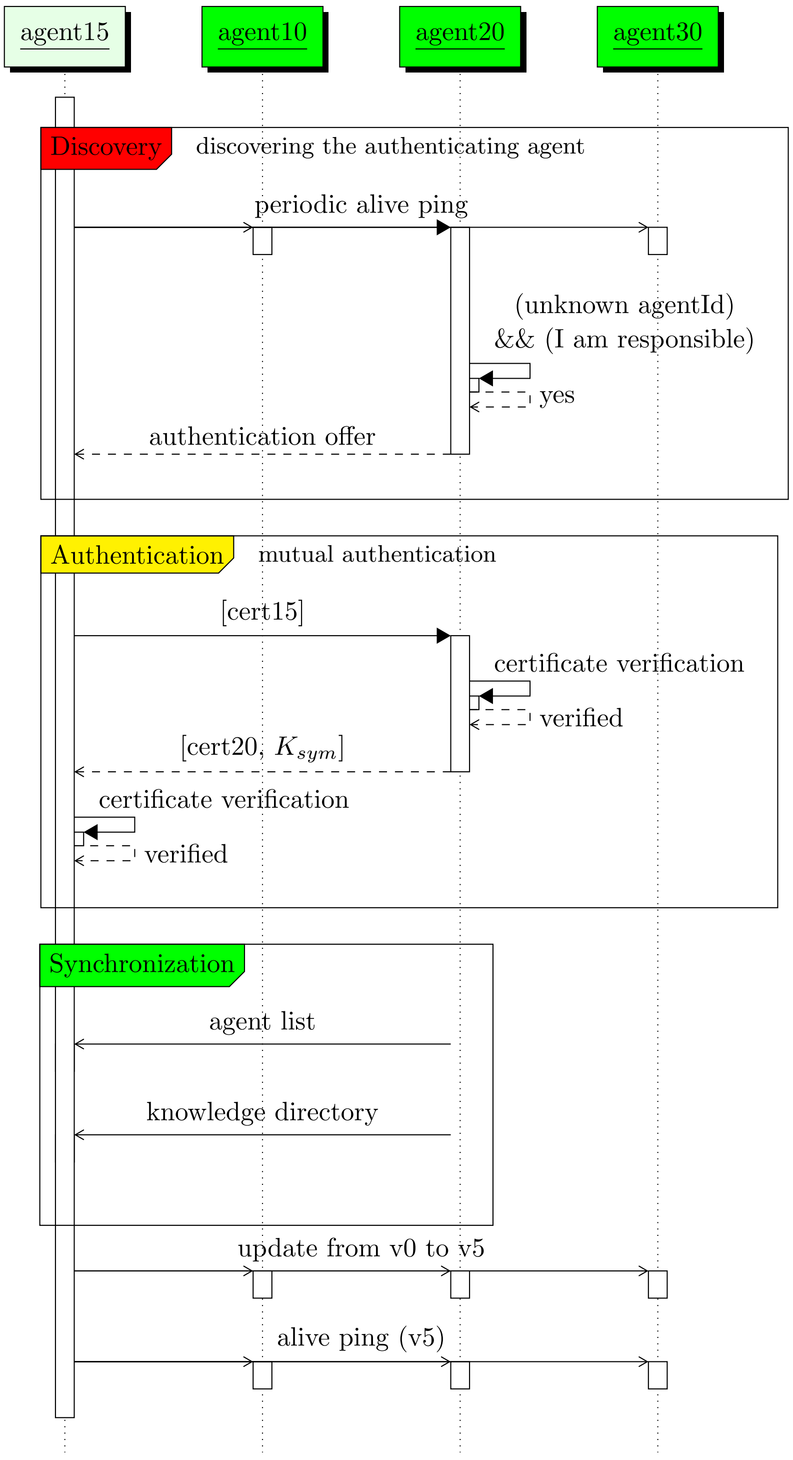


Figure 6.4: Authentication of a new KA “agent15” with a running VSL instance.

transportBinding contains the IP and port mappings to the communication interfaces of the remote KA. The *time to live* is used to determine if a KA is still reachable (Sec. 6.4.9).

The *certificate expiration time* is the time when the certificate of the corresponding KA expires and it has to be excluded from the VSL overlay (Sec. 6.4.1). The *public key* is the public key from the agent certificate that can be used to send encrypted information specifically to that agent.

If valid, the *authenticator* sends the new KA the list of other KAs and the context repository structures of all KAs in the overlay. As result, the new KA is synchronized with the VSL overlay.

The described initial handshake between KA is also used to synchronize the clock of the authenticated KA with the official time of the VSL. This is important to implement the distributed certificate revocation that is based on the validity timestamps in the certificates of KAs and services (Sec. 5.5).

To update all other KAs, the new KA sends an initial *alive ping* that updates all repositories for remote context structures in all KAs. Then the new KA is regularly part of the VSL overlay and sends regular context repository structure updates as described in Sec. 6.4.7.

To secure the process, all communication between KAs is encrypted using the symmetrical key K_{sym}. See center of Fig. 6.4. This allows all KAs to add the information from the new KA sent in its initial *alive ping*. The authenticity of this information is given as the new KA can only know the symmetrical key K_{sym} after authentication.

<R.50> The described mechanism implement a scalable KA discovery and integration (<R.50>).

6.4.9 Agent Synchronization

The KA synchronization happens via the periodic *alive pings* (Sec. 6.4.7). Each time an *alive ping* is received, the *time to live* in the binding record of the corresponding KA is updated.

If the *time to live* is up or the *expiration time* has come, the binding record and the local context repository structure of the KA are deleted. This removes it from the overlay.

6.4.10 Self-Management

To assess the autonomy of a system its *self-management properties* are often used [KC03, MALP12]. They include *self-configuration*, *self-healing*, *self-optimization*, and *self-protection*. The following assessment shows that the VSL design implements self-management (<R.3>). *<R.3>*

Self-management is fundamental for real-world usability (<O.4>). It enhances the dependability (<R.30>) and the security (<R.49>) of the VSL. *<R.30>* *<R.49>*

Self-Configuration

The VSL implementation is *self-configuring* as new KAs and new services are automatically integrated.

Self-Healing

The VSL implementation is *self-healing* as it consists of peers that permanently reorganize. Based on the *alive pings* the topology is periodically updated. In case of a partitioning or failures of KAs the changes are automatically reflected. The same applies when two formerly separate VSL overlays join.

Self-Optimization

The VSL implementation is *self-optimizing* as it removes unreachable context structure information and records about unreachable KAs.

Self-Protection

Via the security features that are described in Sec. 6.5 the VSL implementation is *self-protecting.*

6.5 Security

Diverse VSL security properties are described in Sec. 5.5. Following implementation-specific details are given in the same order like in Sec. 5.5. In addition the communication security that is implementation specific is introduced in Sec. 6.5.6.

As the described security mechanisms are self-managing they implement security-by-design from the perspective of a service (<R.49>). *<R.49>*

The VSL implementation uses Advanced Encryption Standard (AES) [oT01] for symmetric encryption, and RSA [RSA78] for asymmetric encryption.

6.5.1 Identification

As format for the certificates the structure of the X.509 standard [CSF$^+$08] is used. The additional information described in Sec. 5.5 is stored in X.509 v3 extensions. The hierarchical structure of a X.509 public key infrastructure is not used. Instead each VSL site acts as its own Certification Authority (CA) as described in Sec. 5.5.1.

The distributed revocation of certificates that is based on their validity period (Sec. 5.5.1) is supported by using the alive pings for synchronization of the time that is used in a VSL overlay (Sec. 6.4.8).

Holding the expiry timestamp in the agent binding records (Fig. 6.2, Sec. 6.4.8) and the service binding records (Fig. 6.1, Sec. 6.4.1) ensures that entities without a valid certificate are excluded during runtime. This is important as it can be expected that entities in VSL Smart Spaces are running for a long time. Pervasive computing is supposed to “happen” all time in the background and not only within a limited timeframe.

6.5.2 Authentication

The authentication happens as described in Sec. 5.5 via the X.509 certificates.

6.5.3 Authorization

The authorization happens as described in Sec. 5.5. The access rights of services are stored in the service binding records of the KAs at run time. This is shown in Fig 6.1, and described in Sec. 6.4.1.

The access permissions are stored in the context models. Authorization happens by matching the two. The authorization applies to regular context and Virtual Context in the same way (Sec. 5.5).

6.5.4 Rights Delegation

The rights delegation happens as described in Sec. 5.5. Additional certificates are directly stored in the service binding records as additional rows with the *shortName* as suffix (Sec. 5.3.1).

6.5.5 Confidentiality

The data in the database is not encrypted in the current implementation. It is planned to use a chained keying approach with site-local group keys that are mapped to the accessIDs in the certificate and added to it. The feasibility of such an approach is shown in [App12, WOW08].

6.5.6 Communication Security

To prevent eavesdropping the communication between KAs is symmetrically encrypted with a shared key K_{sym} (Fig. 6.4 center) that is exchanged with new KAs during the initial pairing (Sec. 6.4.8, <R.49>). Each KA has a service certificate. It is used for mutual authentication between the KAs during the initial paring. Certificates are externally added to the software components. Ch. 7 introduces an automated workflow for managing VSL service certificates. *<R.49>*

The encryption can be disabled for debugging purposes.

Periodical re-keying

Periodical re-keying happens to make sure that agents which previously had access but do not have a valid certificate anymore are unable to access the VSL. For the re-keying asymmetric encryption is used.

The KA with the highest ID in the agent binding record list sends the re-keying message. The key to decrypt the new group key is encrypted once with each of the public keys that are currently valid. The message is distributed via multicast like an *alive ping.*

The use of the public keys enables only KAs with valid certificates (Sec. 6.4.8) to decrypt and use the new group key. All keys and expiry timestamps that are needed to create the re-keying message are part of the *agent binding record* list making the process scale (Fig. 6.2, Sec. 6.4.8, <R.50>). *<R.50>*

6.6 Basic VSL Services

Ch. 5 introduced the CMR (Sec. 5.2.8), the CMR service (Sec. 5.7.1), the type search (Sec. 5.7.2), and the semantic extensibility at the example of the location search provider (Sec. 5.7.3). This section presents some implementation details.

6.6.1 Context Model Repository

The Context Model Repository (CMR) is the global repository for context models that implements the collaborative ontology (Sec. 5.2.18). See Fig. 6.1.

The CMR implements the functionality described in Sec. 5.2.8. It is implemented as regular website on a regular web server. Context models can be downloaded via HTTP. New context model can be submitted via web form. Context models are stored as plain XML files.

The implementation as website on a regular web server allows to apply existing load balancing techniques [GJP10]. The addressing of the VSL context model matches with that of Unified Resource Locators (URLs) facilitating the implementation and simplifying the use (Sec. 5.2.3, <R.1>). *<R.1>*

As context models have Globally Unique IDentifiers (GUIDs) and new context model versions must have new IDs, the distribution and caching of the CMR is enabled. All described features make the only central component of the VSL programming abstraction scale (<R.50>). *<R.50>*

6.6.2 CMR Service

As described in Sec. 5.7.1 and shown in Fig. 6.1, the *CMR service* provides access to the CMR for all KAs of a VSL site.

The connection to the CMR is implemented via HTTP socket connections. The caching is implemented by storing the obtained XML files locally on disk. The same caching strategy is applied in the distributed KAs.

After initialization of the *CMR service*, the following context is stored in the VSL managing the request handling to the Virtual Node callback:

```
<cmr type="/system/cmr,/system/virtualNode"
     version="0" timeStamp="2014-02-14 21:23:24.014" access="rw">
  <serviceId type="/system/serviceId,/basic/text" version="1"
             timeStamp="2014-02-14 21:23:24.945" access="rw">
    <![CDATA[/agent_mop/client:cmr]]>
  </serviceId>
</cmr>
```

Listing 6.3: Virtual Node callback binding as it is represented in the VSL and used internally by KAs.

The callback for the Virtual Node is identical to that shown in listing 6.4. Only the return statement calls the class that loads the context model from the local cache or the CMR in case of a cache miss.

Fig. A.1 in Sec. A.2 shows the different caches that can be used for instantiating a context model.

6.6.3 Type Search

The *type search* (Sec. 5.7.2) that identifies context node locators based on a given context node type identifier directly uses the database back-end for querying all node addresses a service has access to that are of the searched type as described in Sec. 6.4.2.

The search functionality is implemented as Virtual Node callback as shown in listing 6.4. The binding to the Virtual Node callback is stored analog to listing 6.3.

```
1  @Override
2  public String get(final String address, final String readerID) {
3    String parameter = address.substring(myKnowledgeRoot.length());
4    String startAddress = getStartAddress( parameter );
5    String modelId = getModelId( parameter );
6    return kor.getAccessibleAddressesOfType(startAddress, readerID,
         modelId);
7  }
```

Listing 6.4: Virtual Node callback implementation of the type-based VSL search.

The third line of listing 6.4 shows the extraction of the parameter from the passed address. In the fourth and fifth line the two parts of the parameter are identified using the `"?"` as split character as described in Sec. 5.7.2.

6.6.4 Location Search

The *location search provider* enables a locator-id split based on locations as described in Sec. 5.7.3. It can be combined with the type based locator-id split, e.g. `get /search/typeAtLocation/livingRoom/?/basic/number`.

Sec. 5.7.3 discussed advantages and disadvantages of an *implicit* version (requires context model extension), and an *explicit* version that does not require context model extensions. Both were implemented as prototype. The implicit version uses its own database back-end. The explicit version also uses a database that is periodically populated by a search over all context models instances to make the search for locations faster.

The implicit version is currently used as it does not require any changes on context models in the CMR. As Smart Devices typically will not have the ability to locate themselves within a Smart Space, the *location search provider* is coupled with the generic user interface that is presented in Sec. 8.3.5.

The coupling allows setting locations of VSL services via drag and drop. New services such as an adaptation service for a lamp (Sec. 8.3.1) can be moved to the physical position of the controlled lamp on a map. The location that is determined in the described way is stored in the database of the *location search provider*.

6.7 Chapter Conclusion

The chapter introduced the concept of μ-middleware that enables the use of one middleware for diverse use cases (Sec. 6.2). The μ-middleware properties of the VSL implementation follow directly from the VSL programming abstraction.

As the assessment of the state of the art in Sec. 4.5 that is summarized in table 4.1 shows, none of the existing pervasive computing middleware is a μ-middleware. Most solutions cannot be extended at all and provide fixed core functionality instead. The solutions that can be extended do not offer this feature in a transparent way. Especially they do not provide service discovery. None of the assessed solutions offers Virtual Context since it is a novel concept that is introduced in this work. As the implementation shows, the dynamic extensibility is based on Virtual Context. Chapter 7 will show more extensions to the VSL μ-middleware.

This chapter showed how the VSL can be implemented as autonomous system that manages itself (Sec. 6.4). The key functionalities presented are the agent and context synchronization (Sec. 6.4.9, Sec. 6.4.7), and the security features (Sec. 6.5).

From a scalability perspective the full distribution is most relevant (Sec. 6.3).

From a developer perspective (<R.1>, <R.4>, <R.48>) the openness of the implementation and the availability of native connectors are important (Sec. 6.3.2).

As indicated on the overview page at the beginning of the chapter, the VSL implementation provides solutions to almost all of the remaining requirements that were identified in chapter 3.12 and not solved in Ch. 5 already.

Portability of the VSL implementation (<R.7>) is implemented by using *Java* and *open communication standards*. *Support for multiple programming languages* to implement services (<R.27>) is implemented by the open interfaces and by providing *connectors* to different programming languages. *Portability of service implementations* (<R.8>) is implemented when Java is used.

The only open requirement is the *availability of an App store* (<R.46>). The Smart Space Store (S2Store) that is presented in chapter 7 takes this role. In addition, the next chapter introduces μ-middleware extensions for service management as this functionality is missing compared to the combined reference model shown in Fig. 4.3 so far.

7. The Distributed Smart Space Orchestration System

The developer community is multiplying from the hardcore developers that may be 10m people to over 100m people around the planet writing apps, and the chance to do more and profit economically is growing.

Steve Ballmer (*1956), Chief Executive Officer, Microsoft, in a talk at BUILD developers conference 2011.

Short Summary This chapter introduces the Distributed Smart Space Orchestration System (DS2OS) framework. DS2OS consists of different Virtual State Layer (VSL) services plus an App store. Together they provide the technical foundation for an App economy for Smart Spaces (Sec. 3.10).
All components of DS2OS besides the Smart Space Store (S2Store), and the App store for Smart Spaces, are implemented as regular VSL services. Making us of its own features for implementing core functionality illustrates the power of the programming abstraction.
DS2OS provides autonomous service management within a site, and service deployment from the central S2Store.
Different mechanisms that support crowdsourced development, and real world deployment of Smart Space services are introduced. The collaborative convergence mechanism from Ch. 5 is extended to a crowdsourced convergence mechanism.
A fully distributed, simple-to-use security concept for Smart Space services is introduced.
DS2OS is assessed using the criteria from the state of the art analysis (Sec. 4.5).

Key Results The μ-middleware property of the VSL (Sec. 6.2) allows the extension with functionality that other pervasive computing middleware designs offer dynamically via services at run time.
The VSL programming abstraction simplifies the creation of complex services.

Self-management in the back-end makes it possible to provide an intuitive interface to software orchestrated Smart Spaces.
Crowdsourced convergence is introduced as novel mechanism for collaborative standardization.

Key Contributions The technical basis of an *App economy for Smart Spaces* is introduced. None of the components exists for Smart Spaces in 2014.
A *crowdsourced convergence* mechanism for the Smart Space *ontology* is introduced. It improves the convergence of the approach that was introduced in Ch. 5, and makes it more suitable for real world deployment. It exceeds existing approaches for collaborative ontologies from literature [HJ02, KA06, LST13].
A *crowdsourced convergence* mechanism for Smart Space *services* is introduced. It implements a self-managing quality assurance mechanism.
A *fully distributed simple-to-use security concept* for software orchestrated Smart Spaces is introduced. It includes distributed revocation of access rights, and a crowdsourced convergence mechanism for access identifiers. The concept exceeds existing approaches [CD05, KRM$^+$07, Lac09, AO04, CD07].
A *service management framework* with *local and global* components is introduced.
A concept for an *App store with adapted mechanisms for Smart Space services* is introduced.

Challenges Addressed <R.1> *Simplicity-to-use*, <R.3> *Self-management*, <R.4> *User participation*, <R.8> *Interface portability of services*, <R.23> *Dynamic extensibility*, <R.26> *Modularization support*, <R.30> *Dependability*, <R.45> *Portable service executables*, <R.46> *App store*, <R.48> *Crowdsourced development*, <R.49> *Security-by-design*, <R.50> *Scalability*.

Summary The Distributed Smart Space Orchestration System (DS2OS) framework is introduced that consists of the VSL μ-middleware, the Smart Space Service management (S2S) service management architecture, and the S2Store (Sec. 7.2).
The chapter starts with a comparison of the requirements of service management in Smart Spaces with App management in the App economy (Sec. 7.1.1).
Next, the hierarchical architecture of DS2OS with the Run Time Environment (RTE) (Sec. 7.2.1), the Service Hosting Environment (SHE) (Sec. 7.2.2), the Node Local Service Manager (NLSM) (Sec. 7.2.3), and the Site Local Service Manager (SLSM) (Sec. 7.2.4) is introduced. The different components and their functionality are described. Then the format of DS2OS service packages is presented (Sec. 7.2.5) before the introduction of the S2Store (Sec. 7.2.6).
The self-managing security architecture of DS2OS is introduced that complements the security architecture of the VSL (Sec. 5.5).
Next the crowdsourced convergence mechanisms that are provided by DS2OS are introduced (Sec. 7.4). They provide convergence of context models, access groups, and service executables.
The functionality of DS2OS is illustrated at the example of a service life-cycle (Sec. 7.5).

Finally DS2OS is assessed according to the criteria that were applied for the state of the art analysis in Sec. 4.5 (Sec. 7.7).

7.1 Introduction

This chapter introduces the Distributed Smart Space Orchestration System (DS2OS). Distributed Smart Space Orchestration System (DS2OS) is implemented using the Virtual State Layer (VSL) programming abstraction. It adds functionality for *service management* within a VSL site, a global *App store* called Smart Space Store (S2Store) for VSL service distribution, and further support for *crowdsourced software development* including *crowdsourced convergence.*

The DS2OS components are implemented as regular VSL services. Its major design elements are presented to illustrate how the VSL programming abstraction facilitates the implementation of complex functionality. It also shows that the VSL μ-middleware offers all functionality required to implement diverse Smart Space scenarios in its core. It is shown how things that are considered missing in the μ-middleware core can be added and accessed transparently (Fig. 4.3, Sec. 6.2).

DS2OS comprises three building blocks:

- The Virtual State Layer (VSL) μ-middleware that acts as distributed operating system within a Smart Space.
- The Smart Space Service management (S2S) service management framework that manages VSL services.
- The Smart Space Store (S2Store) that manages services globally, and that supports crowdsourced software development.

At the time this document is written, DS2OS is not fully implemented. The feasibility of the concept follows directly from the use of the VSL as shown in this chapter. The Service Hosting Environment (SHE) (Sec. 7.2.2) is a fundamental component for autonomous management of services that is not implemented as VSL service. It is implemented as prototype.

7.1.1 Comparison to the App Economy

In Sec. 3.10 the App economy was presented as an important success factor for the real world deployment of mobile computing (Sec. 2.3). It is a successful example for fostering crowdsourced development.

Using the VSL programming abstraction (Ch. 5) and its implementation as VSL μ-middleware (Ch. 6), DS2OS provides an infrastructure that enables an App economy for Smart Spaces. A VSL Smart Space that runs the DS2OS framework is called *DS2OS Smart Space*, or *DS2OS site.* The VSL corresponds to the kernel of a Distributed Smart Space Operating System (which is an intended second reading of the acronym DS2OS). Two major differences between the VSL and a smartphone Operating Systems (OSs) are:

- The VSL provides a unified system view on the *distributed* entities of a Smart Space via its context-based interface (Ch. 5, Ch. 6).
- The VSL can be extended dynamically and transparently with adaptation services that act as device drivers for Smart Devices. This extensibility covers the additional diversity of Smart Spaces concerning the available peripherals (Sec. 3.10).

A middleware targets providing a unified system view (Sec. 3.4). The VSL implements the required abstraction over the distribution of nodes with its distributed design, and its self-management properties (Ch. 6). The dynamic extensibility and the transparency are implemented by the μ-middleware properties (Sec. 6.2), in particular the locator-id split (Sec. 5.2.13) and the type search (Sec. 5.7.2). They allow to discover the abstract context interfaces of new services dynamically at run time.

The Smart Space Service management (S2S) provides App management mechanisms within a DS2OS site. Its Site Local Service Manager (SLSM) (Sec. 7.2.4) corresponds to the App managing App store component that runs on smartphones. Its SHE corresponds to the smartphone OS itself. It enables to execute services. Different to smartphone App managers, the S2S provides service management functionality for the *distributed* computing hosts of future Smart Spaces.

The S2Store (see below) takes the role of a smartphone App store. Different to a smartphone App store, the S2Store provides not only management of service executables, but also *context model management*, and *crowdsourced convergence* for service implementations and context models (Sec. 7.4).

7.2 Service Management

Like the VSL μ-middleware, DS2OS operates distributed and fully autonomously (Sec. 6.4, Sec. 7.6.1). This implements scalability (<R.50>) and allows the combined use of the available distributed computing resources in future Smart Spaces as Smart Space Orchestration backbone (Sec. 3.2.4). In addition, the self-managing components (Sec. 7.6.1) make the complexity of Smart Spaces with their distributed, often unattended Smart Devices manageable (Sec. 3.2). *<R.50>*

The usability of DS2OS is comparable to that of a smartphone App store (Sec. 7.6.4). Since App stores enable non experts to deploy Apps to their smartphones, DS2OS enables non experts to deploy VSL services to future Smart Spaces. A good usability is a fundamental requirement for real world usability of pervasive computing (<O.4>) [Abo12, BLM$^+$11, MH12, TPR$^+$12].

DS2OS implements hierarchical management with the following architecture components that are shown in Fig. 7.1:

- **Run Time Environment (RTE)**: The Run Time Environment (RTE) is the execution environment that runs VSL service executables.
- **Service Hosting Environment (SHE)**: The SHE provides low-level service management functionality on a computing node. It allows to control and monitor service executables that run in the RTE.
- **Node Local Service Manager (NLSM)**: The Node Local Service Manager (NLSM) provides *node-local* service management functionality. It starts, stops, pauses, and monitors all VSL services on a computing node.
- **Site Local Service Manager (SLSM)**: The SLSM manages all NLSMs. It provides *site-local* service management. The SLSM optimizes the resource usage in a DS2OS site according to different metrics such as load balancing or resilience.

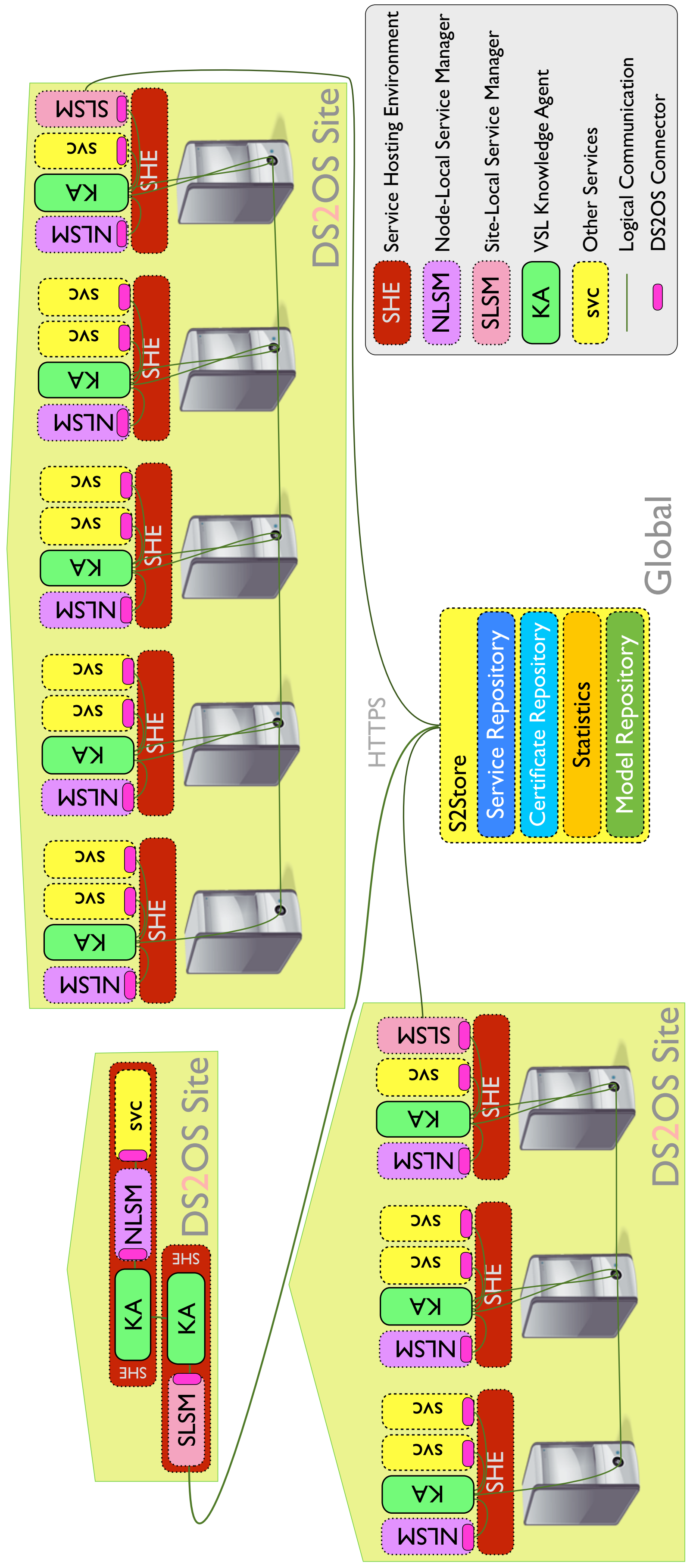


Figure 7.1: System view on the Service Management Architecture.

- **Smart Space Store (S2Store)**: The S2Store comprises the Context Model Repository (CMR) and implements a repository for services. It collects and publishes statistics from distributed DS2OS sites. It implements a crowdsourced convergence of context models and DS2OS services.

Fig. 7.1 shows the relationship between the entities. On the top left the service management hierarchy is shown with the SLSM that manages the site, an NLSM that manages a computing node, and a service. A different view on the hierarchy is shown in Fig. 7.2.

All components of the service management implementation except the S2Store and the SHE are implemented as regular VSL services. The VSL is not modified for implementing DS2OS. Service management is added as-a-service using the μ-middleware property of the VSL (Sec. 6.2).

The VSL connector (Sec. 6.3.3) is extended with functionality for service management. The interface to the additional functionality is made available via Virtual Nodes over an additional context model that gets instantiated when a service that uses the DS2OS connector is started.

The added functionality allows to *start*, *stop*, and *pause* services (Sec. 7.2.2) and to transfer their context (Sec. 7.2.4). See Fig. 7.1 "DS2OS connector". The state *pause* means that a service is started but it does not change any of its context. This state is used for *update*, *migration*, and *replication* of services (Sec. 7.2.4).

Following, the different components of DS2OS are introduced in more detail. Java is chosen for their prototypical implementation (Sec. 7.2.1). The presented concepts are independent of the used format of the service executables. This allows implementing DS2OS in different programming languages.

7.2.1 Run Time Environment

The Run Time Environment (RTE) provides the execution environment for service executables.

As described in Sec. 6.3.1, Java is a run time environment that provides portability (Sec. 5.6.3) to run VSL service executables on heterogeneous computing nodes. It provides a unified abstraction on top of different hardware architectures and different OSs. Therefore Java is used as run time environment for implementing the DS2OS prototype. Any other framework with similar properties such as .net could have been chosen as well.

As described in Sec. 6.3.3 it would also be possible to implement DS2OS services in another programming language, e.g. a scripting language like Python. Java is chosen for the prototype since it is deployed on smartphones [VSCK11, But11], and on other appliances within Smart Spaces already [The12, KBY+12, FMF+12]. The existing deployments show the suitability of Java for the targeted hardware platforms

7.2.2 Service Hosting Environment

The Service Hosting Environment (SHE) adds Smart Space Orchestration capability to the RTE. It provides the low-level service management functionality to *start* and *stop* service executables.

The SHE prototype is based on Open System gateway Initiative (OSGI), a Java framework that allows to start and stop services as so-called bundles [The12]. The OSGI bundles are implemented using iPOJO[30]. iPOJO allows developers to implement services as Plain Old Java Objects (POJOs) without having to care about OSGI-specific details.

The result of using iPOJO is that service developers can develop regular Java applications, while the DS2OS connector (Sec. 7.2) wraps the OSGI functionality that is needed for the *node local* low-level control of service executables. This simplifies the development of DS2OS services (<R.1>). *<R.1>*

7.2.3 Node-Local Service Manager

The Node Local Service Manager (NLSM) manages all services on a computing node. It controls (start, stop, pause, migrate context) and monitors services using the functionality provided by the SHE and the DS2OS connector. The NLSM reflects change requests that it receives from the SLSM such as starting new services or requesting locally running services to migrate their context to another Knowledge Agent (KA) as they are about to be migrated there.

The major functional goal of an NLSM is keeping all services on the computing node it manages running. Services are delegated to managed computing nodes by the SLSM (Sec. 7.2.4). Another functionality of an NLSM is providing monitoring data about each service it manages. To reach the goals, the NLSM checks periodically if all services are up via the SHE. If a service that should be up is down, it tries to restart it. In case of an error it reports to the SLSM. The NLSM monitors different parameters of a service including the Central Processing Unit (CPU)-usage, the Random Access Memory (RAM)-usage, the disk-usage, and run time errors.

The NLSM is a regular DS2OS service. Following the *separation of service logic from service state* (Sec. 5.6.5), the NLSM service stores its state in the VSL. This allows to restart the NLSM without loosing its current state. In combination with the functionality of the SHE always to restart the NLSM on failure, this enhances the dependability of the NLSM as node-local management component (<R.30>). *<R.30>*

The NLSM uses the VSL for communication with the SLSM. The communication includes receiving service executables and commands from the SLSM. Using the VSL simplifies the NLSM implementation as connectivity (Sec. 6.4.8) and security (Sec. 6.5) are handled by the VSL.

To illustrate the use of the VSL programming abstraction, some implementation details are presented. Data about all locally managed services is stored as a VSL *list* (Sec. 5.2.10) in the context model of the NLSM. Listing 7.1 shows parts of the context model of the NLSM service. The list of running services is stored in the context subtree `runningServices` using the context model shown in listing 7.2 for the service records.

```
<nlsm>
  <overallServicesRunning type=".../nlsm/servicesRunning">
    0
  </overallServicesRunning>
  <cpuAvailable type=".../nlsm/cpuValue">
```

[30]http://felix.apache.org/site/apache-felix-ipojo.html

```
    0
  </cpuAvailable>
  <currentCpuUsed type=".../nlsm/cpuValue">
    0
  </currentCpuUsed>
  ...
  <runningServices type="../nlsm/listOfServices">
  </runningServices>
  ...
</nlsm>
```

Listing 7.1: Parts of the context model of the NLSM.

The meta data of a service has the structure shown in listing 7.2. Each service record in the *list* contains another *list* for recording error reports.

```
<serviceRunTimeData> ...
  <currentCpuUsed type=".../nlsm/cpuValue">
    0
  </currentCpuUsed>
  ...
  <errorEvents type="../nlsm/listOfErrorEvents">
  </errorEvents>
  ...
</serviceRunTimeData>
```

Listing 7.2: Parts of the context model of a service that is managed by the NLSM. The entries of the runningServices node of the NLSM context model are of the shown type.

The NLSM runs autonomously. It receives its high-level goals from the SLSM, e.g. service executables to run on its managed computing node. The operation of a NLSM is not affected if a partitioning of the VSL makes the SLSM unreachable (Sec. 5.6.1). A partitioning of the VSL could happen if network connections to a KA are interrupted for instance. If the SLSM, and the NLSM cannot communicate, the only difference for an NLSM is that it does not receive new goals until the SLSM can reach it again. With its local goals the NLSM continues keeping the local services running providing dependability of services (<R.30>). *<R.30>*

7.2.4 Site-Local Service Manager

The Site Local Service Manager (SLSM) provides the functionality for site-local service and resource management. It manages all services within a DS2OS site. Like the NLSM the SLSM is implemented as regular DS2OS service.

The SLSM coordinates the NLSMs on the distributed computing nodes. It provides the interface of a DS2OS site to the S2Store. The SLSM maintains a list of all services that are currently running within a DS2OS. Following, different functionality of the SLSM is introduced.

Service Repository

The SLSM acts as site-local repository for DS2OS services. It stores all service packages (Sec. 7.2.5) that are downloaded from the S2Store into the DS2OS site. See Sec. 7.2.6 and Sec. 7.5.

Service Deployment

The SLSM decides on which computing host of a site a service should be deployed. Then it sends the service to the NLSM on the chosen computing node. It uses the VSL for transfer of the service package (Sec. 7.2.5). The NLSM offers a Virtual Node in its context model for that purpose (Sec. 5.3.4). After receiving the service and the command to do so, the NLSM starts and manages its new service autonomously. During operation it provides statistics about the new service for the SLSM (see below).

Different optimization strategies can be implemented. The prototype implements simple load balancing by sending a service always to the computing node that has most free resources according to the requirements of a service (e.g. CPU, RAM, Hard Disk Drive (HDD)). The SLSM collects load statistics of the available computing nodes regularly from the distributed NLSMs as basis for decisions.

Using the VSL type search (Sec. 5.7.2) for identifying currently available NLSMs (`.../nlsm`, see listing 7.1), the SLSM automatically adapts to changes in the Smart Space backbone. Using the VSL mechanisms implements self-management (<R.3>) and enhances the dependability of the S2S (<R.30>). *<R.3>* *<R.30>*

Service Migration

The *separation of service logic from service state* allows to restart services without loosing their state. This becomes possible as the service state is stored independently from the service logic in the corresponding KA (Sec. 5.6.5). This VSL feature simplifies *migrating* services (<R.1>). *<R.1>*

Code migration describes the process of moving programs between machines [TVS02]. So-called *weak mobility* describes the ability to move the code segment of an application. It can require some initialization. So-called *strong mobility* means that the execution segment can be transferred as well. Strong mobility enables to run an application in the same state after migrating it.

For services that separate logic and state as proposed in Sec. 5.6.5 the execution segment is stored as context in the VSL. Since the VSL context can be moved as described next, DS2OS offers *strong mobility* for services that manage their state only in the VSL. *Strong mobility* of services leads to flexibility in a site's resource management. Services can dynamically be moved to other computing hosts. This can be used to implement a scalable system (<R.50>). *<R.50>*

An extended DS2OS VSL connector (Sec. 6.3.3) offers the possibility to transfer the current context of a service on another KA where a copy of the service is installed. The source of the migration is called *originator* in the following. The context transfer is done via a Virtual Node that is automatically provided by the DS2OS connector.

A connector is compiled or linked into a service. As part of the service, it has the VSL identity of that service. The corresponding access rights allow it to access all context on behalf of its service and send it to the corresponding connector of the service copy independent from the service implementation. The connector of the copy adds the received context to the context repository in the KA on the new computing host.

For *migration*, a copy of a service is started on another computing node in *paused* state (Sec. 7.2.2). Then the originating connector is asked to transfer its current context (service state) to the copy. Before transferring the context, the originating NLSM pauses the service to prevent additional changes that would lead to inconsistencies between the context repositories.

After the context transfer is done the service copy, the copy is started. Consequently, it uses the transferred state in the context repository. In case of a stateless service implementation, this results in the copy being in exactly the same state, the migrated service was before migration. The original service is stopped. All context nodes on the originating KA are marked read-only. The historic context remains available. In case, a service is started on the original node later again, the initialization of the context model overwrites the access permissions re-enabling write access on those context nodes that had write access before the migration.

Via the locator-id split (Sec. 5.2.13) other services automatically discover the new location of the migrated service.

To minimize the downtime of a service further the pause state can be prevented as follows. This could be interesting in case of large amount of context to be transferred from the originator and low downtime requirements.

The originating service connector can transfer its service's state while it is running. But it must log additional changes from when the transfer happened. When the service copy signals that it is ready, the originating service can pause, transfer the remaining changes since the first migration, and start the copy.

For keeping a service reachable on its original location, a proxy can be started on the originating host. This is interesting to cover the time until other services that access context of the migrated service do a rediscovery via the type search. The VSL programming abstraction simplifies the creation of generic proxies via its Virtual *<R.1>* Context (<R.1>). A proxy registers all context nodes of the originating service as Virtual Nodes and forwards all requests to the new location of the service. This is similar to port forwarding in operating systems [RR81].

Migration is controlled by the SLSM.

Service Replication

Service replication works similar to service *migration.* A difference is that the original service is not paused, and the copy is not started after a context transfer. Instead the context is transferred between the participating service copies periodically. Because of the periodical synchronization, a service copy that is running on another host can do a failover on the last synchronized state. It is only started, when the original service fails.

Service replication is interesting in case of computing node failures. It can be used *<R.30>* to enhance the dependability of DS2OS services (<R.30>). A pure service failure can often be handled by simply restarting a service. This is done automatically by the NLSM. This autonomic strategy complements the described service replication.

Service Update

The SLSM periodically checks the S2Store for updates of the locally installed DS2OS services. If an update is available it downloads the new service package and initializes it via the distributed NLSMs on all computing nodes within its managed local site (Sec. 7.3). Then it sends the new service package to all NLSMs that are running the service. The NLSMs try to start the new service in *paused* state. As the new service version typically has the same *serviceID* in its service certificate (Sec. 5.5.1, Sec. 7.3), it connects to the context of the older service version automatically.

If the starting works the old version is *paused* and the new one is set to *started* state. If no error occurs the old version is stopped and removed. If an error occurs the old version is started again and the new version is removed and the SLSM is informed. This implements a self-managed update process (<R.3>). *<R.3>*

DS2OS Update

The described service update mechanism can be used to update the DS2OS components themselves, not only external services. Fig. 7.6 shows that the KAs could run on top of the SHE. If DS2OS is run as service containers, the platform itself can be updated as described before for regular services that run on top of the platform.

While a KA is down its context repository is not available for services. Therefore the NLSM pauses all services first. Then it runs an update process that is independent of VSL context for updating the KA executable. The failure handling can be implemented as described before. In case of the updated version not running properly, the previous version of the KA can be started again. As the context repository is stored in a local database (Sec. 6.3.5) the new KA can directly serve the existing context after start. It runs in the same state, its previous version was when the update started.

The same can be done with the NLSM, the SLSM, and even the SHE. It is only requires to have an update mechanism that is independent of other DS2OS components implemented on each node. It updates the SHE and restarts the old version in case of a failure.

This implements a self-managed update process of the framework itself (<R.3>), making it simple to use DS2OS (<R.1>). In addition it enhances the security of the system as even the system components can be updated automatically (<R.49>). *<R.3>* *<R.1>* *<R.49>*

Statistics Collection

The SLSM periodically collects statistics from the computing nodes that run an NLSM. The collected data include:

- Overall resource usage on the computing node (e.g. CPU, RAM, HDD).
- Resource usage per service.
- Error statistics per service (e.g. service failures).

The SLSM uses the statistics for local optimization (see below) and it communicates aggregated anonymized statistics to the S2Store when the site owner allows that. The statistics that are sent to the S2Store are used for the crowdsourced convergence – primarily the usage statistics– and as feedback to the developers –primarily the error reports (Sec. 7.4).

Optimization Strategies

To change the available resource optimization strategies only the SLSM service has to be exchanged or re-parameterized as the other components only provide basic functionality. As described in Sec. 6.3.6, a re-parametrization can be done via changing the VSL context.

As proof of concept a simple strategy is implemented in the prototype that deploys services always to the host with the least load. It migrates services when a fixed percentage of load is reached on a computing node. Other strategies such as taking into account dependencies between services make sense and are future work.

7.2.5 DS2OS Service Package

A DS2OS service package consists of

- The service certificate (Sec. 5.5.1).
- A service manifest with additional meta data.
- The service executable.

DS2OS service packages are implemented as compressed ZIP containers.

Service Certificate

The *service certificate* (Sec. 5.5.1) is extended with a hash over the service manifest that allows to verify the integrity of the manifest.

Service Manifests

The *service manifest* contains information that is relevant for service management. The integrity of the *service manifest* is protected by the signed hash as part of the extended *service certificate*. A DS2OS service manifest contains:

- The unique name of the service.
- {Required, conflicting, obsoleting} context models [MBdC$^+$06].
- Computational resource requirements.
- Resilience requirements.
- Version number.
- DeveloperID.
- A *cryptographic hash* over the service executable.

The *unique name* is used as mapping to the *service certificate*.

The *required context models* describe dependencies to context models that must exist within a DS2OS site for the service to operate. An example is a service that implements functionality to switch lamps and that has no use if no context model of type `lamp` is available (Sec. 7.5). The described DS2OS architecture enables the automated deployment of depending services making a DS2OS site dynamically extensible
<R.23> (<R.23>).

As the VSL uses context models as abstract service interfaces, dependencies to context models are dependencies to services (Sec. 5.4.1). As adaptation services (Sec. 8.3.1) connect Smart Devices to the VSL, context model dependencies also express dependencies to Smart Devices (e.g. lamps).

The *conflicting context models* represent conflicts with other services, e.g. an *always-heat-up* service may conflict with an *always-cool-down* service. Conflicts are considered by the SLSM before deploying a service to a NLSM.

The *obsoleting context models* are used to express that a service update requires not only the change of the service executable but also the change of the context model that belongs to the service.

The *computational resource requirements* express the resources a service needs for running. They include CPU, RAM, or storage and are intended to be taken into account by the SLSM in its optimization strategy (Sec. 7.2.4).

The *resilience requirements* are to be taken into account by the SLSM in its optimization strategy (Sec. 7.2.4). They influence how many replica of a service are started and on which computation node it is deployed Sec. 7.2.4).

The *version number* is used by the update mechanism (Sec. 7.2.4) to determine if a newer version of an installed service is available.

The *developerID* is used to identify the origin of a service. It is used for making the feedback of a service available to its developers for instance (Sec. 7.4.5).

The *cryptographic hash* over the service executable is used by the SLSM and the NLSMs before starting a service to verify the integrity of the service executable.

Service Executable

As format of the service executable the Java ARchive (JAR) format is used in the prototype.

7.2.6 Smart Space Store

The sharing of context models via the CMR (Sec. 5.7.1) is necessary for developers to create portable (Sec. 5.6.3) services. In combination with the simplicity of the VSL programming abstraction it implements a Service Oriented Architecture (SOA) enabling crowdsourced development (Sec. 5.4.4).

The portability of DS2OS services is relevant for end-users as it enables running the same DS2OS service in different DS2OS sites. In the previous chapter it was shown how the VSL provides portability (Sec. 5.6.3, Sec. 6.2, Sec. 6.3.1).

The VSL-based portability enables the creation of a DS2OS service distribution infrastructure. Such an infrastructure is relevant for end-users as it allows a simple installation of services in Smart Spaces (Sec. 7.6.4). It is also relevant for developers as it allows to create service mash ups via expressing requirements to other context models (services). Such requirements can be resolved by the SLSM when a user installs a new service (Sec. 7.6.4). The S2Store acts as service repository for service distribution.

For end-users context models are transparent. They are only used by services internally to implement their functionality. End-users are typically more interested in the functionality that is offered for their Smart Space by a service than in the back-end of that service [BLM+11, MH12, TPR+12]. But for the development and the portability of services context models are fundamental (Sec. 5.6.4).

The S2Store is the central entity all SLSMs are connected to. The S2Store contains the CMR in DS2OS. Like the CMR (Sec. 6.6.1) it is implemented using regular web server technology enabling the use of known load balancing techniques [GJP10] to
<R.50> make the central component scale (<R.50>).

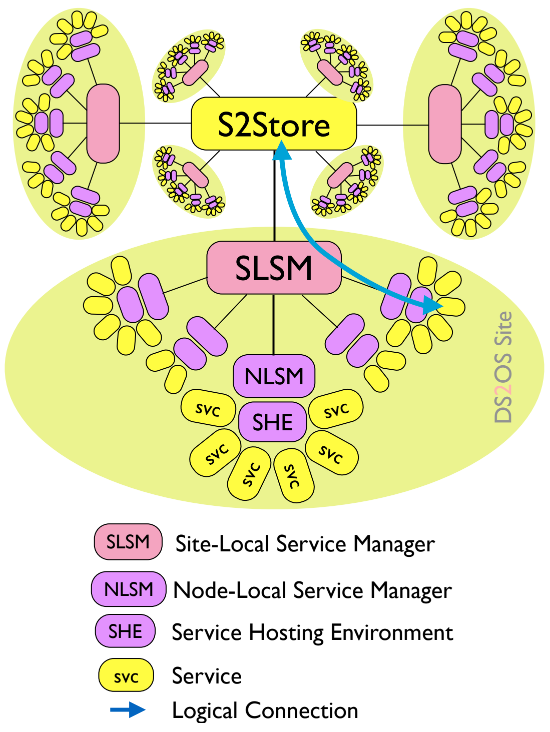

S2Store
SLSM
NLSM
SHE
svc
DS2OS Site
SLSM Site-Local Service Manager
NLSM Node-Local Service Manager
SHE Service Hosting Environment
svc Service
Logical Connection

Figure 7.2: Topology between the App Store and the DS2OS sites.

Fig. 7.2 shows the relation between DS2OS services that run in a DS2OS site and the S2Store.

The blue arrow shows the propagation of service updates. Initiated by the SLSM, a new service packages is loaded from the S2Store. It is initialized by the SLSM using the permissions, the user gave to the previous version of the service before. Then is is transferred to all NLSMs that manage the service locally on the distributed computing node inside a Smart Space.

The other way round usage statistics and failure reports are collected from the NLSMs. They are aggregated by the SLSM in each DS2OS Smart Space. Depending on the permissions given by the site owner, the aggregated reports are sent to the S2Store.

The connection between the S2Store and the SLSM happens via Hyper Text Transfer Protocol (HTTP). The same transport mechanism is used for transferring context models from the CMR to the site-local CMR service (Sec. 6.6.2).

The DS2OS S2Store has the following main functionality:

- Repository for service packageses.

- Context Model Repository (CMR).
- Collector of statistics about services.
- Security anchor for services.

Repository for Service Packages

The S2Store implements a versioning repository of service packages (Sec. 7.2.5). The versioning is done as some versions of a service may be compatible with some context models while others are not. Service compatibility is determined by the interfaces of services that are defined in their context models.

Context models define the availability of functionality in a Smart Space. Some versions of a service may be suitable for some Smart Spaces while others are not. Offering all versions of a service results in the highest compatibility to the diverse DS2OS Smart Spaces. Locally resolvable dependencies are identified by the locally available VSL data types (context models) in a DS2OS Smart Space. Depending on a match of the available services and the required dependencies, expressed in the context model dependencies, the most suitable version of a service in the S2Store is chosen.

Context Model Repository

The S2Store contains the CMR. The service repository functionality and the CMR require a scalable secure infrastructure that is reachable at a well-known location from all DS2OS sites. Combining them brings synergies as only one infrastructure has to be maintained.

Another reason for the combination is that the functionality of the CMR and the S2Store service repository interrelate as shown in Sec. 7.4.

Statistics Collector

The S2Store collects usage and error statistics from the distributed DS2OS Smart Spaces. This is relevant for the crowdsourced convergence (Sec. 7.4.3), and as feedback for developers (Sec. 7.4.2).

Security Anchor

Like the smartphone App stores the S2Store secures the integrity of services as described in Sec. 7.3.

7.3 Security

DS2OS provides a self-managing security architecture (<R.49>). *<R.49>*

Existing App stores typically sign their Apps for ensuring their integrity [App12, And]. They also use developer certificates as described next. As smartphones are *not distributed*, the presented security architecture within a DS2OS site is novel compared to the App economy.

Sec. 5.5 and Sec. 6.5 introduced the security architecture of the VSL that is based on certificates. The DS2OS security architecture explains how the service certificates can be used in a real world deployment.

The following description of the mechanisms gives an overview on the concept. It illustrates how the VSL programming abstraction facilitates the creation of a self-managing security infrastructure for Smart Space services. A detailed security analysis of the approach is part of future work. The concept is presented as security is relevant for a real world deployment of DS2OS [CDB$^+$12, BCG12, CD12, DDMS12].

The presentation of service packages in Sec. 7.2.5 introduced two security relevant supplements to the VSL service certificates: manifests and an additional record in the service certificate that protects the integrity of its corresponding manifest.

Fig. 7.3 shows an extended version of the security architecture of the VSL shown in Fig. 5.4. Instead of one site multiple sites are shown. Each has a different Site Local Certificate Authority (SLCA) (Sec. 5.5) and uses a different key to sign its certificates as indicated by the colored keys. The S2Store also has its own public-private key-pair.

The figure shows that each service has its own certificate (different colors). Inside a DS2OS site, each certificate is signed with the DS2OS site key. As the public key of the site is part of the certificate each entity has, this allows all distributed hosts to verify all site-locally issued certificates (Sec. 5.5). Different to Fig. 5.4, the SLCA is now a regular DS2OS service. This is needed for automated certificate renewal as described in Sec. 7.3.5.

The big stick person in the DS2OS site in the center illustrates how the described security architecture enables end-users to specify the access rights of their services. The goal is giving full control to the users (Sec. 7.6.4) while being fully distributed (Sec. 7.3.3).

7.3.1 VSL Access Control

The security concept of the VSL is based on certificates that contain the *accessIDs* for read and write access of a service (Sec. 5.5, Sec. 6.5). The access control of the KAs uses those values for granting access to context. The *accessIDs* are protected against change by the signature of the local site. Changes to the content of service certificates can only be made by the SLCA.

7.3.2 Three Different Signatures

DS2OS uses three different classes of certificates:

- *Site certificates* are used to securely store metadata to services as described before.
- The *store certificate* is used to protect the integrity of service executables, their meta datas, and the context models that come from the S2Store.
- *Developer certificates* are used by developers to protect their data when submitting DS2OS services to the store. They also have the purpose of non-repudiation, ensuring that a submited service really comes from a certain developer.

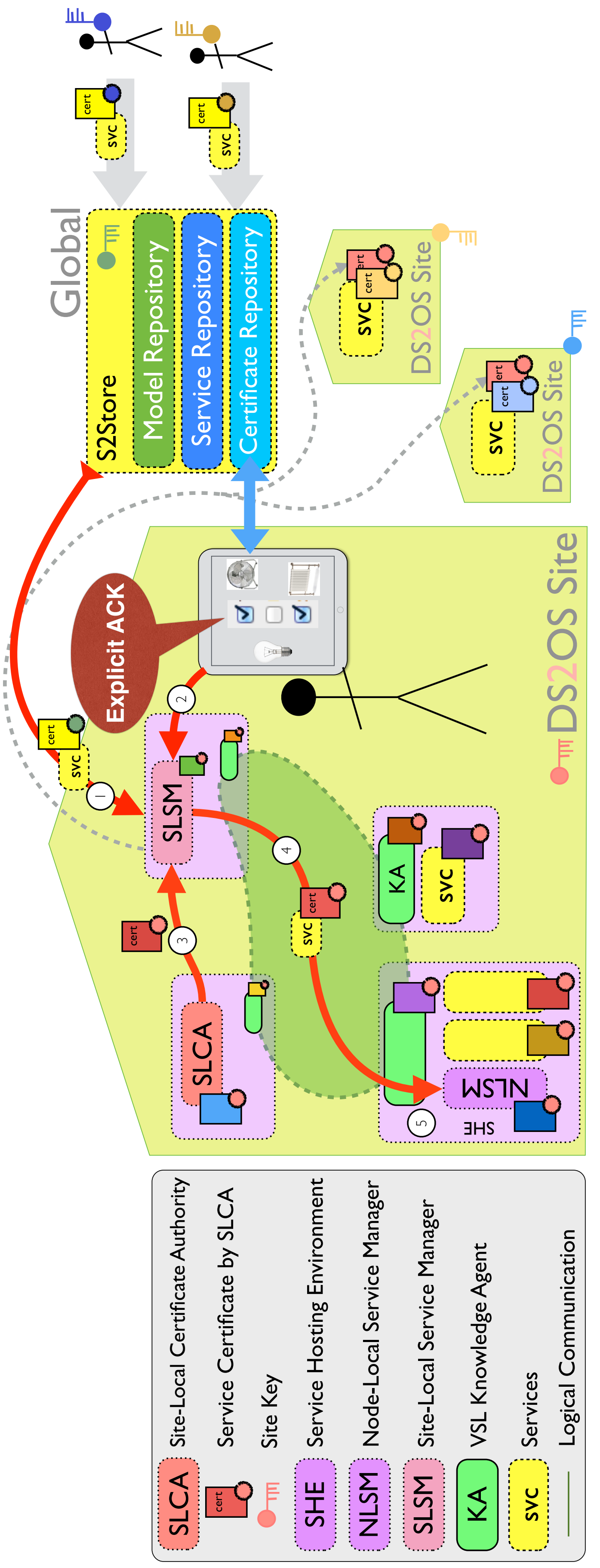


Figure 7.3: Security architecture of DS2OS.

The key pairs for the *site certificates* are generated within a DS2OS site. The key of the site-local SLCA acts as trust anchor for the certificates within a site as described in Sec. 5.5 and Sec. 6.5.

The key pair of the S2Store that is used for creating the *store certificates* is fixed. The public key must be configured at each SLSM. It is used to verify the integrity of the data that comes from the store.

The key pairs for the *developer certificates* are generated at each developer's site. The public key is uploaded to the S2Store on registration. It is used to verify the integrity of the data that comes from a developer. The certificate identifies a submitted service as belonging to a developer (non-repudiation).

The described keying scheme allows to protect DS2OS services during their entire life-cycle (Sec. 7.5). The service life-cycle comprises the development at a developer's site, the S2Store, and a DS2OS site with its SLSM and NLSMs that manage the service locally. The signed hash of the manifest protects the manifest against tamper. The hash over the service executable that is protected in the manifest protects the executable against change.

7.3.3 Access Rights Setting

The access rights for services are critical within a DS2OS site. They determine which context a service can access.

As described in Sec. 5.5 the *accessIDs* within a site certificate of a service determine the access rights of a service. The access rights that are needed to access context nodes are stored within the context models (Sec. 5.5).

Access Groups

For smartphones the amount of peripherals, and thereby the amount of decision possibilities about what an App can access, are limited (Sec. 3.10). Fig. 7.5 shows the available categories of an Apple iPhone that runs iOS 7.

For Smart Spaces there are diverse possibilities for defining such access control groups. The proposed approach of DS2OS is collaborative, and user-based. It is connected to the CMR (Sec. 5.2.8).

Developers can choose identifiers for access groups they want to use in their context models freely. They upload them together with a human understandable description to the S2Store. The S2Store acts as directory for access groups. Via the context models, developers can look up which access groups the context models they want to access use. The access group Identifiers (IDs) used in the relevant context models, a developer wants to interface in its service, determine, which access group IDs the service needs.

As shown in the center of Fig. 7.3, when installing a new service, the user is asked to grant the access rights for a service. Those rights were added by the developer as described in the last paragraph. The *accessIDs* from the service certificate that comes from the S2Store are presented to the user together with their explanatory descriptions that gets loaded from the S2Store's access group repository.

Based-on the description corresponding to the accessIDs from the service certificate, the user decides, which access she wants to grant and which not[31]. As support, the SLCA could show which decisions were taken before or what other users chose for instance[32].

After checking the integrity of the service package (Sec. 7.3.7), and after asking the user for setting the access rights, the SLSM sends the service certificate and the user decision to the SLCA. The SLCA issues a new service certificate based on the certificate from the store. It includes only those accessIDs that were confirmed by the user. This site-local certificate is used by the S2S service management within a site. It is sent back to the SLCA and replaces locally the service certificate from the store.

The original certificate from the store remains in the SLCA for enabling the user to change the access rights later. Similar to the iOS screen shown on the right in Fig. 7.5, the user can see the current access rights she set for a service and change them. A change in the access rights needs a change in the certificate. It is automatically applied as described for the certificate renewal process in Sec. 7.3.5.

The site-local certificates contain only the current access rights that were set by the user. They are used by the services to register to a KA. This implements a centralized access control that scales as the set access rights are verified and applied on each KA independently without contacting a central component.

7.3.4 Distributed Revocation

The VSL uses a decentralized approach for verifying service certificates as described in Sec. 5.5. The advantage of this approach are its scalability, and that it provides high dependability. The solution is tolerant to partitioning in the VSL overlay (Sec. 6.5).

The drawback of using a fully distributed approach is that it makes revocation difficult. In an X.509 infrastructure, revocation lists are distributed from a central server and used for revocation [CSF$^+$08].

Such a centralized approach is not chosen in DS2OS for scalability and dependability reasons. Instead the fact that the certificates are locally issued and can locally be renewed is used for setting short validity periods in certificates. When the validity period of a service certificate has expired, a service cannot use the VSL anymore as described in Sec. 6.5. This implements a distributed time-based revocation.

The *alive pings* are used for synchronizing the VSL time as described in Sec. 6.4.9. This supports timely exclusion of services with invalid certificates on all KAs.

DS2OS implements an automated certificate renewal process that is described in Sec. 7.3.5. As it does not require user intervention to renew its certificates, short validity periods can be configured (e.g. days) resulting in a timely exclusion of unwanted services.

7.3.5 Automated Certificate Renewal

The only required user intervention in the DS2OS security process is setting the desired access rights when installing a new service.

[31]This is an incentive for developers to write good descriptions.

[32]Such mechanisms could enhance the security (majority recommendation) and the usability (recommendations).

When starting a service, the NLSM stores the end of the validity of the certificate of a service similar to the KA (Sec. 6.4.1, Fig. 6.1). At a configured period before that date the NLSM sends the current service certificate to the SLSM.

If the service was not removed by the user, the SLSM requests a new certificate from the SLCA using the current access rights (Sec. 7.3.3). Then the new service certificate is sent back to the NLSM that replaces the old certificate with the new one. When the service re-registers at the KA it uses the new certificate. This results in an automated certificate renewal.

7.3.6 Access Rights Change

The user can change the access rights at any time via a user interface similar to the one shown in Fig. 7.5 on the right. In case of a change of the access rights of a service in the SLCA, the SLCA sends a new certificate with the new access rights to all NLSMs that run the service and asks them to restart the services to reflect the access rights changes.

Updating the access rights may not be successful, e.g. as some NLSMs could not be reached. In such a case, the unchanged, then wrong access rights remain set on those nodes until the service certificates of the services on the node expire (Sec. 7.3.4).

7.3.7 Integrity Check

The described security scheme protects the integrity of the context model, a service uses, of the service manifest, and of the service executable.

The service certificate protects the integrity of the service manifest with its hash over the manifest (Sec. 7.2.5). Via the hash of the modelID and the content of the context model it protects the integrity of the context model (Sec. 5.5). The manifest protects the integrity of the service executable by containing a hash of it.

Before starting a service, the NLSM validates all three, service certificate, context model, and service executable.

7.4 Crowdsourcing

It is the user who should parametrize procedures, not their creators.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

A motivation for the design of the VSL programming abstraction was to facilitate the creation of services for Smart Spaces. Creating Smart Space services should become simple enough to enable distributed user-based, crowdsourced software development (<O.0>, <R.1>, <R.4>, <R.48>).

Ch. 5 introduced the VSL programming abstraction and Ch. 6 presented its implementation as autonomous system. Both facilitates the development of Smart Space services. The DS2OS framework complements the effort by providing a mechanism for securely distributing DS2OS services to distributed DS2OS sites.

In addition DS2OS provides mechanisms that implement a crowdsourced convergence of services, of context models, and of access groups. The convergence of all three is relevant for portability (Sec. 5.6.3, 3.3.2), and for a good usability of the programming abstraction (<R.1>, <R.46>).

Two major reasons for crowdsourced development are the diversity of Smart Devices (<O.1>) and the diversity of pervasive computing scenarios (<O.2>).

The diversity of Smart Devices describes that many Smart Devices are attractive for implementing pervasive computing scenarios. The diversity includes Do-It-Yourself (DIY) devices (Sec. 3.2.3). All those devices need adaptation services (Sec. 8.3.1) for being usable over the VSL.

As discussed in Sec. 3.3, it is unlikely that vendors or other relatively small groups of people are able to provide the required amount of adaptation services. Crowdsourcing is an approach that scales [NMMA13, BCPR09] (<R.50>). *<R.50>*

The diversity of pervasive computing scenarios is known from smartphone Apps that are diverse. Different programmers develop them, introducing their diverse creativity. Sec. 2.5 discussed the creative potential that users have which can be used by enabling crowdsourced development. Enabling crowdsourced development empowers the users to implement their pervasive computing scenarios.

A fundamental problem of crowdsourcing are the *varying quality* of the results, and the *emerging diversity* [NMMA13, GZ13, BCPR09].

The *varying quality* of the contributions is often based on the varying skills of crowdsourced developers. Automated quality control mechanisms such as the validation mechanisms that are implemented for the VSL meta model (Sec. 5.2.14), and those that are proposed for services (Sec. 5.5.6) can help enhancing the quality. Providing a simple-to-use development environment can help to enhance the software quality [NMMA13, GZ13, BCPR09].

The *emerging diversity* of software is wanted for supporting diverse use cases. For abstract interfaces to Smart Devices it is not desired as it hinders portability (Sec. 5.6.4, <R.8>, Sec. 5.6.3). To have unified interfaces (context models) for identical functionality, standardization is needed. *<R.8>*

As discussed in Sec. 3.3.2, typical standardization processes are unsuitable for the described distributed crowdsourced development mainly because of the distribution, the divergence of the participating entities, the time they take, and the high amount of context models that have to be standardized.

As solution to raise the quality of software, to converge context models, and to converge access groups, a crowdsourced convergence mechanism that is based on the S2Store is introduced in this section.

7.4.1 Feedback Collection

To enable the crowdsourced convergence, the S2Store collects automatically generated and personally given feedback from DS2OS sites and users as follows:

- *Service download statistics* are collected.
- *Service usage statistics* such as the amount of time a service was running in a site are collected from the distributed DS2OS sites via the SLSMs if the user agrees.
- *Service error reports* are collected from the distributed DS2OS sites via the SLSMs if the user agrees.
- *User ratings* about services can be sent to the S2Store.
- *Developer ratings* about context models in the CMR in the S2Store can be given by developers.

In addition the context models that are used by services are correlated with the popularity of the services to infer a popularity of context models.

All statistics are publicly available for users and developers.

7.4.2 Crowdsourced Service Convergence

Smartphone App stores typically create different statistics based on the amount of downloads and the user ratings[33]. Such statistics help guiding users as they emphasize popular, and apparently good Apps. This enhances the software quality indirectly as developers aim to get good ratings (e.g. to sell more) [BCPR09, NMMA13].

In DS2OS, The mechanism of smartphone App stores are adapted for reflecting the higher diversity of Smart Spaces.

One important adaption is filtering existing ratings according to the configuration of a specific Smart Space. Only ratings to those services are shown that can run inside the Smart Space, e.g. because of dependencies to other services that are installed or missing.

Another important adaptation refers to quality assurance mechanisms. For a good software quality, service testing and the collection of bug reports on service failures are important. The diversity of Smart Spaces (Sec. 3.2) makes it difficult to run sufficient service tests as each Smart Space configuration is likely to be unique. Emulators can help (Sec. 5.6.7) but still it is difficult to test all Smart Space configurations [BJMS12].

Therefore, DS2OS proposes to collect automated error statistics about services in DS2OS spaces if the user agrees. The distributed SLSMs can file statistics about the status of all locally running DS2OS services to the S2Store. Such statistics include the uptime of a service, the amount of instances running, and errors that happened. This exchange of usage data is described in Sec. 7.2.4.

Developers can use the reports from the participating distributed Smart Space instances for improving their software (Sec. 7.4.5). If the site owner agrees, the configuration of a DS2OS Smart Space is sent as feedback with the crash report. It consists of the identifiers of the available context models. Having this information, developers can test and improve their services in the real context where the error occurred by rebuilding the topology, or by emulating it. If emulators for the reported context

[33]http://www.apple.com/itunes/charts/free-apps/, https://play.google.com/store/apps/top?hl=en

models are available they can be used to simulate the original environment where the crash happened (Sec. 5.6.7). The use of context models makes it transparent to the service to be tested, if the other services are emulated or real.

In addition to the use as direct developer feedback, collecting the error statistics and usage statistics can help providing users with feedback about the reliability and the use of a service which can be used to conclude about the quality and usefulness of a service.

A combination of ratings is expected to support an indirect crowdsourced quality assurance by providing an incentive for developers to develop services that get good subjective ratings by the users, and objective ratings by the automatically collected run time statistics.

It can be combined with the quality approval techniques that are applied by the current big App stores:

- *Manual approval* of Apps in the case of Apple [App10].
- *Automated tests* and reactions to complaints in the case of Google [Goo].

7.4.3 Crowdsourced Context Model Convergence

The convergence of context models is essential for pervasive computing with the VSL as it defines the interoperability of services and thereby the possibilities of service developers to implement their desired pervasive computing scenarios (Sec. 5.6.4, <R.8>, Sec. 5.6.3).

As described in Sec. 5.6.4, it is desired to have ideally one context model per device class (e.g. *lamp*). At the same time, polymorphism is desired as it enables interfacing the same service via different interfaces (see listing 5.2.11). Polymorphism is enabled by multi-inheritance in the VSL (Sec. 5.2.11). Multi-inheritance makes it possible to extend existing context models while remaining compatible with the original interface.

The targeted distributed, user-based development can be expected to result in divergence of context models as reinventing the wheel is a typical phenomenon in software development. It is unlikely that convergence of context models emerges automatically [BCPR09]. As an incentive for convergence, the S2Store correlates the popularity of services with their exposed and used context models (Sec. 7.4.2). The result is a rating of the popularity of context models. Grouping to certain purposes, such as “lighting”, helps finding the relevant statistics on the most popular context models when implementing a new adaptation service for a lamp (Sec. 8.3.1).

If a high amount context models, e.g. 4200, exist for a *lamp*, probably only few of them are used in a significant amount of services installed and running in real Smart Spaces. The reason is that the context models directly correlate with the functionality in a DS2OS site. Rarely available functionality is not so attractive for developers as functionality that can be found and used in many Smart Spaces. In the example of the lamp, a high rating of a certain context model expresses that a service that interfaces this context model is likely able to control lamps in real Smart Spaces. If a context model with low popularity is used, the probability to discover adaptation services within a real Smart Space that use such a context model, and therefore allow controlling a real lamp, is low.

Via the public statistics, developers can identify which *lamp* context models are often used and currently most used in running DS2OS services. It is likely that service developers will choose more popular context models for their implementations as that results in more users being able to run the service in their DS2OS space.

If most services support the *lamp23* context model, it is likely that writing an adaptation service using the context model *lamp23* will be most useful as it can be used with most available services. The other way round, a service for implementing blinking lights should support *lamp23* to be interoperable with most *lamps* in DS2OS Smart Spaces.

The described mechanism implements a crowdsourced convergence of context models. As the context models are the abstract interfaces of DS2OS services, it implements a convergence of abstract interfaces at the same time.

The described CMR is self-managing. The automated deployment of DS2OS services (Sec. 7.2) enables the automated resolving of service dependencies at installation time (Sec. 7.2.5). Both fosters the modular design of DS2OS services (<R.26>). *<R.26>*

7.4.4 Crowdsourced Access Group Convergence

The DS2OS *access group identifiers* (Sec. 7.3.3) are used in the service certificates and the context models. A ranking of the popularity of access group identifiers can be made using the above statistics, and evaluating their popularity in service certificates and context models in the S2Store.

As with the context models, it seems likely that developers of a new context model will support suitable available access group identifiers as they are known to other developers and used by their services already. Using the same access group identifiers for similar functionality exposed by different services results in compatibility. It enhances the probability that a new service is used as it is compatible with a high amount of existing services. This compatibility enables user to install the new service in their Smart Spaces. From a user perspective this is highly desired as it reduces the complexity of the access rights assignment process when existing groups are reused (Sec. 7.3.3).

7.4.5 Developer Feedback

Via the publicly available statistics, developers get feedback on popular context models and on the popularity of their DS2OS services. In addition, error reports are made available to developers for their applications. By monitoring deployed service instances, the participating DS2OS sites become a distributed testbed. The testbed continuously collects debugging information about running services [BJMS12].

The described mechanisms support developers in enhancing their software.

7.5 Life-Cycle of a DS2OS Service

To demonstrate the described functionality of the DS2OS framework, the life-cycle of a DS2OS service is described at the example of a *developer A* who develops an *orchestration service*, and a *developer B* who develops an *adaptation service* (Sec. 8.3.1).

<R.3> The examples show the self-management functionality (<R.3>) from the perspective of developers and users. It simplifies using the S2Store (<R.46>). Fig. 7.4 gives an overview of the presented DS2OS functionality with a focus on the crowdsourcing elements. *<R.46>*

Development

Developer A wants to develop a *lightsOff* DS2OS service that searches for *lamps* in a DS2OS site and switches them off. Such a service could be useful in combination with a *lastOneLeavesTheHouse* service that indicates that nobody is in the DS2OS orchestrated Smart Space anymore.

The developer looks in the S2Store statistics which is the most popular context model for a *lamp* and identifies `lamp23`. She writes her service:

```
import org.ds2os.connector.Connector;
import org.ds2os.exceptions.VslException;

public class LightsOff {
    private final Connector c;

    public LightsOff() throws VslException {
        c = new Connector();
        c.registerService("lightsOff", "lightsOff.crt");
        for (String nextLightInstanceAddress :
                    c.getNodesOfType("/", ".../lamp23")){
          c.set(nextLightInstanceAddress+'/isOn/desired', 0);
        }

    public static void main(String[] args) throws VslException {
        LightsOff mylightsOffInstance = new LightsOff();
    }
}
```

Listing 7.3: A DS2OS service that switches all lamps of type `lamp23` off.

Listing 7.5 contains the entire code that provides the described functionality using the VSL. It illustrates how the VSL programming abstraction facilitates the development of Smart Space services (<R.1>, Ch. 8). *<R.1>*

The developer creates the *service manifest* (Sec. 7.2.5) that requests access for the group `lamp23access`[34] that is used in the `lamp23` context model:

```
<lamp23>
  <isOn type=".../isOn42" reader="lamp23access">
    0
    <desired reader="lamp23access" writer="lamp23access">
      0
    </desired>
  </isOn>
</lamp23>
```

Listing 7.4: VSL context model `lamp23`.

Next, the developer creates the service certificate using her developer key (see Fig. 7.3) and creates a ZIP container[35]. Last she uploads the resulting service package to the S2Store together with the context model of the `lightsOff` service:

[34]As expressed with the name, the access group `lamp23access` arbiters the access to instances of `lamp23`.

[35]When implemented, developer tools will be provided for this process.

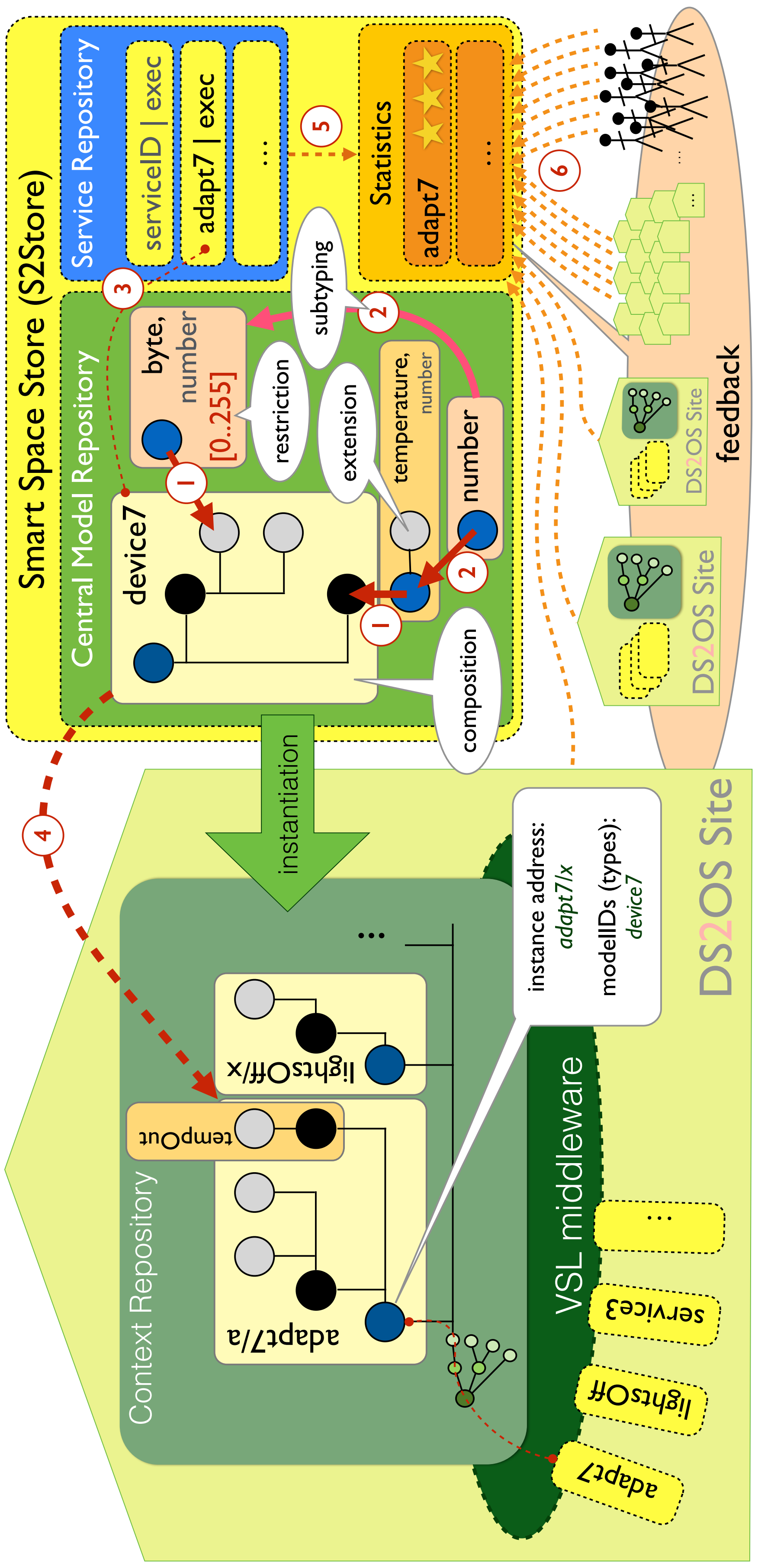


Figure 7.4: Overview of the DS2OS functionality with a focus on the crowdsourcing support.

```
<lightsOff>
  <allOff type="../lightsOffSwitch" writer="lamp23access">
    0
    <desired reader="lamp23access" writer="lamp23access">
      0
    </desired>
  </allOff>
</lightsOff>
```

Listing 7.5: VSL context model `lightsOff`.

Listing 7.5 shows that the access rights for the new service are identical to those of `lamp23`. This makes sense as the service provides additional functionality for the users of this context model. In the DS2OS site on the left in Fig. 7.4, an instance of the *lightsOff service* is shown.

Developer B wants to develop an *adaptation service* (VSL Smart Device driver, Sec. 8.3.1) for a self-made Smart Device "*device7*" that offers different functionality including a temperature sensor (Sec. 9.6.4). First he creates the context model using existing context models from the CMR.

The center of Fig. 7.4 graphically shows how the context model for `device7` inherits different existing types that are defined as context models in the CMR and identified by their *modelIDs*. The object oriented methods *composition* (1), and *subtyping* (2) are shown as described in Sec. 5.2.

Then *developer B* develops the corresponding adaptation service, `adapt7` as described in Sec. 8.3.1. He creates the *service certificate* and the *service manifest* that link the `adapt7` executable with the context model `device7` (3). He includes the security mechanisms described in Sec. 7.3. Finally he uploads the resulting `adapt7` *service package* (Sec. 7.2.5) to the S2Store.

At the S2Store

The S2Store receives the service packages and the context models. It verifies both using the public keys of the developers that are stored in the S2Store. The CMR validates the context models (Sec. 5.2.14) before storing.

After a successful validation, the *service certificates* are signed with the S2Store's public key. The services and the context models are stored. They can be obtained from the S2Store now.

Deployment to a DS2OS Site

A user finds the tutorial to build his own `device7` in the Internet, builds the device, and looks for the *adaptation service* in the S2Store. As the S2Store interface recommends the *lightsOff service*, the user clicks on install for both services on the SLSM interface (Sec. 7.6.4).

The SLSM loads the *service packages* (Sec. 7.2.5) from the S2Store using a HTTP connection. It verifies the service certificates with the store's public key that it has preconfigured. Then it presents the requested *accessIDs* (Sec. 5.5.1) to the user.

The user allows the access to the `lamp23access` group for the *lightsOff service* with a click. The *adapt7 service* does not require access permissions. The SLSM sends the *service certificates* and the user's *access rights decision* to the SLCA that creates a *locally signed service certificate*, and sends it back to the SLSM (Sec. 7.3.3, Fig. 7.3).

The S2Store keeps track of the downloaded services as equired for the crowdsourced convergence (Sec. 7.4). See (5) in Fig. 7.4.

Starting the new Services

To start the new services, the user simply clicks on the play button next to the services in his list of installed services in his SLSM user interface (Sec. 7.6.4). The SLSM applies its placement strategy using the statistics it collected via the NLSMs from the distributed computing nodes.

It decides to deploy the *lightsOff service* to the computing node `node42`. It sends the service package including the site-local service certificate to the NLSM running on `node42` (Sec. 7.2.4). The *adapt7 service* is placed on the same node.

The NLSM verifies the integrity of the *service certificates* with the *site's public key* from its own *service certificate*. Then it verifies the *service manifests* with the hash in the *service certificates*. Finally it verifies the *service executables* via the hash values in the *service manifests*. Then it starts the *lightsOff service* and the *adapt7 service* via the SHE (Sec. 7.2.3).

The *lightsOff service* registers at the KA on `node42`. The KA obtains the context model `lightsOff` shown in listing 7.5 via the CMR service (Sec. 6.6.2). It checks the integrity of the context model[36] via the *signed hash* in the *service certificate* and loads it into the local context repository under the address `/node42/lightsOff/x` as shown in Fig. 7.4. The same happens for the context model `device7` for the *adapt7 service*. Its context repository gets the address `/node42/adapt7/a`.

Now both services are running and can be used. Fig. 7.4 shows the situation in the DS2OS site on the left. For better readability, the details of the VSL as distributed KAs are not shown (Fig. 7.1). Instead, the VSL is shown from a service's perspective, as ambient context overlay with a unified interface.

Running the new Services

During the run time of the services other services can discover the instances by searching for the type "`lightsOff`" (listing 7.5) for the *lightsOff* service, and the type "`device7`" for the *adapt7* services.

As described before, the results of the queries will be "`/node42/lightsOff/x`", and "`/node42/adapt7/a`" in the example. Other services can invoke the *lightsOff* service by calling the VSL function:

```
set /node42/lightsOff/x/allOff/desired 1
```

After the trigger is set, the *lightsOff* service searches for *lamp23* instances

```
get /search/type/?/.../lamp23
```

[36]The hash is over the self-contained context model that has all dependencies resolved already. This is done to ensure that a loaded context model (e.g. `/derived/boolean`) is not manipulated.

Assuming the instance addresses are `/node82/lampAdaptMe7/d`, `/node82/lampAdaptMe1/p`, `/node50/lampAdaptMe6/c`, `/node47/lampAdaptMe3/w`, and `/node79/lampAdaptMe5/m` it issues the following commands to switch the lamps off:

```
set /node82/lampAdaptMe7/d/isOn/desired 0
set /node82/lampAdaptMe1/p/isOn/desired 0
set /node50/lampAdaptMe6/c/isOn/desired 0
set /node47/lampAdaptMe3/w/isOn/desired 0
set /node79/lampAdaptMe5/m/isOn/desired 0
```

The autonomic adaptation services reflect the desired value to the reality then (Sec. 8.3.1), and the lamps are switched off.

The temperature from the *device7* can be obtained by issuing:

```
get /node42/adapt7/a/temperature/tempOut
```

Monitoring

Services are monitored while they run by the NLSM. The SLSM periodically collects reports from the NLSMs and sends them to the S2Store if the site owner allows this (Sec. 7.2.6). See Fig. 7.2 and Fig. 7.1.

Other DS2OS sites that are running the services contribute feedback according to their configuration too. This is shown at the bottom center of Fig. 7.4 (6).

Feedback

Liking the *lightsOff service*, the user gives a five star rating and a nice comment at the S2Store via the SLSM interface (Sec. 7.6.4). The user experiences has some problems using the *adapt7 service*. Therefore, it gets a four star rating.

Error reports from the distributed DS2OS sites are sent to the developers automatically if the user agrees.

The S2Store receives different information about locally running services from participating SLSM instances in distributed Smart Spaces. From all received, the S2Store computes different statistics about the services, including the most used and liked context models, access identifiers, and services. The statistics are made available to users and developers as described in Sec. 7.4.1.

The statistics are symbolically shown on the right of Fig. 7.4. The shown three star rating of the *adapt7 service* could follow from a metric that combines different of the collected feedback values, including the four star user rating given above.

Update

Developer A uploads an update of the `lightsOff` service to the S2Store. The new version is automatically detected by the SLSM based on the version number in the service manifest. Depending on the configuration of the DS2OS Smart Space, the update is automatically installed as described in Sec. 7.2.4, or the user is notified about its availability and can install the new version manually.

7.6 Resulting Properties

This section discusses the properties *autonomy*, *dependability*, *scalability*, and *usability* of the presented DS2OS framework.

7.6.1 Autonomy

<R.3> The S2S service management framework is fully self-managing (<R.3>):

- The SLSM applies its optimization strategies autonomously (Sec. 7.2.4).
- The NLSM manages all node-local services autonomously (Sec. 7.2.3).
- The security is self-managing as described in Sec. 7.3.

The user can control the SLSM by installing new services, managing the access rights of services, and starting and stopping services. For those operations, user interfaces similar to those shown in Sec. 7.6.4 are sufficient. For more experienced users more advanced interfaces could be interesting, e.g. for showing the current load of computing nodes.

7.6.2 Scalability

<R.50> DS2OS is designed as scalable architecture (<R.50>). It operates fully distributed. The SLSM distributes services to the distributed NLSMs that operate autonomously (Sec. 7.2.3).

The SLSM caches all locally installed services. This makes it partly independent of the S2Store similar to the caching strategy of the context models (Sec. 6.6.2).

The SLSM and the SLCA are potential bottlenecks. The load of the SLSM depends on the amount of NLSMs, and on the optimization strategies it implements. The hierarchical delegation of management between the SLSM, the NLSM, and the SHE improves the scalability. The SLCA is only needed for issuing and renewing service certificates (Sec. 7.3). Its load can be expected to be low.

The S2Store is implemented as a regular web server which makes known optimization strategies applicable [GJP10].

Via the load balancing strategies that are implemented by the SLSM the scalability of services can be improved. Services that need lots of resources can be migrated to computing nodes that offer such resources. The specification of resource requirements in the service manifests provides a mechanism to communicate such needs.

<R.50> Via the described mechanisms, DS2OS provides scalability (<R.50>).

7.6.3 Dependability

<R.30> Different features enhance the dependability of DS2OS (<R.30>). The most important ones are:

- The NLSMs manage all services on their computing node independent of the availability of other components (Sec. 7.2.3).
- The SLSM uses the type search for discovering NLSMs at run time (Sec. 5.7.2). The use of the underlying locator-id split (Sec. 5.2.13) makes topology changes in the VSL transparent to the SLSM and other services.
- The SLSM pulls information from the NLSM, implementing a monitoring of the state of the VSL.
- The certificate-based authorization works fully distributed. It is independent of the availability of DS2OS components (Sec. 6.5).
- All services that build the DS2OS framework act autonomously (Sec. 7.6.1).

The previous features enhance the dependability of the DS2OS backbone. Services are continuously monitored via the NLSMs and the SLSM. This allows to run counter measurements in case of service failures, e.g. by restarting the failed service. This dependability of services at run time. The early migration (Sec. 7.2.4) of services according to the requirements in the service manifest and the optimization strategies of the SLSM enhance the dependability further. They prevent that computation nodes get overloaded.

The feedback mechanisms of the hierarchical service management (Sec. 7.4.1) in combination with the validation mechanisms of the VSL (Sec. 5.2.14) are expected to enhance the quality of service implementations. Such automated mechanisms support crowdsourced development as they support each developer automatically (<R.48>). *<R.48>*

The described DS2OS mechanisms enhance the dependability of software orchestrated Smart Spaces (<R.30>). *<R.30>*

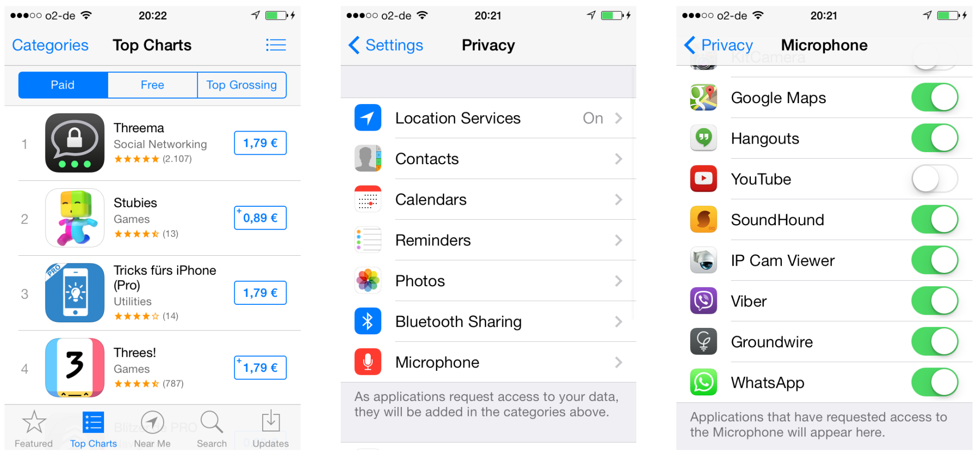
7.6.4 Usability

As described in Sec. 7.6.1, DS2OS is self-managing. This is necessary to enable real world use by non experts (<O.4>). Following, a simple end-user interface is presented that could foster real world deployment of DS2OS Smart Spaces (<R.1>). It shows how the S2Store is simple-to-use (<R.46>). *<R.1>* *<R.46>*

With the methods that were introduced in this chapter, an interface that is as simple as the interface of a smartphone can be implemented to control a Smart Space. As DS2OS is not implemented in its entirety, screenshots of an Apple iPhone running iOS 7 are used to illustrate a possible user interface for DS2OS.

Fig. 7.5 shows the user interface of iOS 7. The left screenshot shows the local App manager (Sec. 7.1.1) that is used to browse the App store.

DS2OS can offer such a presentation to browse DS2OS services. Smart Spaces are more complicated than smartphones. They have (more) service dependencies (Sec. 7.2.5). Therefore the S2Store interface must take into account, which context is available in the Smart Space of the current user. The available context models represent which services are already running in the Smart Space instance. Indirectly it also represents, which hardware is available (e.g. a lamp).


●●●○○ o2-de 20:22
Categories Top Charts
Paid | Free | Top Grossing
1 Threema, Social Networking ★★★★★ (2.107) 1,79 €
2 Stubies, Games ★★★★★ (13) 0,89 €
3 Tricks fürs iPhone (Pro), Utilities ★★★★☆ (14) 1,79 €
4 Threes!, Games ★★★★★ (787) 1,79 €
Featured · Top Charts · Near Me · Search · Updates

●●●○○ o2-de 20:21
Settings Privacy
Location Services On
Contacts
Calendars
Reminders
Photos
Bluetooth Sharing
Microphone
As applications request access to your data, they will be added in the categories above.

●●●○○ o2-de 20:21
Privacy Microphone
Google Maps: on
Hangouts: on
YouTube: off
SoundHound: on
IP Cam Viewer: on
Viber: on
Groundwire: on
WhatsApp: on
Applications that have requested access to the Microphone will appear here.

Figure 7.5: User interface of iOS7 for browsing Apps from the App store (left), selecting an access category (center), and granting access rights to Apps (right).

Missing dependencies can often be automatically resolved by installing additional services that provide the missing context. When this is not possible, e.g. as hardware is missing, the S2Store interface can propose to users to buy missing hardware, e.g. controllable lamps. Such proposals could act as an enabler for a Smart Device industry (Sec. 10.4).

The interface on the left can be used to start and stop services, or to monitor the state of services within a DS2OS site. For that purpose, a simple play and pause button could be added. Pressing it would communicate to the SLSM to run a service.

The second screen shows the access rights categories. As described in Sec. 7.3.3, a similar list can be shown for DS2OS Smart Spaces as the SLSM knows about all services it deployed to the NLSMs and about their access groups. Next to each access group the explanatory text from the S2Store can be shown for making users understand the corresponding right.

The third screen shows the Apps that asked for permission for a certain access group, and their current access rights. This screen could also show the current expiry dates of the DS2OS service certificates to inform the user until when the rights are granted at most (Sec. 7.3.4). A change in the access rights can be reflected to a service as described in Sec. 7.3.6.

This chapter showed why such a simple interface is sufficient for future Smart Spaces. As the same interface apparently works on smartphones [vE13], it can be expected
<R.1> that it is usable by non expert users in Smart Spaces as well (<R.1>, <R.46>).

<R.46> The usability from the perspective of a developer was outlined in Sec. 7.5. This entire thesis is about supporting developers by structuring and facilitating the development of services that implement pervasive computing scenarios.

7.7 Assessment

In Sec. 4.5 different state of the art pervasive computing middleware was assessed. The assessment criteria of Sec. 4.5.1 are used in this section to assess the DS2OS middleware framework. The result of the assessment is shown in table. 4.1.

7.7.1 Evaluation of DS2OS

The Distributed Smart Space Orchestration System (DS2OS) aims to foster the implementation of pervasive computing in the real world. To reach the goal, it offers functionality to structure and facilitate the development of pervasive computing applications at large (<O.0>).

DS2OS is based on the VSL programming abstraction that enables the transparent coupling of distributed services via context models (Sec. 5.4).

System Design & Architecture

DS2OS consists of

- the self-managing VSL μ-middleware (Ch. 6),
- the S2S service management framework (Sec. 7.2), and
- the S2Store (Sec. 7.2.6) that comprises the CMR (Sec. 5.2.8), a repository for service executables, and as information exchange platform between developers and users (Sec. 7.4).

The VSL middleware consists of distributed peers, the Knowledge Agents (KAs). The KAs use Extensible Markup Language (XML)-Remote Procedure Call (RPC) and HTTP for communication (Sec. 6.3.2). Via so-called connectors other communication protocols can be interfaced (Sec. 6.3.3).

The VSL is implemented in Java (Sec. 6.3.1). Java allows the execution of the KAs on heterogeneous computing hosts.

The VSL is implemented as a system of distributed peers, the KAs. They provide distributed context repositories and manage context transparently and autonomously (Sec. 6.4). The VSL provides a vertical organization with horizontal communication between the distributed KA peers of the middleware (Ch. 5).

Context Management

The VSL implements a hybrid meta model (Sec. 5.2). It consists of self-contained tuples that are logically structured to trees. During the instantiation of context models, the VSL meta model supports object oriented principles (multi-inheritance). Context models are represented in the XML markup language.

DS2OS does not differentiate between services (Sec. 5.4). All kinds of services can be added and removed at run time – including those services that connect Smart Devices to the VSL (Sec. 6.2, Sec. 8.3.1). DS2OS uses a Context Model Repository (CMR) for distributing context models. The CMR is open for submissions allowing the dynamic extension at run time (Sec. 7.2.6).

DS2OS provides crowdsourced convergence mechanisms for context models (Sec. 7.4). Via the VSL concept Virtual Context, the semantics of the VSL can be extended dynamically at run time (e.g. with location dependencies, Sec. 5.7.3).

The VSL puts historic context automatically under version control (Sec. 5.3.2).

Service Management

The KA provide a fixed Application Programming Interface (API) of 13 commands (Sec. 5.3.1). Using a fixed API is possible as the VSL uses context models as abstract service interfaces (Sec. 5.4.1).

All DS2OS services communicate over the VSL using its fixed API. This makes all DS2OS services inherently interface compatible and composable (Sec. 5.4.4).

The VSL supports all forms of inter-process communication. Direct communication is supported via Virtual Nodes. Message-based-, group-, and generative communication are supported via regular context nodes (Sec. 5.4.3).

The VSL only provides fundamental context management and inter-service communication mechanisms including context storage, context routing, access control, and encryption. In contrast to all related work (Sec. 4.5), the VSL can be dynamically extended. Functionality can be added at run time in a way that makes it transparent to services whether a functionality is provided by the VSL or a service. This makes the VSL a μ-middleware (Sec. 6.2).

DS2OS provides autonomous service-management including the provisioning of services from the S2Store to a local Smart Space, service authentication, the placing and deployment of services on available distributed computing nodes, and resource usage optimization via service migration (Sec. 7.2.4).

Real World Deployment Support

The VSL and DS2OS are fully distributed and partition tolerant (Sec. 6.3). Context tupleses are self-contained (Sec. 5.2.4). In contrast to typical ontologies (Sec. 3.9.1), this allows a fast context processing.

The single central component of DS2OS, the S2Store, is only needed to deploy new services. It is implemented as web sites which allows applying existing mechanisms to ensure scalability [GJP10] (7.2.6). In addition, DS2OS sites apply caching for services and context models making them independent from the S2Store (Sec. 6.6.2). This makes DS2OS scale. The SLSM and the SLCA could be distributed in a DS2OS site. As discussed in Sec. 7.6.2, they are designed to scale.

DS2OS provides several security mechanisms that are based on certificates. The integrity of services and context models can be verified at any point of the life-cycle of a service [**?**], from the developer to the computing node that uses them (Sec. 7.3). Communication in the Peer-to-Peer (P2P) network is encrypted (Sec. 6.5).

In addition, context models and context values are validated using the VSL type system (Sec. 5.2.14).

DS2OS supports user-based development with several mechanisms including the following. The VSL programming abstraction, implemented in the KAs, offers a fixed API of 13 functions (Sec. 5.3). The service interfaces are descriptive (5.4). Context models support modularization (Sec. 5.2.11). Modularization of services is fostered via the VSL SOA (Sec. 5.4.4). Bug tracking is facilitated by the automated validation of context (Sec. 5.2.14), and different feedback mechanisms for developers that the S2Store offers (Sec. 7.4.1). DS2OS operates autonomously (Sec. 7.6.1).

7.7.2 Comparison to the Combined Reference Model

The μ-middleware property of the VSL (Sec. 6.2) results in a design of DS2OS that is fundamentally different to the designs of the state of the art. The pervasive computing middleware that was assessed in Sec. 4.5 could be represented by the combined reference architecture that is shown in Fig. 4.3.

The combined reference model from Sec. 4.5 is unsuitable for DS2OS as its VSL middleware part only contains basic functionality (μ-middleware). Therefore the functionality that is shown in Fig. 4.3 is transferred to the model of the DS2OS architecture that is shown in Fig. 7.6. For comparability, the numbers, the terms, and the colors are identical in the two figures.

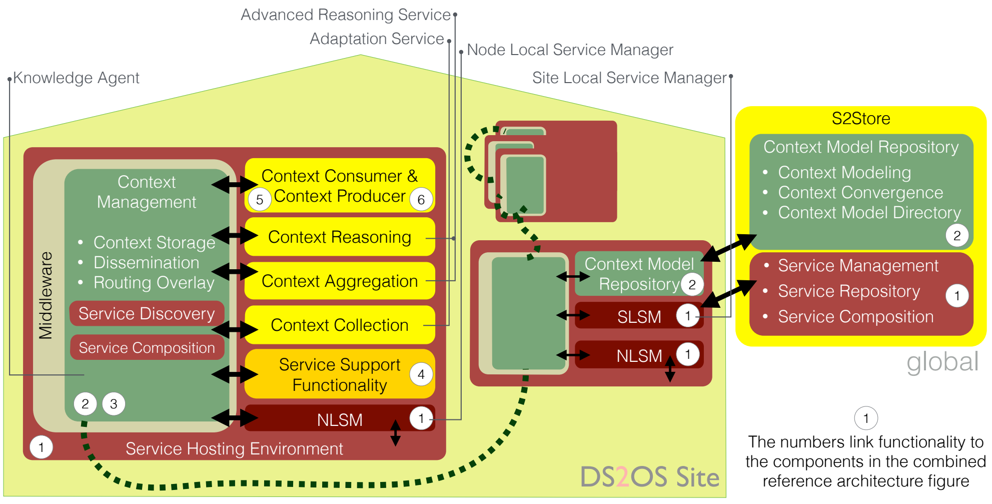


Figure 7.6: The architecture of DS2OS using the components, colors, and numbers from the combined reference architecture in Fig. 4.3 for comparison.

Following the μ-middleware-based DS2OS model is presented. Differences to the combined reference architecture are discussed (Fig. 4.3).

Context Management

The VSL middleware on the left of Fig. 7.6 provides the functionality for *context storage* (2), *context dissemination* (3), and *context routing* (3) (Ch. 5, Ch. 6). For deploying context models from the CMR to a DS2OS site, the CMR service is used (Sec. 6.6.2).

In addition to the functionality that is typically offered by a context-provisioning middleware, the VSL provides the functionality for *service discovery* (1) (Sec. 6.7), and *service composition* (1) (Sec. 5.4.3).

The *service discovery* (4) is identical to the *context discovery* (3) as the VSL uses context models a abstract service interfaces (Sec. 5.4.1). The *context modeling* (2) is moved into the S2Store (Sec. 5.2.8). *Service composition* is implemented by using the VSL for inter-service communication (Sec. 5.4). The CMR acts as directory for abstract service interfaces that can be used for composition (Sec. 7.5).

Service Management

The service management functionality is distributed. The basis is a component that implements an additional middleware layer, the SHE (Sec. 7.2.2). Besides the VSL, the DS2OS framework participates in *service composition*. The service certificates in the S2Store contain the access groups that are distributed over the S2Store (Sec. 7.2.5). They define which services can couple (Sec. 5.5).

The functionality for *service hand-off* (1), and *service migration* (1) are handled co-operatively by the SLSM and the NLSMs services (Sec. 7.2.4).

Service monitoring (1) is done by the NLSMs and propagated to the SLSM, and the S2Store (Fig. 7.2, Sec. 7.2.3).

The *service repository* (1) is in the S2Store. The SLSM acts as local repository for all installed services of a DS2OS site (Sec. 7.2.4).

Additional Functionality

The functionality for *context reasoning* (2) (Sec. 5.6.6), *context aggregation* (2), and *context collection* (2) is moved from the middleware in the combined reference model into DS2OS services in Fig. 7.6 (Ch. 8).

All DS2OS services are *context producer* (5) and *context consumer* (6) as context is used for inter-service communication (Sec. 5.4.1).

Potential *service support functionality* (4) is moved into DS2OS services. This includes services that extend the semantics of the VSL (Sec. 5.7.3).

7.8 Chapter Conclusion

The DS2OS framework implements distributed service management. Sec. 7.5 summarizes the major functionality at the example of a service life cycle.

The VSL is mainly a *context-provisioning* middleware. DS2OS shows how it can be used to implement most of the functionality provided by the assessed *service managing* middleware (Sec. 4.5). See Sec. 7.7.

DS2OS illustrates how the μ-middleware property of the VSL allows transparent extension. The VSL can be extended with services to provide the functionality of the state-of-the-art pervasive computing middleware.

The chapter shows how the VSL programming abstraction facilitates the creation of a distributed service management framework. Though the described services provide complex functionality (Sec. 7.2), the implementation of each is typically a few hundred lines of code (listing 7.5).

Such code compactification is only enabled by the use of the VSL programming abstraction that allows developers to focus on the logic while the VSL manages the state and the context exchange (Sec. 5.6.5).

Besides demonstrating the use of the VSL programming abstraction, DS2OS provides relevant functionality for real world deployment of pervasive computing. As described in Sec. 7.6, the presented solution is fully autonomous (Sec. 7.6.1), provides security (Sec. 7.3), scalability (Sec. 7.6.2), and dependability (Sec. 7.6.3).

It supports strong mobility of services, and it supports request forwarding via proxies resulting in seamless context delivery from the perspective of a context consumer (Sec. 7.2.4).

The introduced collaborative access groups and the convergence mechanism for context models, services, and access groups are probably novel as they could not be found in literature (Sec. 7.3.3, Sec. 7.4).

The collaborative convergence mechanisms Sec. 7.4) complement the collaborative standardization that was presented in Sec. 5.6.4. The added metrics support the convergence. It is fundamentally important to provide portability via convergence of abstract interfaces.

Without a convergence of the context model (abstract service interfaces) the heterogeneity of Smart Devices is only lifted up to a higher layer of abstraction as discussed in Sec. 5.6.4.

Sec. 7.6.4 gives an impression how the end-user interface of a DS2OS Smart Space may look like in the future. This chapter showed how DS2OS hides the complexity from the end-user. As the same interfaces proved their usability on smartphones [vE13], it is likely that such interfaces make pervasive computing usable in the real world.

Another outcome of the chapter is that the VSL programming abstraction simplifies the development of complex services for Smart Spaces significantly. This can be expected to enhance the dependability of future DS2OS operated Smart Spaces (<R.30>). The dependability includes security (<R.49>, Sec. 7.3).

The presented DS2OS framework provides the technical basis for an App economy for Smart Space (table 3.4). The App economy for smartphones that implemented real world ubiquity of mobile computing. DS2OS shows that a similar concept is possible despite the increased complexity of Smart Spaces compared to smartphones (Sec. 3.10.9).

8. Smart Space Services

A programming language is low level when its programs require attention to the irrelevant.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

Short Summary This chapter demonstrates how the Virtual State Layer (VSL) programming abstraction facilitates the implementation of services. Implementations of six relevant service classes demonstrate the practical use of the VSL programming abstraction.
As additional structuring element for Java services, a service template is introduced. It is shown how the VSL abstraction allows automated integration of Smart Devices, automated correlation of context on different levels of abstraction, remote access, generic user interfaces, and the implementation of complex scenarios using simple-to-configure Event-Condition-Action (ECA) rules.

Key Results The VSL programming abstraction structures and facilitates the creation of services for Smart Spaces.
Smart Devices can automatically be integrated into VSL operated Smart Spaces.
Cascaded services can be used for transparently connecting context on different levels of abstraction.
Remote access with full local control can be implemented using the VSL coupling and security mechanisms.
The VSL abstraction enables the use of a single generic interface for accessing all functionality in a software orchestrated Smart Space.
Using the VSL with the presented mechanisms enables the implementation of complex pervasive computing scenarios with the simple mechanism ECA rules.

Key Contributions A concept for *automated integration of new Smart Devices* into a software orchestrated Smart Space is introduced. It includes automated context model generation, and collaborative creation of adaptation services.

An architecture for *automated creation of context on different levels* of abstraction is introduced.
A fully transparent service coupling mechanism between Smart Spaces that gives local control about the remote access is introduced.
An *interface with multi-user support* that can be used to *access all functionality within a Smart Space in a uniform way* is introduced. The VSL architecture enables offering such an interface that can be used by all Smart Space services as it can be customized.
Building blocks that enable the *implementation of complex pervasive computing scenarios with simple ECA rules* are introduced.

Challenges Addressed <R.1> *Simplicity-to-use*, <R.2> *Multi-user support*, <R.4> *User participation*, <R.9> *Information loss* prevention, <R.23> *Dynamic extensibility*, <R.26> *Modularization support*, <R.30> *Dependability*, <R.37> *Service statelessness*, <R.43> *Dynamic extensibility*, <R.47> *Emulator* support, <R.49> *Security-by-design*, <R.50> *Scalability*.

Summary This chapter shows how the VSL programming abstraction facilitates the implementation of diverse classes of services in real world scenarios.
The chapter starts with the introduction of a service template that can be used to structure the implementation of Distributed Smart Space Orchestration System (DS2OS) services. It shows the practical use of the VSL programming abstraction for implementing services in Java.
Then examples from six different service classes are introduced to illustrate the support for diverse scenarios that is provided by the VSL.
It is shown how the connection of Smart Device to the VSL is supported by *adaptation services*. In addition, it is shown how their modular design enables a crowdsourced creation of drivers for formerly unsupported hardware.
At the example of an *advanced reasoning service* it is shown how the VSL supports the creation of context on different levels of abstraction, and how this facilitates the development of service and enhances their inter-operability.
The full encapsulation of services behind their abstract service interfaces (context models) enables the creation of transparent emulators. This is introduced.
The connection of two DS2OS Smart Spaces is introduced.
A generic *user interface* that can be used for all services in a DS2OS space is presented.
Finally it is shown how the use of a descriptive abstraction fosters the implementation of complex functionality without programming by parameterizing ECA rules.

8.1 Introduction

The Virtual State Layer (VSL) programming abstraction uses context models to structure the interaction between Smart Space services. The mandatory use of a structured interface helps structuring services. In Sec. 5.6.5 a separation of service logic from service state was proposed. It helps facilitating services further as it allows developers to focus on the logic of the pervasive computing scenarios they have in mind while the VSL handles the necessary context storage and context brokering functionality.

The Distributed Smart Space Orchestration System (DS2OS) provides mechanisms for managing services within a Smart Space (Smart Space Service management (S2S)), and to distribute services to Smart Spaces (Smart Space Store (S2Store)). It provides an infrastructure for crowdsourced development (Sec. 7.4).

This chapter introduces best practices in DS2OS service development, and several example services that are relevant for real world deployment (<O.4>). First a *service template* is introduced that provides programmers with a structure for modularizing their services. Then a classification of Smart Space services is given. It helps modularizing Smart Space functionality into different services.

The service classification is illustrated at the example of different services that show how the VSL programming abstraction facilitates the creation of applications with diverse functionality. As the VSL fosters a Service Oriented Architecture (SOA) (Sec. 5.4.4), an example for implementing complex services via service mash ups is given. The example shows how the use of a descriptive interface (context models) enables the use of the simple mechanism Event-Condition-Action (ECA) (Sec. 3.6.4) to implement complex functionality.

8.2 Wiring and Logic

The VSL programming abstraction offers two mechanisms for service coupling:

- Subscriptions.
- Virtual Nodes.

Both are handled via callbacks. The abstract interface of the subscription callback is shown in listing 5.3. The abstract interface of the Virtual Node callback is shown in listing 5.5.

The context model of a service can be compared to the header file in the C programming language. It provides the abstract service interface.

The next functionality a VSL service needs is the initialization of the service. It consists of registering at a Knowledge Agent (KA) and registering the callbacks.

The third functional part of a DS2OS service is the application logic.

Fig. 8.1 proposes a modularization of program code into the functional areas described above. The connector provides the Java interface to the VSL. See listing 6.1.

The `ServiceWiring.java` class is intended to contain the initialization functionality described above (listing 8.1). It calls the connector method to register the service, and it registers the callbacks. The body of the callbacks points to the `ServiceHandler.java` class that contains the application logic (listing 8.2).

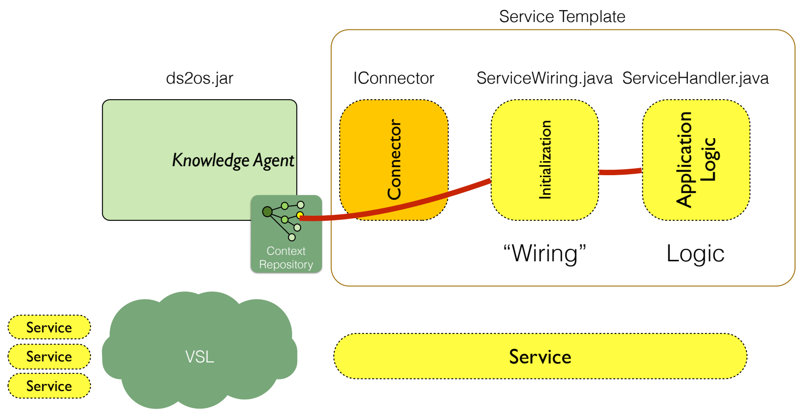


Figure 8.1: The service template that was used for the user study.

Greet and Tickle Service

The aim of this chapter is illustrating the practical use of the VSL programming abstraction. Therefore an example is shown.

The presented service is called *greet and tickle* since it offers this functionality. It introduces developers to the VSL Application Programming Interface (API) (Sec. 5.3.1).

The service offers the following context nodes:

```
<demoService type="/services/template/demoService,/basic/container,/system/service" version="0" timeStamp="2014-02-24 14:19:02.653" access="rw">
   <greet type="/basic/text,/system/virtualNode" version="0" timeStamp="2014-02-24 14:19:02.688" access="rw" />
   <isLaughing type="/derived/boolean,/basic/number" version="1" timeStamp="2014-02-24 14:19:02.878" access="rw" />
   <tickle type="/derived/boolean,/basic/number" version="0" timeStamp="2014-02-24 14:19:02.705" access="rw" />
</demoService>
```

Setting the Virtual Node `greet` node by issuing `set greet Marc-Oliver`, the *Java* console outputs "Hi Marc-Oliver!". Using Virtual Subtrees (`set greet/Marc-Oliver VSL`) two greetings are returned in the *Java* console:

```
Hi Marc-Oliver!
Hi VSL!
```

Calling a `get` on the Virtual Subtree returns the value in the *VSL* console[37]:

```
system@agent_mop: /agent_mop/demoService % get greet/Marc-Oliver
Hi Marc-Oliver!
```

Setting the `tickle` node, the service begins laughing in the *Java* console as specified in the notification callback (lines 22-35 in listing 8.2):

```
HaHaHaHa...
HaHaHaHa...
HaHaHaHa...
HaHaHaHa...
HaHaHaHa...
HaHaHaHa...
```

[37]The *VSL* console is like a shell. It is implemented as VSL service and returns the answers to the VSL to the user instead of processing it.

```
HaHaHaHa...
HaHaHaHa...
HaHaHaHa...
HaHaHaHa...
```

While it is laughing, the context node `isLaughing` returns `1` (TRUE). The rest of the time it returns `0` (FALSE) in the VSL shell:

```
system@agent_mop: /agent_mop/demoService % set tickle 1
Setting data...
system@agent_mop: /agent_mop/demoService % get isLaughing/value
1
system@agent_mop: /agent_mop/demoService % get isLaughing/value
0
```

With the described functionality, the *greet and tickle* service gives an introduction into VSL programming. It introduces the concepts *subscriptions*, Virtual Context, and regular context node access. In addition it shows how to register at a KA.

Listing 8.1 shows the wiring part of the service. Lines 2-11 register the service at the KA. Lines 13-22 register a *notification callback* on the context node `tickle`. Lines 24-30 are a helper function that extracts the suffix of a Virtual Node address in the Virtual Node callback (Sec. 5.4.2). Lines 32-48 register the Virtual Node callback on the context node `greet`. The rest of the code starts the service.

```
1  public class ServiceTemplate {
2      public ServiceTemplate() {
3          try {
4              c = new Connector();
5              c.registerService(myServiceIdentifier,
                   myServiceCertificate);
6              myKnowledgeRoot = c.getKORSubtree();
7              h = new ServiceTemplateHandler(c, myKnowledgeRoot);
8          } catch (Exception e) {
9              e.printStackTrace();
10         }
11     }
12
13     public void registerSubscriptions() throws VslException {
14         c.subscribe(myKnowledgeRoot + "/tickle", new ISubscriber() {
15             @Override
16             public void notificationCallback(String address) {
17                if (c.get(myKnowledgeRoot + "/tickle/value").equals
                      ("1")) {
18                     h.tickleHandler();
19                }
20             }
21         });
22     }
23
24     private String getSuffix(String addressPrefix, String
           fullAddress) {
25         String suffix = "";
26         if (addressPrefix.length() < fullAddress.length()) {
27             suffix = fullAddress.substring(addressPrefix.length());
28         }
29         return suffix;
30     }
31
32     public void registerVirtualNodeHandlers() throws VslException {
```

```
33          c.registerVirtualNode(myKnowledgeRoot + "/greet", new
                IVirtualNodeHandler() {
34              @Override
35              public void set(String address, String value, String
                    writerID) {
36                  String suffix = getSuffix(myKnowledgeRoot + "/greet
                        /", address);
37                  if (!suffix.isEmpty()) {
38                      h.greetConsole(suffix);
39                  }
40                  h.greetConsole(value);
41              }
42
43              @Override
44              public String get(String address, String readerID) {
45                  return h.greetBack(getSuffix(myKnowledgeRoot + "/
                        greet/", address));
46              }
47          });
48      }
49
50      public void shutdown() {
51          LOGGER.info("Shutting down...");
52          c.shutdown();
53      }
54
55      public static void main(String[] args) {
56          ServiceTemplate s = new ServiceTemplate();
57          try {
58              s.registerSubscriptions();
59              s.registerVirtualNodeHandlers();
60          } catch (VslException e1) {
61              LOGGER.error("We caught an error: {}", e1.
                    getLocalizedMessage());
62              e1.printStackTrace();
63          }
64          // We will just wait here until the user sends us a "q" [
                return]
65          [...]
66          s.shutdown();
67      }
68  }
```

Listing 8.1: The "wiring" of the greet and tickle service.

Listing 8.2 shows the application logic of the *greet and tickle* service.

Lines 2-6 collect a link to the connector and the root node of the service in the context repository of the KA (Sec. 5.3.1). Both are passed from the `wiring` for simplifying the access in the logic class. Line 5 initializes the `isLaughing` context node.

The Virtual Node handlers in lines 8-14 directly implement the intended logic that does not require further VSL interaction as the parameters are extracted in the `wiring` class (lines 32-48 in listing 8.1).

Lines 16-19 implement the logic behind the *notification callback*. They start the thread that is implemented in lines 22-35. The *tickle* thread sets the context node `isLaughing` to `1`. Then it does the laughing as shown above (listing 8.2). When it is done, it sets the context node `isLaughing` to `0`.

```
1   public class ServiceTemplateHandler implements Runnable {
    public ServiceTemplateHandler(Connector c, String
        myKnowledgeRoot) throws VslException {
        this.c = c;
        this.myKnowledgeRoot = myKnowledgeRoot;
        c.set(myKnowledgeRoot + "/isLaughing", "0");
    }

    public String greetBack(String name) {
        return "Hi " + name + "!";
    }

    public void greetConsole(String name) {
        System.out.println("Hi " + name + "!");
    }

   public void tickleHandler() throws InterruptedException,
       AgentCommunicationException,
            AgentErrorException {
        new Thread(this).start();
    }

    @Override
    public void run() {
        try {
            c.set(myKnowledgeRoot + "/isLaughing", "1");
            c.set(myKnowledgeRoot + "/tickle", "0");
            for (int i = 10; i > 0; i--) {
                System.out.println("HaHaHaHa...");
                Thread.sleep(1000);
            }
            c.set(myKnowledgeRoot + "/isLaughing", "0");
        } catch (Exception e) {
            LOGGER.error(e.getLocalizedMessage());
            e.printStackTrace();
        }
    }
}
```

Listing 8.2: The logic of the greet and tickle service.

The *greet and tickle* service gives an impression how the VSL programming abstraction is practically used. Via the transparency that the VSL provides (Sec. 5.6.1), implementing functionality by mashing up distributed services works the same way. The VSL makes the distribution transparent.

The only functionality not shown yet is the search. It can be invoked as described in Sec. 5.7.2. Running the command in the VSL shell returns:

```
system@agent_mop: /agent_mop % get /search/type/?/services/template/
    demoService
Result for type /services/template/demoService in address /
1    /agent_mop/demoService
```

The Java connector function (Sec. 6.3.3) returns a Java list. The returned addresses are the unique addresses of the context nodes of the type `/services/template/demoService`. They can directly be used in the `get` and `set` methods to retrieve and change context nodes.

Access rights are directly set in the context model and the service certificate (Sec. 5.5). Therefore this section presented all necessary knowledge about the VSL programming abstraction from the point of view of a developer.

The two code listings show how the separation between *wiring* and *application logic* provides additional structure and facilitates the software development.

8.3 Service Classes

To illustrate the practical use of the VSL programming abstraction, example services of the following functionality classes are introduced:

- *Adaptation services* connect Smart Devices to the VSL.
- *Advanced Reasoning Services (ARSs)* infer new context from existing VSL context.
- *Emulator services* emulate the behavior of a service, e.g. an *adaptation service* that interfaces a Smart Device.
- *Remote Access Services (RASs)* provide access to functionality in a remote DS2OS site.
- *User interface services* provide a User Interface (UI) for DS2OS services.
- *Primary context providers* extend the semantics that can be used to identify context nodes.

The VSL programming abstraction does not differentiate between service classes. All services are equal. When using the VSL, a differentiation between functionality happens only based on a service's context model (=abstract service interface) (Sec. 5.2.9, e.g. `lamp23`).

The VSL programming abstraction fosters modularization (Sec. 5.4.4). The identified classes can be used as guideline for programmers to separate functionality into different services to foster reuse (Sec. 8.4, Sec. 7.4.3).

An example for modularization is given next with the *adaptation services.* One service contains the basic reflection of low-level state between Smart Devices and the VSL, and the other part contains the logic that is needed to create higher-level context.

8.3.1 Adaptation Services

> Control of an undertaking consists of seeing that everything is being carried out in accordance with the plan which has been adapted, the orders which have been given, and the principles which have been laid down.
>
> Henri Fayol, French mining engineer, 1841-1925

Connecting Smart Devices to the VSL is a fundamental requirement to implement Smart Space Orchestration with DS2OS (Sec. 2.4.2). Sec. 3.3.3 discussed that the Smart Devices heterogeneity is best bridged *in-the-middle* between the orchestration service and the Smart Device.

As problem transparent loss of information was identified. Sec. 5.2.15 and Sec. 5.6.6 showed how such a loss can be prevented by offering different levels of abstraction of context through the VSL. Sec. 8.3.2 shows an example.

The encapsulation of services that is provided by the VSL (Sec. 5.4.4) requires so-called *adaptation services* that connect Smart Device with the VSL to be autonomous. Sec. 3.7 introduced a structure for the autonomic management of entities. The Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) structure is used to implement a generic adaptation service. The presented generic adaptation service can be used as foundation for creating adaptation services for Smart Devices.

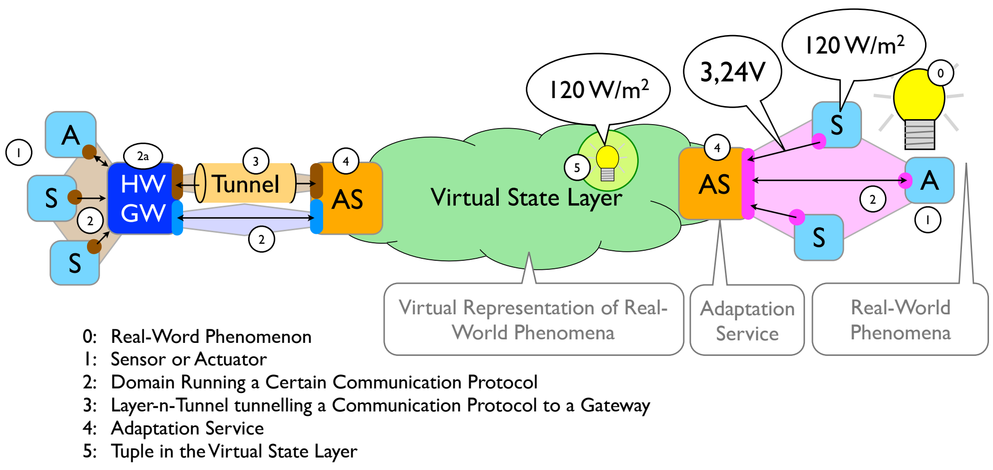

120 W/m²
3,24V
120 W/m²
0
A
S
S
1
2a
HW GW
3
Tunnel
4
AS
2
2
Virtual State Layer
5
4
AS
S
A
2
1
S
Virtual Representation of Real-World Phenomena
Adaptation Service
Real-World Phenomena
0: Real-Word Phenomenon
1: Sensor or Actuator
2: Domain Running a Certain Communication Protocol
3: Layer-n-Tunnel tunnelling a Communication Protocol to a Gateway
4: Adaptation Service
5: Tuple in the Virtual State Layer

Figure 8.2: Embedding of adaptation services in the DS2OS architecture.

Fig. 8.2 shows a DS2OS Smart Space topology with the VSL in the center. The domain marked by (1) identifies the Smart Devices in form of sensors and actuators. Domain (2) is a device-specific protocol domain (Sec. 3.2). As described in Sec. 3.3.5, Smart Devices often offer gateways towards the Internet (e.g. for remote access via smartphones). Such a gateway is shown as (2a) on the left.

The protocol that is offered by such a vendor-gateway can be diverse again (2). Sometimes it can be necessary to tunnel it (3) to the computing node running the corresponding *adaptation service*. An example for such a tunnel is running a device-specific communication protocol over a standard technology such as Internet Protocol (IP).

The communication with the device ends at the *adaptation service* (4). It implements the conversion between the proprietary information exchange with a Smart Device and context model that represents the information of the device in the VSL. The adaptation is *bidirectional*. Changes on the device are reflected into the VSL and changes in the VSL are reflected on the device via the *adaptation service*.

To get informed about changes in the VSL, an *adaptation service* can for instance register for notifications on the root node of its context model, or offer Virtual Nodes (Sec. 8.2).

Direct interaction between orchestration services and Smart Devices does not happen. The communication always goes through the *adaptation service*. This allows the *adaptation service* to arbiter the device access (Sec. 3.3). The arbitration is not limited to pure serialization of parallel device access from different services. By using regular context nodes such access is automatically handled by the VSL. It can also include applying logic such as "fulfilling the change request gradually saves lots of energy", or "from the previous changes it can be expected that this change request is undone very soon".

The upper right of Fig. 8.2 shows an example operation. The real world phenomenon "irradiance" is measured by a sensor as 3.24V via a photo sensor. This information is encoded in a communication protocol and sent-to or pulled-by an *adaptation service*. The *adaptation service* contains logic to reverse the transformation from the real world phenomenon into a measured sample. Ideally it can reconstruct the phenomenon which is an irradiance of $120\frac{W}{m^2}$ in the example.

As described in the Sections 5.2.15, 5.6.6, and 8.3.2, it can also store the 3.24V and another service can apply the transformation from 3.24V to $120\frac{W}{m^2}$. For minimizing the information loss (<R.9>) it should store the 3.24V as additional context in the VSL in any case (Sec. 5.2.15). See (6) in Fig. 8.3.

A Generic Adaptation Service

Fig. 8.3 shows details of a possible architecture of an *adaptation service*. The center of the architecture (4) is the MAPE-K loop (Sec. 3.7).

The *plan* module (P) decides how changes in the VSL should be reflected to the Smart Device. It contains the decision logic. The *execute* module (E) sends the commands that reflect the decision of the *planner* to the Smart Device.

The *monitor* module (M) receives or collects information from the Smart Device. The *analyze* module (A) transforms the information from the Smart Device into VSL context.

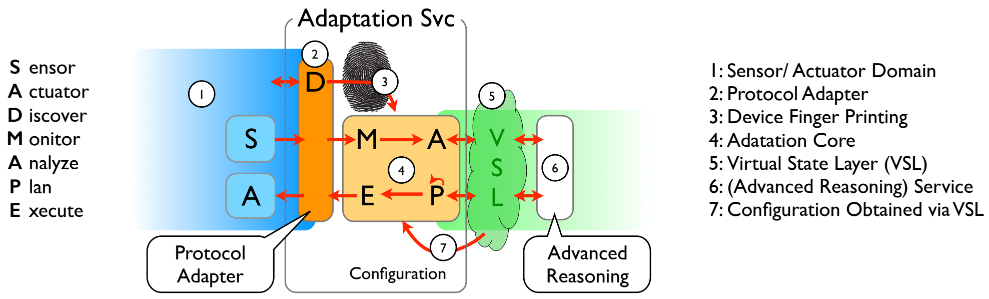


Figure 8.3: Architecture of a Generic Smart Gateway.

To establish a feedback loop between a service that wants a change, and the autonomous *adaptation service*, additional context nodes are added to the context models of a service that provides autonomous adaptation. This use of context models is independent of the functionality provided by the VSL programming abstraction. The additional nodes are handled like any other VSL node. Their semantic comes with the use in services.

```
<timer1 type="/ilab/triggerTimer,/ilab/timer,/basic/number" version
    ="0"
        timeStamp="2014-02-25 16:50:42.884" access="r">
  <desired type="/ilab/timer,/basic/number" version="0"
           timeStamp="2014-02-25 16:50:42.941" access="rw" />
</timer1>
```

Listing 8.3: A context node with a `desired` subnode.

Listing 8.3 shows the instance of a context nodes that has a `desired` subnode. The field `access` shows that `/timer1` is read-only. It contains the current state of the timer. The `/timer1/desired` node can be written. It contains the state, the timer should have.

If the `/timer1/desired` node is different from its parent `/timer1`, a change request is issued and the *planner* (P) gets active. After a change of the state of the Smart Device that is represented by the context node, the *analyzer* (A) changes the corresponding node. This is the node `/timer1` in the example. This loop tells the service that changed the `desired` node when a change request is reflected. Fig. 9.3 in Sec. 9.3.2 shows the node relationship graphically.

The simplest form of an *adaptation service* is reflecting VSL changes to the Smart Device and vice versa without additional reasoning. Such behavior is sufficient if another service adds the reasoning logic (Sec. 5.6.6). Interfacing the particularities of a Smart Device protocol is fundamental for interfacing formerly unsupported device. If an *adaptation service* with such basic functionality is available over the S2Store, other developers can develop the reasoning logic and add it via another service (6) (Sec. 7.4).

The availability of the described basic adaptation is relevant to provide support for diverse heterogeneous Smart Devices (Sec. 2.4.2). The adaptation has to be done for each Smart Device that should become usable for Smart Space Orchestration via the VSL. If a suitable *adaptation service* is available in the S2Store, it can directly be used. The selection based on a fingerprint is described below.

The decoupling from the abstract logic facilitates the creation of adaptation services as described next. In addition, the modularization allows to reuse the additional logic for multiple *adaptation services*, e.g. a *lamp23* that uses *protocol23* and a *lamp42* that uses *protocol42*.

The functionality of the generic *adaptation service* is transforming VSL context changes into commands for a Smart Device and to reflect the state of a Smart Device to the VSL.

Fig. 8.3 shows the decoupling of the functionality into a generic part (4) that can be parameterized (7), and a device-specific part (2). The device-specific part is called *protocol adapter* (2). Its task is transforming device-specific information into an intermediate format that is used by the Monitor-Analyze-Plan-Execute (MAPE) in (4).

The MAPE takes the intermediate information and transforms it into VSL context via a simple mapping. In a prototype implementation the parametrization of the MAPE is a lookup table with regular expressions. Writing a protocol adapter and the entries of the lookup table is sufficient to enable the generic adaptation service to interface a new Smart Device. This is expected to facilitate the creation of *adaptation services* and thereby the VSL support of diverse Smart Devices (<O.3>, <O.4>).

Automated Smart Device Integration

The fingerprint (3) illustrates another feature that was prototyped: the automatic discovery of new devices. For devices that use a known protocol, the generic adaptation service can be extended to integrate new devices autonomously.

When it discovers a new device, it tries to fingerprint it. Via the fingerprint a suitable configuration for the MAPE of the generic adaptation service can be loaded from the S2Store, it can be configured, and the new device is usable via the VSL.

Other *adaptation services* that do not follow the described generic scheme can also be automatically installed from the S2Store. The fingerprinting service can ask the Site Local Service Manager (SLSM) to install a certain adaptation service for the newly discovered device from the store. If the user enables the feature, this can happen fully automatically. Smart Devices that can be integrated with the described mechanism do not require manual setup by their buyers besides connecting the device to the network, the *new device discovery service* runs in.

Further investigation of this feature is future work.

Automated Context Model Generation

For protocols that have a similar context representation to the VSL this step can be taken further. Such a protocol is the Simple Network Management Protocol (SNMP) protocol [Har11]. In case of SNMP, a prototype was implemented that discovers new SNMP devices, crawls their context model, and creates a VSL context model automatically. It uploads the resulting context model to the VSL for additional modifications by other developers. The described method generates *adaptation services* for formerly unsupported devices automatically. This is possible as the methods of the SNMP protocol are fixed.

Such automated integration is promising as it allows the automated integration of formerly unknown devices and their formerly unknown context models into the VSL. The support of divers Smart Devices is an important requirement for the real world penetration of pervasive computing (<O.4>).

The investigation of techniques for automated integration of new devices into the VSL by automatically generating suitable *adaptation services* is future work.

The presented mechanisms support user-based and crowdsourced development of *adaptation services* (<R.4>, <R.48>).

<R.4>

<R.48>

8.3.2 Advanced Reasoning Services

One can't proceed from the informal to the formal by formal means.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

Looking at the human mind in Sec. 3.5 identified the availability of context on *different levels of abstraction as supportive* for humans to plan tasks (<R.25> <R.28>). VSL subscriptions (Sec. 5.3.2) can be used to couple context representations on different levels of abstraction with each other. See Fig. 5.6 in Sec. 5.6.6.

A benefit of such coupling is that services become inter-operable on different levels of abstraction. An example is shown in Fig. 8.4. The service on the left can only interface the context *sunshine*. It could not interact with the shown sensor without the coupling of a so-called Advanced Reasoning Service (ARS).

Offering context on different levels of abstraction facilitates the development of services. In the example, the functionality "*is the sun shining?*" is transparently provided. A developer does not have to care about it and can simply use the higher level context. This simplifies the implementation of the service on the top left. Via DS2OS it is sufficient that a developer specifies a dependency to the *sunshine* context model in the service manifest. The SLSM automatically proposes suitable services to the user on installation of the service if it has locally unresolved dependencies (Sec. 7.4.3).

Advanced Reasoning Services (ARSs) acquire context from the VSL, reason on it, and publish new context in the VSL. An example is shown in Fig. 8.4. An *adaptation service* gets the measurement of 3.24V and –as it knows the properties of the photo sensor– inverts the transformation from an irradiance into a current that was done on the sensor[38].

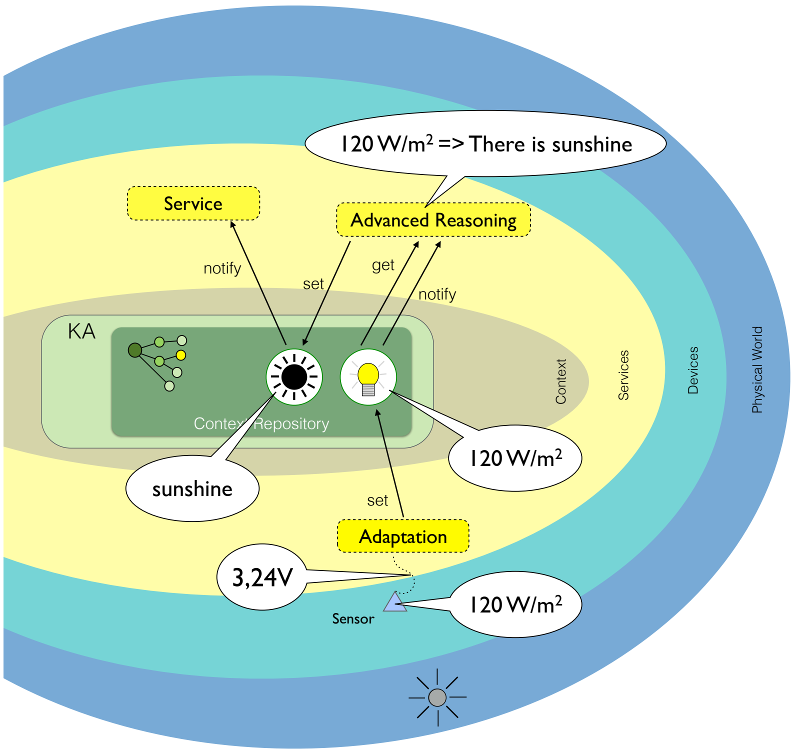


Figure 8.4: Automatic correlation of different levels of context abstraction via advanced reasoning services using the VSL programming abstraction.

Using the context from the *adaptation service* and possibly other context such as location or other sensor readings (not shown in the figure), the ARS on the top right infers that it is sunny.

To couple updates of all levels of abstraction in the context representation (context nodes), the subscription feature of the VSL (Sec. 5.3.2) is used. The ARS subscribes the VSL context node that stores the value from the *adaptation service*. When it changes, the VSL notifies the ARS of the change. The ARS reads the new context

[38]This process can be modularized so that the *adaptation service* only stores 3.24V and an ARS that knows the properties of the sensor transforms the raw data into $120W/m^2$ as described in Sec. 8.3.1.

value, reasons about it, and updates the higher level context accordingly. This implements a coupling between different levels of abstraction in the context repository as shown in Fig. 5.6 in Sec. 5.6.6.

The described coupling also works top-down. A change on a higher level of context abstraction leads to a change of lower level context. If the context node that is subscribed by the *adaptation service* (lowest level of abstraction) changes, the service automatically tries to reflect the change to the actuator (Sec. 8.3.1). That the change was initiated on a higher level of abstraction is transparent to the *adaptation service.*

Via the implemented VSL coupling of services over context nodes (Sec. 5.4), this coupling can be used in a fully transparent way. At the same time, the coupling can also be looked up by inspecting the subscriber identifier lists of all participating nodes. This may be of interest for services to know if a context node contains primary information or if it is derived (Sec. 5.2.15).

For implementing a *synchronous coupling*, Virtual Context (Sec. 5.4.2) can be used to chain different *advanced reasoning services.* In this case, a request on a Virtual Node on a high level of abstraction automatically leads to subsequent synchronous requests on the subsequent *advanced reasoning services* and finally on the *adaptation service.*

Reasoning Process

Reasoning can be done via different methods [KKR+13, BBH+10] including probabilistic models such as Hidden Markov Models, or Conditional Random Fields [KHC10], Finite State Machines (FSMs) (Sec. 3.6), rule-based reasoning e.g. with policies (Sec. 3.6.4), or description logic (Sec. 3.9).

Fig. 8.5 shows *different levels of abstraction.* They can be coupled via the VSL as described. The picture in Fig. 8.4 and the formalization in Fig. 8.5 show similar things. In the picture an *adaptation service* creates a *primitive context.* It gets further reasoned via one or multiple ARSs. The ARSs provide different levels of abstraction.

In the reference model in Fig. 8.5 ARSs are situated in the labeled box at the top. They provide context on the upper abstraction levels *mid-level context* to *high-level context.*

As described in Sec. 8.3.1, the VSL design allows to implement the steps below the box too. Depending on what a Smart Device communicates, all steps from the *raw sensor data* to the *low-level context* can be implemented with ARSs.

The coupling of services implies that all steps of abstraction have a representation in the VSL. Starting the VSL based processing at the raw data solves the problem of information loss (<R.9>) as more precise data than the raw data is not available for electronic processing [Sha48, Nyq24] (Sec. 5.2.15). *<R.9>*

8.3.3 Emulator Services

Sec. 5.6.7 described how the use of context and Virtual Context for information exchange between services in VSL Smart Spaces fosters the development and use of emulators (<R.47>). The section also described that providing emulators can foster Smart Device sales as it enables developers to create orchestration services without *<R.47>*

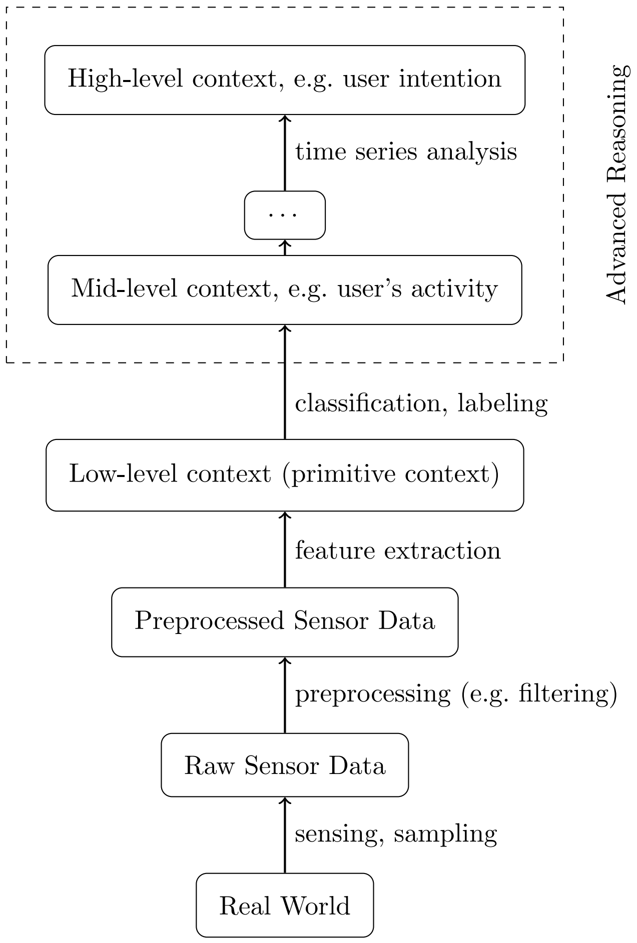


Figure 8.5: Context processing stages with different levels of context and different methods for inference. Adapted from [KKR+13].

possessing a hardware. The resulting services support provides an incentive for other users to buy a hardware. Sec. 5.5.6 emphasized the relevance of emulators for testing. Both benefits of emulators are detailed in this section.

Using context as interface for services has the advantage that service states are limited by the exposed context nodes and their data types. This helps validating the behavior of a service as described in Sec. 5.5.6, and it facilitates the creation of emulators for services.

Everything in DS2OS is a service. Smart Devices are connected via adaptation services (Sec. 8.3.1) to their virtual representation as instance of their context model in the VSL. This makes it possible to emulate a Smart Device by interacting with the context model instance accordingly.

As an emulator service uses the original VSL context model the emulation is fully transparent to other services.

The possibility to emulate Smart Devices is important for developers as it allows to develop and test services without using the real hardware. This makes it possible to emulate the orchestration of Smart Devices without having to buy them [EBDN02]. Emulators foster crowdsourced development as they allow development without hardware investment (<R.48>). *<R.48>*

Software Development Kits (SDKs) for smartphones typically allow to run Apps in an emulator in 2014 [App13, Goo]. Using the emulator for testing reduces the development time as testing is faster than loading an App on a device first [GR11]. Smart Spaces are more complex than smartphones with their distributed and diverse Smart Devices (Sec. 3.10). Trying things out and experiencing the effects of actions are important methods that help humans to learn [PPB$^+$04, Pap05].

Large installations and the interplay of different Smart Devices can be tested with emulators. As each future Smart Space can be expected to contain different sensing and actuation possibilities, this is relevant since developers can impossibly have all possible configurations at hand. Such testing fosters the dependability (<R.30>) and the security (<R.49>) of DS2OS services. *<R.30>* *<R.49>*

Emulators are required as developer support (<R.1>). *<R.1>*

Typical emulated Smart Space environments could be provided as test environments to developers. Such environments could also be used for automated tests of the functionality of new services before publishing them over the S2Store (Sec. 7.4, Sec. 7.4).

Fig. 8.6 shows an emulator on the monitor in the back. It was developed for a demonstration of the VSL. A window, a heater, and a personal computer are emulated. The physical devices in the setup are a light (front left), a light (back on top of the screen), a fan (back on top of the screen), and a switch that is on the table and hidden by the people but reflected in the screen below the hand.

The emulation is transparent to DS2OS services. The touch screen in the center runs the graphical user interface service that is introduced in Sec. 8.3.5. It allows to interact with all physical and emulated Smart Devices the same way (transparency).

For fostering the support of new device classes, it could be interesting for hardware manufacturers to provide emulators to their devices in an early stage of the product development already to have services that can use the new product on the market already when the product is released. See Sec. 10.4.

Figure 8.6: A mixed demonstration setup of a DS2OS Smart Space consisting of emulated and real Smart Devices as shown at Technische Universität München (TUM) Tag der offenen Tür 2011. The photo shows Deniz Ugurlu in the back, who contibuted in programming the KAs prototype, and Ursula Eschbach, the public relations manager of the TUM computer science department, in the front.

8.3.4 Remote Space Access

For sharing sensor data, or other information such as learned optimization strategies [PNKC13], it might be interesting to connect distributed DS2OS Smart Spaces. This can be done via a *Remote Access Service (RAS)*.

The design is similar to the one described for service proxies in Sec. 7.2.4. A *RAS* acts as proxy for functionality that is available in another Smart Space. It makes remote functionality locally available in a transparent way.

Functionality is represented by context models in the VSL (Sec. 5.4.1). To provide context from a remote DS2OS site, a *RAS* is started in the local DS2OS site and another instance is started in the remote DS2OS site. Both are connected independent of the VSL. This can happen via any connection such as a Virtual Private Network (VPN) tunnel running an Hyper Text Transfer Protocol (HTTP) connection.

Both *RASs* run as regular DS2OS services which keeps all security properties of DS2OS intact, including the access rights setting mechanism. The user can dynamically change the access rights of the *RASs* as described in Sec. 7.3. This gives local control over the functionality that can be accessed from the remote site.

Both *RASs* register the context models of the *remote* context they want to make locally available in the VSL. To do so, their own context models contain a *list* that allows nodes of any type to be added (Sec. 5.2.10). When a new context model is registered it is added as new subtree in the list.

Providing the context models allows interfacing the remote functionality locally. For providing the functionality, both *RASs* register all context nodes as Virtual Nodes and forward requests to the remote site via their connection. According to the (dynamically changeable) access rights, the remote RAS has in the remote DS2OS site, a request is carried out. The result is sent back and returned transparently to the service that accessed the context of the RAS in its DS2OS site. This results in a transparent provisioning of remote context within a DS2OS site.

By using the original context models, a *RAS* is a transparent context provider and context consumer[39]. Fig. 7.3 shows a connection between two *RASs* with the gray dotted lines.

8.3.5 Web-Based Graphical User Interface

Smart Spaces need User Interfaces (UIs) to enable their users to provide (high-level) goals to their software orchestrated spaces. The diverse silo-systems that are present in spaces today (Sec. 3.2) have typically each a different User Interface (UI). Consistent UIs are preferable to heterogeneous interfaces as they provide better usability [MN90, Opp02].

The VSL programming abstraction enables the creation of a *single* UI for *all* services within a DS2OS site. Enabler are the *multi-inheritance* of the VSL meta model (Sec. 5.2.11) and the *rights delegation* that is possible via the VSL API (Sec. 5.3.1)

The use of descriptive context as interface to all functionality of a DS2OS Smart Space allows the representation as state in the interface. The inheritance of all data types from three basic data types allows to provide a generic interface that can present all context that is available in the VSL.

Context Representation

The context validation in Sec. 5.2.14 is based on the fact that the restrictions of all context nodes with values are defined by the basic data types. The UI uses the same property by providing interaction elements for the three basic data types `text`, `number`, and `list`. This allows to represent *all* available VSL context.

The upper context subnodes of the left open context user interface in Fig. 8.7 show basic renderings that are automatically available as the nodes are of type `/basic/number`. The bottom nodes and the nodes in the right context menu have custom renderings.

Sec. 5.2.14 proposed the introduction of custom validators that get selected by traversing the inheritance chain from the type with most semantics (left) to the type with least semantics (right). The UI uses the same mechanism for selecting custom renderers. The interface looks for the first type from the left it finds a renderer for.

The availability of a renderer on a more semantically rich level (left) allows to render the same value with more semantics as shown with the buttons in Fig. 8.7 that render context nodes of type *isOn*.

The custom renderers can be provided via context models over the S2Store. In the prototype they are based on Hyper Text Markup Language (HTML) and regular expressions. The intention of this design is that developer additionally provide renderer for their services in addition to the other data when submitting services to the S2Store (Sec. 7.5).

[39]All VSL mechanisms including security and discovery remain regularly working by this approach.

The ability to use the standard UI for their services frees developers from most of the work that is involved when developing a UI. The unification facilitates the use for end-users, and it facilitates the development of DS2OS services (<R.1>). *<R.1>*

As renderer can be added to the *interface service* at run time, this implements a modular (<R.26>), and dynamically extensible (<R.23>) UI. *<R.26>*

<R.23>

Context Access

The API of the VSL allows to delegate rights to services (Sec. 5.3.1). The UI service has only basic access rights in its service certificate.

When a user logs-on the rights get extended via the user certificate[40]. The log-on process and the temporal rights delegation implement multi-user support (<R.2>). *<R.2>* As VSL functionality they implement security-by-design (<R.49>) for the interface service. *<R.49>*

The interface is stateless (<R.37>) as it loads all data on request with the current access rights. Always the newest data is shown according to the current rights that can be narrow (e.g. not logged on) or broad (e.g. logged on as administrator). *<R.37>*

Interface Design

To make the UI for Smart Spaces intuitive, a spatial interface is implemented in this service. The implementation is based on Open Layers[41], an open source JavaScript library for displaying map data in web browsers. Open Layers supports all major map providers in the web, Google[42], Microsoft[43], and OpenStreeMap[44]. It allows to display map data, satellite images, and custom overlays.

Fig. 8.7 shows the interface. It is usable like other map services in the web. On the right, the menu to select the view and the presented data is opened. Currently the satellite map of the TUM computer science department is shown.

The screenshot shows that the map interface has a custom overlay added. It provides the floor plan of the building. This allows to locate entities in rooms. Map providers do not offer spatial information of that granularity in 2014. The interface allows to load custom images such as scanned floor plans. In a real world deployment users could also draw their room plans if they do not have professional floor plans from their architects like in the case of the shown building.

Via the search provider (Sec. 6.6.4), location information is provided for VSL context nodes. The upper three nodes in the left displayed context UI element show the geolocation of the context node. The uppermost value is the height that is not provided in the example.

The interface can be used to position context nodes that are symbolized by the markers on the map. If a user brings a new device or installs a new functionality that has

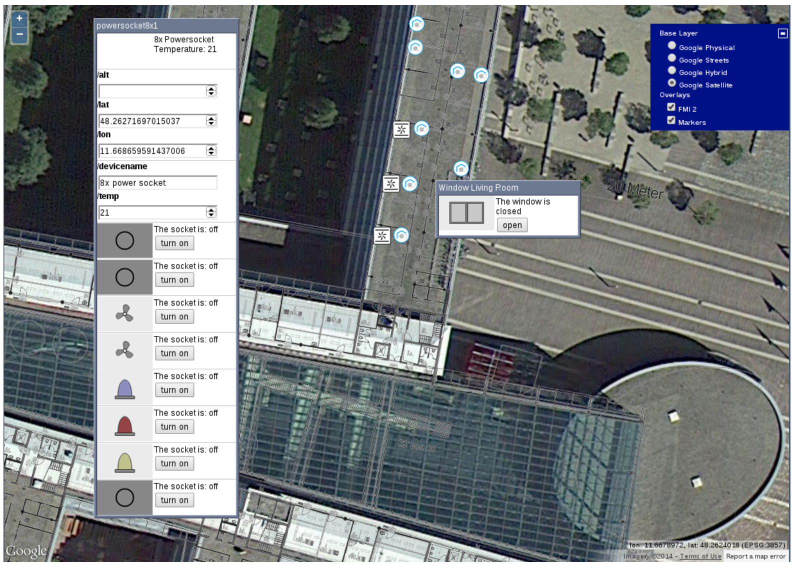
[40]In the prototype the user certificates are stored in a central service and protected by a password. The log-on acquires the user's certificate from that service and adds it to the UI service.

[41]http://openlayers.org/

[42]https://maps.google.com/

[43]http://www.bing.com/maps/

[44]http://www.openstreetmap.org/


powersocket8x1

8x Powersocket
Temperature: 21

/alt

/lat
48.26271697015037

/lon
11.668659591437006

/devicename
8x power socket

/temp
21

The socket is: off
turn on

The socket is: off
turn on

The socket is: off
turn on

The socket is: off
turn on

The socket is: off
turn on

The socket is: off
turn on

The socket is: off
turn on

The socket is: off
turn on

Window Living Room

The window is closed
open

Base Layer
Google Physical
Google Streets
Google Hybrid
[✓ Google Satellite]
Overlays
[✓ FMI 2]
[✓ Markers]

lon: 11.6678972, lat: 48.2624018 (EPSG:3857)

Figure 8.7: Interface screenshot of the of the map-based UI service.

geolocation properties, a new marker appears on the map and can be positioned via drag&drop. The interface submits the coordinates of the position the user chooses to the search provider (Sec. 6.6.4) that stores them so that the position is associated to the node for all users.

System Architecture

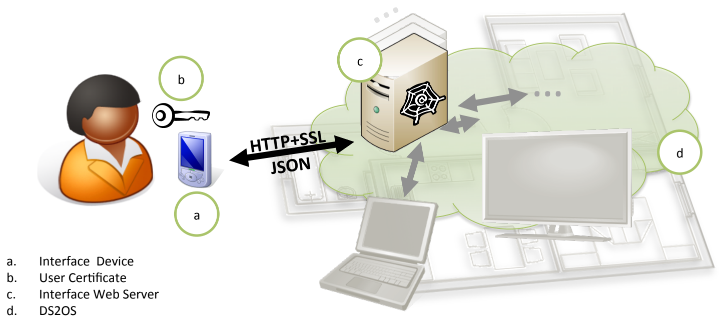


Figure 8.8: Architecture of the web-based UI service.

Fig. 8.8 shows the architecture of the described UI service. It is implemented as regular DS2OS service that provides the UI as website. To secure the log-on and the Smart Space data, the connection is protected via Secure Hyper Text Transfer Protocol (HTTPS). If no user is logged in, only publicly accessible context nodes can be browsed on the map (Fig. 8.7).

Using web technology allows diverse devices such as smartphones, tablets, Personal Computers (PCs), and probably many future devices to access the UI service without having to install additional software to a web browser. Fig. 8.6 shows the interface on the touchscreen.

8.3.6 Primary Context Provider

Primary context providers are a VSL extension for semantic queries (<R.43>). *Primary context providers* can be retrieved over their modelID. Semantic queries over the available context information in the VSL knowledge base are an important tool for implementing advanced reasoning and orchestration functionality [BBH+10, HIM05a, RCKZ12]. *<R.43>*

The μ-middleware property of the VSL allows the dynamic extension with additional *primary context providers* (Sec. 5.7.3). The location search was presented in Sec. 5.7.2 and Sec. 6.6.3.

Primary context providers extend the `/system/searchProvider` model for their context model (listing 5.7). They register their root node as Virtual Node and provide an implementation for the `get` method as shown in listing 6.4 or listing 8.1 lines 44-46.

Dependency Search Provider

As an example for the extensibility, dependency information between entities can be added as primary context to the VSL. The BOSS middleware that was presented as an existing solution in Sec. 4.5.7 provides such dependency information as primary context. In BOSS, dependency information is represented as a graph. Having such a graph, a dependency search provider can be added to the VSL at run time as described above.

The implementation is similar to the implementation of the location search shown in listing 6.4. Queries can be issued as follows:

```
system@agent_mop: / % get /search/dependencies/agent_mop/lamp32
Result for dependencies of node /agent_mop/lamp32
1    /agent_mop/powerSupply42
```

8.4 Complex Functionality via Simple Service Mash Up

The trigger service that is introduced below is an example how the VSL abstraction structures and facilitates the implementation of pervasive computing scenarios (<O.2>) in the real world (<O.4>). The unified transparent context access that is implemented by the VSL allows the creation of diverse scenarios (<O.3>) over the borders of the domain silos that exist in 2014 (Sec. 3.3.1, Sec. 6.2) via simple Event-Condition-Action (ECA) rules.

The presented mechanisms support user-based and crowdsourced development of pervasive computing scenarios (<R.4>, <R.48>). *<R.4>*

<R.48>

8.4.1 Trigger Service

The trigger service implements the evaluation of ECA rules (Sec. 3.6.4). Its context model has two VSL *lists* (Sec. 5.2.10). The first *list* contains the conditions, the second *list* contains the actions. Using *lists* as data structure allows the dynamic definition of ECA rules at run time.

To implement the event notification, the trigger service automatically subscribes for change notifications on all nodes in the condition set.

VSL context nodes are always derived from the basic data types *text* or *number*. Their fixed restrictions enable the validation of the conditions and actions of the trigger service on submission. This enhances the dependability of the service (<R.30>). *<R.30>*

The restrictions define the valid range for the right side of a condition argument. For the following context node instance:

```
<mySpace>
  <luminosityOutside7 type="...,/basic/number" minimumValue="0" />
</mySpace>
```

`/mySpace/luminosityOutside7/value > 5000` would be a valid condition as the type of `/mySpace/luminosityOutside7` is inherited from `/basic/number`. `/mySpace/luminosityOutside7/value == hot` would be invalid.

The VSL typing is also applied to check the validity of the actions in advance. The actions are value assignments in the VSL programming abstraction.

The trigger service provides a simple boolean algebra with the operations `not` (!), `and` (&), `or` (|), grouping ("(",")") to define the evaluation order, and simple calculation (+,-,*,/). All described operations can be implemented via graphical direct manipulation which simplifies the use (<R.1>). *<R.1>*

The generic DS2OS Graphical User Interface (GUI) can be used to configure triggers (Fig. 8.7).

For triggering periodic events, the trigger service can be coupled with a timer service. Such a service is described next.

8.4.2 Timer Service

The *timer service* implements timer triggers. It can be used for creating periodic events with the trigger service for instance.

The *timer service* offers Virtual Nodes that trigger a notification when their event happens. Usage the *timer service* is subscribing on a Virtual Node in the service's Virtual Subtree and waiting for the notification (Sec. 5.4.2).

An example node is `mySpace/timer/periodic/every/seconds/42`. On subscription of the node, the timer service creates a timer and sends a notification to its subscribers every time the timer is up. When all subscribers unsubscribed the timer is deleted.

Another example is the node `mySpace/timer/once/datetime/14/02/2014/13/52/10` that sends a notification on February 14, 2014 at 13:52:10.

The example shows how the Virtual Node can be used to pass parameters. Depending on the passed parameter, new timers are created in the service. An example for the asynchronous coupling of the *timer service* and the *trigger service* is the *sunshine service* that is introduced next.

8.4.3 Sunshine Service

The VSL programming abstraction enables the implementation of complex behavior via the mash up of simple services. The *sunshine service* gives an example.

The *sunshine service* is an ARS that periodically infers the knowledge if the sun is shining. It is implemented as parametrization of the *trigger service*. Table 8.1 shows the condition and the actions.

condition set
`/mySpace/timer/periodic/every/minutes/5`

action set
`/myAddressSpace/isSunshine` =
& `/mySpace/irradianceEntrance` > 120
& `/mySpace/irradianceBackyard` > 120
& `/mySpace/irradianceRooftop` > 120

Table 8.1: Sunshine service implemented with the trigger service.

The service is triggered by the *timer service* every five minutes. As action it decides based on the current readings from three preconfigured sensors if the sun is shining and sets the value of the context node `/mySpace/timer/periodic/every/minutes/5` accordingly [45].

A topology view on the example is shown in Fig. 8.2. The sensor on the right next to the light symbol measures the irradiance, the adaptation service on the gateway (4) saves the information in the VSL (5). Fig. 8.3 shows the positioning of the sunshine service as advanced reasoning service (6) on the right.

The result, if the sun is shining, can be used as regular context by other services as described in Sec. 8.3.2. The sunshine service encapsulates the complexity to determine if the sun is shining. It offers the new context that is on a higher level of abstraction than the sensor readings, and the timer.

8.4.4 Event Condition Action

As described in Sec. 3.6.4, ECA rules are a concept that can easily be understood by humans and that suits to express the logic of context-aware services [GZI10]. With the presented VSL programming abstraction, the limitations for the use of ECA rules for implementing complex orchestration workflows are eliminated. See Sec. 3.6.4 for a list of the identified limitations.

- The **lack of context** is solved by the fixed function interface of the VSL that allows to access all context in a uniform way, making all context that a service has access to retrievable in a transparent uniform way (Sec. 5.3.2).

[45]Though the value of 120 W/m^2 that is used by the World Meteorological Organization to determine sunshine (http://www.wmo.int/pages/prog/www/IMOP/publications/IOM-104_TECO-2010/P3_25_Vuerich_Italy.pdf) seems too scientific for real world usage, it could be determined by Smart Spaces users by looking at the current sensor values on a sunny day for instance.

The versioning of the VSL provides **historic context** using the regular addressing scheme (Sec. 5.3.2).
The Virtual Nodes give access to **dynamic context** that can be created on demand (Sec. 5.3.4).

- The restriction of not being able to **access non-local context** is eliminated by the VSLs location transparency that allows to access context in a uniform way independent of its storage location (Sec. 5.6.1).
- The dual use of the context models as knowledge description and as abstract interfaces to services allows to describe and define **interactions between services** by changing states in the VSL (Sec. 5.4.1).
- The **complexity of multiple layers of reactive services that interact with each other** is simplified by the strict service encapsulation. It is enforced by the context model as well-defined interface, and the statelessness of services (Sec. 5.4.4). The VSL makes it transparent to services if the origin of a fact in the VSL is generated by another ECA service or any other service such as an adaptation service. This hides complexity.

As shown on the example of the *sunshine service* and analyzed in this paragraph, the VSL programming abstraction allows to implement complex functionality by using simple mechanisms such as ECA (Sec. 3.6.4). Structuring and facilitating the creation of services that implement pervasive computing scenarios is a major goal of this work (<O.0>).

8.5 Chapter Conclusion

The main purpose of this chapter is giving a practical overview on the usability of the VSL programming abstraction.

The *greet and tickle service* shows the simplicity of programs that results from using the VSL. The six service classes were chosen to give an overview on how the VSL facilitates the development of services from different functional areas. Besides being examples, the presented services provide fundamental functionality for the implementation of pervasive computing in the real world (<O.4>).

The modularization of the *adaptation service* shows how little background in programming is needed to create an adaptation stub that can be used to interface devices that end-users have in their spaces. The automatic discovery of devices, and the automated generation of *adaptation service* are probably novel in the presented way.

The *advanced reasoning service* example shows how the VSL programming abstraction fosters the creation and coupling of context on different levels of abstraction. This facilitates the use in services, and it makes services compatible as the providing of new context nodes is identical to the process of interface adaption since the context models of the VSL are its abstract service interfaces.

The *emulator service* section showed how the VSL programming abstraction fosters the implementation of emulators. They are important for creative exploring, testing, and to foster the business with Smart Device hardware.

The coupling of DS2OS spaces is attractive for sharing functionality such as specialized devices (e.g. wind sensor), or knowledge (e.g. learned energy consumption optimization schemes). The *remote access service* shows how such functionality can seamlessly be integrated into the VSL. This is enabled by the interface model of the VSL programming abstraction.

The web-based *user interface* makes use of the VSL type hierarchy. It provides a starting point for the relevant problem how user interfaces for future Smart Spaces can be designed. Offering such a unified interface for all services is probably a novel approach.

The *primary context provider service* shows the extensibility of the μ-middleware design in all functional areas including the core functionality of locating context based on diverse semantic identifiers. Such functionality cannot be found in the state of the art (Sec. 4.5).

The mash up finally shows the power of the descriptive abstraction that is provided by the VSL. It allows to implement complex functionality without programming by the simple coupling via ECA rules that can be parametrized.

The introduced services illustrate and provide fundamental functionality for future Smart Spaces. They illustrate the simplification on the process of implementing pervasive computing scenarios that is introduced with the VSL programming abstraction.

Part III

Evaluation

9. Evaluation

There are two ways to write error-free programs; only the third one works.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

Short Summary The chapter evaluates the Virtual State Layer (VSL) quantitatively and qualitatively.
The *latency* of service coupling over the VSL is measured using the two coupling mechanisms, *value change notification* and *Virtual Context*.
The *scalability* of the VSL is evaluated in a large testbed, and a smaller one that can be expected to be closer to future real-world deployments. The self-management overhead of the VSL is evaluated.
A user study is presented that shows the real world usability of the VSL.

Key Results The VSL programming abstraction allows the implementation of service functionality that is perceived interactive by users. This is the case for simultaneous access by multiple services at the same time. The VSL implementation scales. The Knowledge Agent (KA) synchronization mechanism scales and is dependable. The VSL programming abstraction is simple-to-use in real world. The VSL enables the implementation of complex pervasive computing scenarios in a structured way within a short time.

Challenges Addressed <R.1> *Simplicity-to-use*, <R.3> *Self-management*, <R.4> *User participation*, <R.5> *Context-awareness support*, <R.40> *Meta model with suitable properties*, <R.29> *Simple-to-use context abstraction*, <R.30> *Dependability*, <R.49> *Security-by-design*, <R.50> *Scalability*.

Summary The chapter evaluates the VSL in different dimensions.
First the hardware setup is introduced (Sec. 9.2). Then the reproducibility of the measurement results is discussed (Sec. 9.2.2).
The *latency* of service coupling over the VSL is measured using the two coupling mechanisms, *value change notification* and *Virtual Context*. A first measurement compares the latency of the two mechanisms for manipulating the state of a physical Smart Device (Sec. 9.3.3). For obtaining absolute latencies, a second measurement with an emulator (Sec. 8.3.3) is done (Sec. 9.3.4). Next the effect of different notification delays is evaluated (Sec. 9.3.5).
The *scalability* of the VSL is shown in a larger testbed. The first setup measures latency and throughput of up to 1600 simultaneous requests to one service on one KA (Sec. 9.4.3). A second measurement gives more details over the range between one and 226 simultaneous requests to one service on one KA (Sec. 9.4.4).
A last measurement evaluates self-management overhead of the VSL by inspecting the synchronization of KAs (Sec. 9.5).
A user study is presented to show the real world usability of the VSL (Sec. 9.6). It is quantitatively (Sec. 9.6.5) and qualitatively evaluated (Sec. 9.6.6). As part of the regular curriculum at Technische Universität München, the study material teaches computer science students about Smart Space Orchestration.

9.1 Introduction

This section evaluates the Virtual State Layer (VSL) in different dimensions.

First the *delay* of service interaction over the VSL is measured (Sec. 9.3). It is relevant as pervasive computing scenarios can have timing requirements. The two coupling mechanisms of the VSL, *subscriptions on node value changes*, and *Virtual Context* are compared. The measurement includes a real world scenario with communication to a Smart Device, and the interaction with an emulator (Sec. 8.3.3).

Then the *scalability* of the VSL is measured (Sec. 9.4). A first measurement evaluates if the VSL implementation can handle a high amount of simultaneous requests (1600 requests) against a single service on a single Knowledge Agent (KA). A second scenario measures the range between one and 226 simultaneous requests with higher resolution. As the VSL is designed as Peer-to-Peer (P2P) system, this access pattern is more likely to happen within future Smart Spaces as the load is expected to be distributed between different independent KAs with distributed services (Sec. 6.3).

Another measurement evaluates the synchronization behavior of the VSL (Sec. 9.5). This is important to see which *management overhead* in traffic and processing the self-management of the KAs creates. It shows the *convergence time* of a VSL instance via self-organization.

To assess the *real world usability* of the VSL programming abstraction, a user study is conducted (Sec. 9.6). It aims to confirm the simplicity-to-use, and the real world usability of the VSL programming abstraction as basis for the proposed distributed user-based, crowdsourced development (Sec. 7.4).

All tests are implemented as regular Distributed Smart Space Orchestration System (DS2OS) services. The VSL KAs are not modified.

9.2 Hardware Setup

For comparability of the results, all measurements presented in this chapter were done in the same testbed. The testbed consists of six so-called isles with six computers per isle. Each computer is equipped with an Intel Quad Core-i5 2.5Ghz processor, 4GB of main memory, 128GB solid state disk storage and 1Gbit/s Ethernet interfaces.

To enable user interaction for conducting the user study (Sec. 9.6), each isle is equipped with monitors and keyboards as shown in Fig. 9.1. Each of the shown monitors and keyboards can directly control three computers using a Keyboard-Video-Monitor (KVM) switch. All six computers in an isle are reachable via Secure SHell (SSH) for remote control.

All participating PCs are connected over switches for the measurements forming one layer 2 segment. The network interfaces of the computers are configured to the same Internet Protocol (IP) subnet (Sec. 6.3).

9.2.1 Smart Device

To test the interaction with future Smart Devices (Sec. 3.2.3), a Do-It-Yourself (DIY) Smart Device based on the Arduino platform [Ban08] is constructed. This Arduino

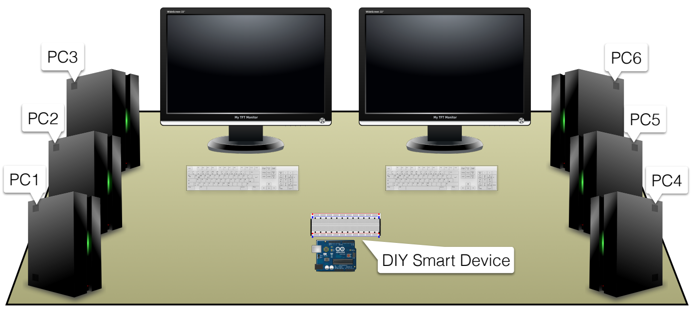


Figure 9.1: Setup of one isle for the user study.

Mega 2560 is equipped with an Ethernet shield that allows 100Mbit/s network connections for remote access. See Fig. 9.2.

The Arduino is programmed to allow remote control of all connected peripherals. For the latency measurements (see Sec. 9.3), two remotely settable timers are implemented on the device.

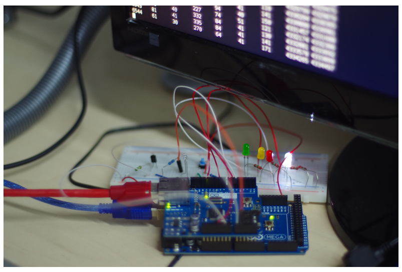


Figure 9.2: The Arduino-based Smart Device that is used for some of the measurements.

9.2.2 Reproducibility

In the science community there is ongoing discussion about the reproducibility of measurement results [BBDR+13].

The following measurements are reproducible. The identical measurements that were carried out to obtain the shown results are in a smaller scale (less participating hosts, less iterations) part of the user study (Sec. 9.6).

Eight students confirmed the measured results in four independent teams in late 2013/ early 2014 during the study (Sec. 9.6.4). In a pre-study, six students in three teams confirmed similar results in summer 2013.

9.2.3 Generalizability of the Results

In DS2OS, Orchestration workflows are implemented by coupling services with each other. Therefore the latency that is introduced by coupling services pairwise can be used as building block to compute the latency that can be expected when coupling services over the VSL. Services are either coupled over a shared KA or over two connected KAs.

DS2OS KAs are directly connected in an IP subnet (Sec. 9.2). Resulting from this topology the maximum distance between any two DS2OS services is at most one hop.

Either the other service runs on the same host (0 hops) and they share the KA, or it runs on another computing node (1 hop, two connected KAs). At most two KAs are involved, the KA of the accessing service and that of the accessed service.

Therefore measuring the latencies between two services is representative to a setup with N services that are connected in a chain (Sec. 5.6.6). N services would only involve N pairwise service interactions. The results in Fig. 9.5 are thus representative for the communication of any two services within a DS2OS site.

In the test setup a 1 GBps network is used. The communication delay between two hosts can be neglected since it is about $\frac{8*10^2 Bit}{10^9 Bit}s = 8*10^{-7}s = 0.8\mu s$ for 100 Byte packet size. DS2OS packets that transport small values have about this size.

As described in Sec. 3.2, the networks within buildings are heterogeneous and have different characteristics. For a 1 MBps connection the communication delay becomes more relevant but is still small: $\frac{8*10^2 Bit}{10^6 Bit}s = 8*10^{-4}s = 0.8ms$. For slow communication lines such as building buses of about 10 KBps (see Sec. 3.2.1) the communication delay becomes a relevant factor for the overall delay with about $\frac{8*10^2 Bit}{10^4 Bit}s = 8*10^{-1}s = 800ms$ communication delay.

These values have to be added to the measured delay values for getting comparable values for topologies where distributed KAs are connected over slower networks. Considering the technological advance (Sec. 3.2.4) it can be assumed that KAs in future Smart Spaces will run on Smart Devices that are at least connected over 1MBps.

The peripherals that are connected via *adaptation services* can be connected over slower links. This is relevant for the delay towards the Smart Device measured in Sec. 9.3.3. In that measurement the Arduino DIY device is connected with 100MBps.

9.2.4 Complex Orchestration Scenario

Besides creating the measurement results, the test services that were implemented for doing the following measurements are an example for a complex orchestration scenario. They show how the VSL programming abstraction structures and facilitates such scenarios.

The implementation of each test took only few days including the debugging. All Java classes that were written are below 300 lines for the latency measurements, and below 500 lines for the scalability measurements. This shows that VSL services can implement complex functionality with few Lines Of Codes (LOCs). The experience with the test shows the practical benefit of the separation of service logic from service state, and from inter-service communication (Sec. 5.6.5).

Listing 9.1 shows the debug output of the fully automated measurement run for obtaining the results in Sec. 9.3.3.

```
grml@pc1 ~% java -jar latencyToDeviceMeasurer.jar
07:07:27.355 [I] [DssosGlobals   ] Config read from /home/grml/ds2os/config.
     txt
07:07:27.362 [I] [StreamServer   ] Socket bound to /127.0.0.1:16610
07:07:27.363 [I] [StreamServer   ] Socket bound to /0:0:0:0:0:0:0:1:16610
07:07:27.375 [I] [NetInterfaces  ] Available interfaces:
        eth-man (eth-man) @ /fe80:0:0:0:be5f:f4ff:fe4a:486c%2 /192.168.6.1
        lo (lo) @ /0:0:0:0:0:0:0:1%1 /127.0.0.1

07:07:27.375 [I] [NetInterfaces  ] Using configured interface: eth-man
07:07:27.512 [I] [Connector      ] Connector 'system2DeviceMeasure' registered
```

```
11  with local agent 'pc6.1' @ ipv4: /127.0.0.1 ipv6: null, tcp port: 6666
12  07:07:27.513[I] [LatencyToDevice] Collecting 10000 samples.
13  07:07:27.544 [I] [LatencyToDevice] Using /pc6.2/smartGW/timer0 as timer0.
14  07:07:27.545 [I] [LatencyToDevice] Using /pc6.2/smartGW/timer1 as timer1.
15  07:07:27.576 [I] [LatencyToDevice] Using /pc6.2/smartGW/vTimer0 as vTimer0.
16  07:07:27.576 [I] [LatencyToDevice] Using /pc6.2/smartGW/vTimer1 as vTimer1.
17  Appending log to 1387174047659_latency2device.log...
```

Listing 9.1: Console output of the invocation of the latencyToDeviceMeasurer service as example for a practical use of a DS2OS service

Line 3 and 4 of listing 9.1 show the binding to stream sockets on the IPv4 and IPv6 addresses of the local host that are used for communication with the KA. The lines 9-11 show the binding to the locally running KA on port 6666. Lines 13-16 show the search results (instance addresses) for the VSL data type `/ilab/triggerTimer` for the regular VSL nodes and `/ilab/vTimer` for the Virtual Nodes.

9.3 Latency Between Services

Je crois qu'ils font ça pour remplir le temps, tout simplement. Mais le temps est trop large, il ne se laisse pas remplir. Tout ce qu'on y plonge s'amollit et s'étire.

Jean-Paul Sartre, French philosopher, (1905-1980), "La Nausée" (1938)

DS2OS is used to connect services in Smart Spaces. As shown in Sec. 8 and Sec. 9.6, the VSL programming abstraction facilitates the creation of services for Smart Spaces.

The mechanisms of the VSL introduce delays to the inter-service communication that is needed to implement pervasive computing scenarios. This delay cannot be prevented as the implementation of the scenarios without DS2OS is difficult. However, the measured delays are low enough to implement diverse pervasive computing scenarios (<O.3>). An implementation of the KAs that focuses on low latency can be expected to lower the measured delay significantly as our focus in the prototype was not on low latency.

As shown in Sec. 3.2.1, Building Automation Systems (BASs) typically handle time-critical applications on the *field level*, or the *automation level*. For DS2OS this approach is recommended too. It results in a coupling of time-critical Smart Devices using protocols that give timing guarantees.

The VSL can be coupled with activities on the field level by reflecting the actions happening on that layer into the VSL. This is a typical task for adaptation services (Sec. 8.3.1). For non time-critical applications it is recommended to handle functionality over the VSL as this enables flexible Smart Space Orchestration (Sec. 2.4.1).

Following, the prototypical implementation of the VSL μ-middleware is evaluated in different scenarios. The functionality of the VSL is fixed (Sec. 5.3.1). In the following measurements, the `set` and `get` operations of the Application Programming Interface (API) are tested. They are the basis for inter-service communication over the VSL. The evaluation of the basic interaction mechanisms makes the results of the measurements applicable to diverse services that are composed of them.

Use Case

Examples for a time-critical operations are *interactive services.* A goal of an interactive service is offering users the impression that services are reactive as they respond within small latency boundaries.

An example is a switch that is pressed by a user to switch a light on. For implementing Smart Space Orchestration, the physical switch and the light can be coupled with an *adaptation service* each. One *adaptation service* reflects state changes of the switch into the VSL, the other *adaptation service* reflects changes in the VSL to the light (Sec. 8.3.1). Fig. 8.6 shows such a hardware setup at our demo day.

Via Smart Space Orchestration, diverse actions can be linked. The pressing of a hardware button becomes a software event that can be coupled with diverse actions[46].

[46]The following video shows a virtual office demo that connects several actions with a switch using the VSL: https://youtu.be/jHByqMn6WCE.

Using the VSL for coupling real world entities introduces access control (Sec. 5.5). Via the coupling through the *adaptation services* VSL access rights are checked automatically. Only after reasoning about the button-press-event of the switch in an Advanced Reasoning Service (ARS) (Sec. 8.3.2), certain lamps could be switched on (Sec. 7.5). In this software-orchestration use case, the latency of the VSL contributes to the overall delay the user experiences between pressing the switch and seeing the light.

The results of this measurement show that it is possible to implement Smart Space Orchestration over the VSL with a delay that appears interactive or responsive to users. Criteria for such impression are introduced in Sec. 9.3.1.

9.3.1 Assessment Criteria

For being able to assess the measured delay values, it is important to know which delays are acceptable for users. The answer depends on the previous experiences of users and the use cases [SP05, EWCS96]. When typing a text on a PC, a delay up to 100ms is not noticeable. A delay up to 160ms is not noted negatively in the studies presented in [SP05]. In telesurgery, operations are reported to be well feasible up to a delay of 500ms [FLC+00]. For loading web sites, a delay of about 600ms is reported as short in the literature [JSB00]. Delays of more than 2sec are perceived as non-interactive [EWCS96, SP05].

The authors of [PBSZ13] made a study with the switch scenario, described above (Sec. 9.3). They report that delays up to 300ms were *almost not recognized* by the participants of the study in 2013. Delays from 500ms to 1000ms were *recognized but accepted*. Delays above 2200ms were *not acceptable* for the users.

Summarizing, delays below 100ms can be considered as *not noticeable*. Delays up to 300ms are still perceived *very interactive*. Delays from 500ms-1s are *tolerable*.

In the result plots shown in this chapter the delay range between 100ms and 300ms is colored for highlighting the ideal delay boundaries.

9.3.2 Setup

Fig. 9.3 shows the setup of this measurement. It consists of two distributed machines (PC1, PC2). Each machine runs one KA. The KAs span the VSL. On the right, an Arduino-based Smart Device is shown (Sec. 9.2).

Two timers (`timer_0`, `timer_1`) are implemented on the Smart Device. When the *set* function of a timer is called, it stores the current time since the start of the Smart Device in milliseconds. The *get* function returns the stored time.

Using a Smart Device, this setup goes beyond the pure measurement between services. The constellation with a Smart Device is chosen as it reflects a typical use case: a Smart Device is used as output or input for an orchestration workflow in a future Smart Space. At the same time the communication with a physical device introduces additional latency. It is relevant for the user-experienced delay (Sec. 9.3) and therefore measured. Two measurements are done, one with the Smart Device, and one without it. The second measurement shows the delays of pure service coupling.

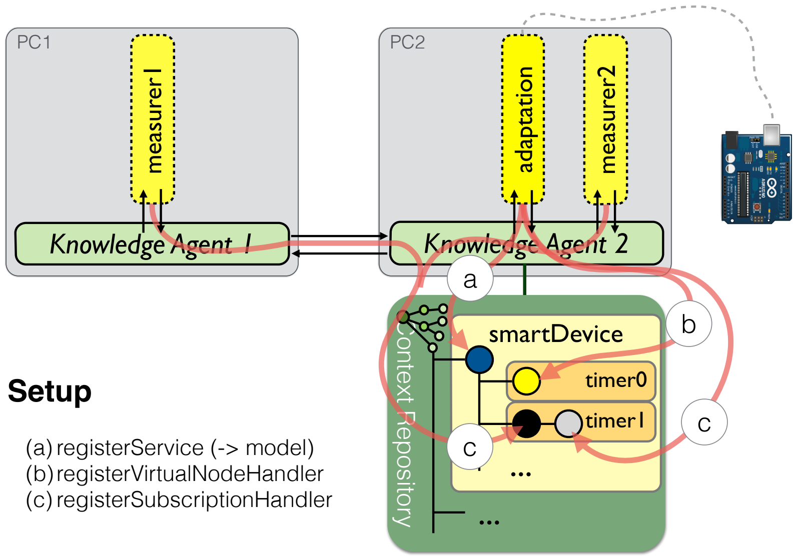


Figure 9.3: Setup of the Latency-Between-Services measurement.

Each host runs an identical instance of a so-called *measurement service* that does the measurement (`measurer1`, `measurer2`). The shown *adaptation service* (`adaptation`, Fig. 9.3) contains the logic to call the *timer* functionality of the Smart Device described before.

The context model of the *adaptation service* offers three nodes for coupling: `vTimer0`, `timer1`, and `timer1/desired`. See Fig. 9.3.

`timer0` uses a coupling via a Virtual Node (Sec. 5.3.4). Issuing a `set` on it results in *set* of `timer0` on the Smart Device. A `get` request returns the current value of `timer0` by requesting it from the Smart Device.

`timer1`, and `timer1/desired` are regular VSL nodes. They are used for service coupling over subscriptions (Sec. 5.3.2). Listing 8.3 shows an instance of `timer1` and its subnode `timer1/desired` in the VSL. A change of `timer1/desired` results in a notification callback in the *adaptation service*. The notification callback function calls the *set* function for `timer1` on the device.

Calling the *set* function for a timer on the Smart Device returns the new timer value. The returned value is stored in the context node `/timer1` as it reflects the current state of `timer_0` on the Smart Device. The described behavior implements an autonomous control loop as described in Sec. 8.3.1.

The software setup consists of the following steps:

- The *adaptation service* registers at KA2.
- The context model `smartDevice` is instantiated in the context repository of KA2 (Sec. 5.3.1).
- The *adaptation service* registers a Virtual Node callback on the context node `/timer0` (Sec. 5.3.4).
- The *adaptation service* registers a notification callback on the context node `/timer1/desired` (Sec. 5.3.2).
- Depending on the measurement scenario (see below), `measurer1` or `measurer2` searches for the location of the context model instance that is identified by the type `smartDevice` to get its VSL address.

- The participating *measurement service* registers a notification callback on the context node `/timer1`.
- The test is run.

In this measurement the delta between setting both timers is of interest (see Fig. 9.5). It shows the time difference between controlling a device via a Virtual Node (synchronous coupling, `timer0`) and via the VSL notification mechanism (asynchronous coupling, `timer1`). The measured difference is shown as (6)-(1) in Fig. 9.4.

Each test is run 10000 times. Via the registered notification callback the *measurement service* is informed when a test round is over and it can start the next one. This is shown in Fig. 9.4.

Two measurement scenarios are implemented. The first scenario measures the delay to a Smart Device as described above. The second measurement scenario runs an emulator (Sec. 8.3.3) of the Smart Device on *PC2*. The use of an emulator removes the additional delay that the communication to the Smart Device adds to the two `set` calls in the first measurement. In addition it allows to measure the absolute delays of the two different set calls as the times of the two Personal Computers (PCs) are synchronized. The Smart Device in the first scenario only allows to measure relative delays as it has its own time base.

Two runs are done in each scenario. In the first run both services are connected to the same KA. In the second run the *measurement service* and the *adaptation service* run on different hosts as shown in Fig. 9.3. The difference between the two runs shows the latency that is added by the introduction of a second KA. The result shows the latency of coupling services on distributed hosts (36ms).

9.3.3 Measurement I: Service - Service - Device

This measurement has two main goals. The first is comparing the latency of the two VSL coupling mechanisms. The second goal is measuring the time a VSL operation takes until it returns from the perspective of the *measurement service*.

Figure 9.4 shows the time-sequence diagram of this scenario. The figure shows the run where the service and the *measurement service* are connected to different KAs on distributed computing hosts. By removing the left KA and connecting the communication lines, the diagram shows the time-sequence when only one KA is involved.

The delay measurements (1)-(5) are done by the *measurement service*. The delay between the two set operations to the Smart Device (6) is measured on the Arduino and collected by the *measurement service* with two `get` requests on `/timer0` and `/timer1` of the *adaptation service* at the start of the time periods (4) and (5).

The measurement is structured into four phases as shown in fig. 9.4 on the right.

Phase I measures the execution delay of a `set` on the Virtual Node `timer0`. The set message of the service is routed via the left KA and the right KA into the Virtual Node callback of the *measurement service*. The service callback sets the value on the device. As shown in Fig. 9.4 in the time span marked with 1 on the left, the `set` command on a Virtual Node only returns when the value is set on the device.

Phase II measures the execution delay of a `set` on the `timer1/desired` VSL node. As shown in the diagram, the `set` command on a regular VSL node returns when

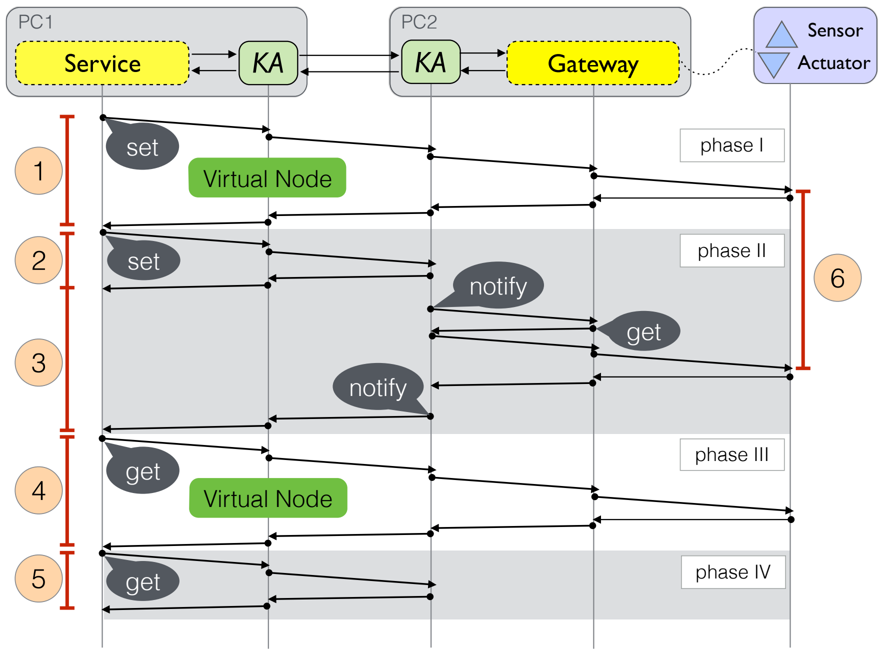


Figure 9.4: Time-sequence diagram of the Latency-Between-Services measurement.

the value is set on the destination KA (2). The processing continuous asynchronously coupled. The value change notification of the node `timer1/desired` is sent to the *measurement service*. The *measurement service* requests the changed value via `get` from the KA, sets the timer on the device, and receives the new timer value from the device. This new value is then stored into the local VSL node `timer1`.

As the measurement service subscribes the `timer1` node, it receives a notification when the *measurement service* updated the value. This coupling implements the feedback loop between the distributed services. It is an example for the intended interaction with an autonomous *adaptation service* (Sec. 8.3.1). The sum of the delays (2) and (3) is the time it takes until the service has the confirmation that the `set` operation was executed via the asynchronous coupling. In case of the synchronous coupling this information is available after the time span (1).

A typical service routine would only wait for time (2), until the `set` returns as the *measurement services* are designed to work autonomously and therefore independent of the service. If a service developer wants to implement a feedback loop in case of the asynchronous coupling, it can subscribe the context node that reflects the current state of an entity as described above. When the autonomic adaptation service (Sec. 3.7) reflects its changes to the device, it receives the feedback, reflects it to the VSL context model, and the change requesting service gets informed over its subscription. This implements a feedback loop between the hardware and the calling service.

In phase III, the Virtual Node coupling is used to query the value of `timer0` directly from the device via executing a `get` on the node `timer0`. The measured time (4) shows the time of this operation.

In phase IV, the regular VSL node that contains the updated value is queried. The time span (5) shows the latency of this operation.

Each measurement was executed 10000 times for reducing the impact of outliers, and for increasing the statistic significance of the sample. Additionally the test was repeated by four student teams independently with a lower sample size as described in Sec. 9.2.2. The lower sample size was chosen not to make the students wait too long for the results as their focus was on the evaluation of the measurement. The student results confirm the shown plot (Sec. 9.6.4).

Hypothesis

From the design of the VSL mechanisms (Ch. 5) and the time-sequence diagram (Fig. 9.4) it can be expected that a blocking call of the `set` command takes longer to return for a Virtual Node (1) than for a regular VSL node (2) on the caller site (left). The same applies for the `get` requests (4), (5).

The reason for the expected higher delay of the access to Virtual Nodes is that it requires the lookup of the corresponding service, the invocation of the callback through the connector (Sec. 6.4.3), and the access to the Smart Device.

The call of `set`/ `get` on a Virtual Node results in a callback into the *measurement service* and from there to the Smart Device. The call of `set`/ `get` on a regular node ends in the destination KA.

It can be expected that the `set` on the Virtual Node arrives faster on the Smart Device than the `set` that the asynchronous notification mechanism of the VSL "(6)-(1)" (time span (6) - time span (1)). The second measurement scenario (Sec. 9.3.4) measures the absolute times each `set` operation takes. The first mechanism only evaluates the difference between the two mechanisms.

Result

Figure 9.5 shows the delay measurement results for the different delays shown in Fig. 9.4. The prefix "v" (*vget*, *vset*) is used in the label on the x-axis to express that the operation accessed a Virtual Node.

Explanation of the Result Boxplot

The red boxplots (left) show the delays when both services are coupled over a shared KA (KA2). The blue boxplots (right) show the delays when both services are coupled over two different KAs (KA1 and KA2, see Fig. 9.3).

Each boxplot contains 99.9% of the samples within the ends of the two whisker bars (9990 samples) [WK12]. The dots show the remaining 0.1% outliers (10 samples). The lower whisker bar shows the first quartile of the measured samples, the box contains the second and third quartile. The line in the box is the median. The second quartile is below the median, the third quartile above it. The line is plotted at the median of the 99.9% used samples. The upper whisker bar contains the fourth quartile.

The red line between two boxplots shows the mean of each measurement. The lower horizontal line shows the mean of the left boxplot. The upper horizontal line shows the mean of the right boxplot. The printed value on the bottom is the value of the left mean rounded to a full number. The rounded mean of the right samples is printed

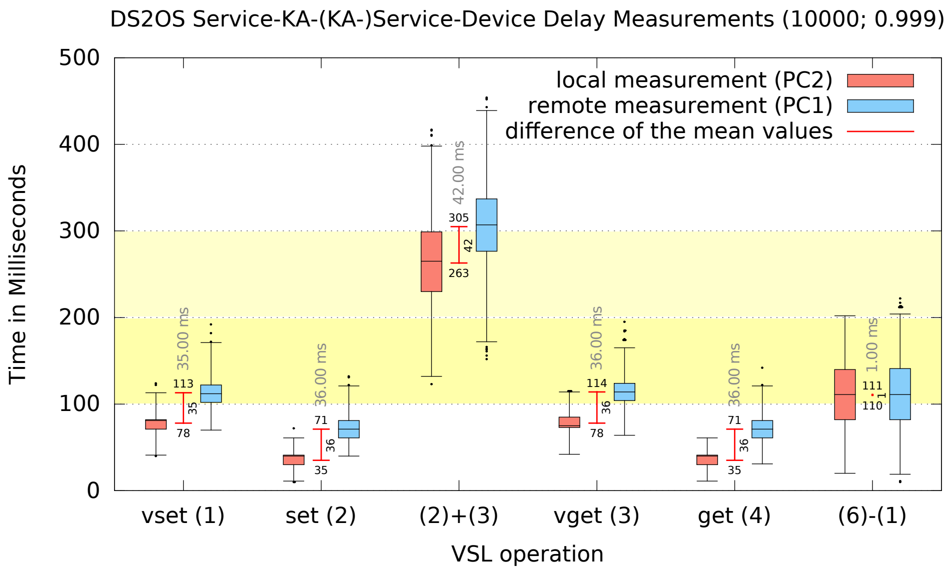


Figure 9.5: Delay Measurement Result of the 1st Measurement: Service <-> KA (<-> KA) <-> Service <-> Smart Device.

over the top line. The upended value is the rounded value of the difference between the mean of the right boxplot and the mean of the left boxplot. The gray number that is printed upended with a little distance over the upper value is the delta between the right and the left mean rounded to 2 decimal places.

Interpretation of the Results

The measurement confirms the hypothesis: at the caller, the asynchronous coupling returns faster for `set` (2) and `get` (5) than the synchronous coupling (1), (4). This means that a single threaded service can continue with its operation sooner in case of asynchronous coupling.

In the asynchronous coupling case, notifications are only sent periodically by the KAs. This is done to aggregate all notifications for a receiver for a certain time to save bandwidth. VSL notifications contain the address that changed and not the value (Sec, 5.3.2). Therefore, it is not necessary to send out multiple notifications within a shorter period of time. To obtain a value, a notification receiver will automatically query the newest value.

For the measurement the notification periodicity was set to 60ms. It can be configured in the configuration of the KA. The third measurement in Sec. 9.3.5 shows the effect of configuring different notification delays.

The plots in this chapter, in particular Fig. 9.8, show that the notification mechanism influences the resulting delays for executing the KA API functions. The time-sequence diagram (Fig. 9.4) illustrates that when setting a timer on the Smart Device via asynchronous coupling, additional delay occurs compared to the Virtual Node coupling. The resulting `set` chain on the regular node contains two notification intervals in the measured interval (2)+(3).

Interval (2)+(3) in Fig. 9.4 contains all necessary operations in the asynchronous coupling case until the service on the left knows that the `set` operation arrived at the device. As expected from the time-sequence diagram, the time span (2)+(3) is significantly longer than (1).

Comparing the `set` on a Virtual Node with the same functionality implemented over asynchronous coupling is interesting to compare the different latencies of implementing a feedback loop using the two mechanisms. However, blocking waiting on the notification in the asynchronous coupling is not good programming style. It is orthogonal to the programming methodology of asynchronous coupling of autonomous services. The caller would typically either assume that the request succeeds and continue with its operation at the end of (2), or register a notification callback on the current value of `timer1` to get an asynchronous notification as done here.

The last row of Fig. 9.4 shows the difference between triggering two different timers on the device (Sec. 9.2). The delta value is calculated as the subtraction of the delays (6) and (1) in Fig. 9.4. This is done to have the resulting delay that would occur when both `set` commands would have been executed synchronously. The measurement service implementation is linear. It only issues the `set` on `timer1/desired`, when the `set` on `timer0` returned (Fig. 9.4). This results in a start delay of (1) for the regular `set` operation. For a valid result, the assumption is that the clocks on the sender and on the smart device run synchronously.

The measurement confirms that with the asynchronous coupling it takes about 111ms longer until an operation arrives on the destination Smart Device. The plotted shorter delays in the first quartile can be explained with high values in the delay (1) going along with regular or fast operations in (2) and (3) until the value is set on the device. The over-proportional high value of (1) reduces the delta value ((6)-(1)) significantly in such cases. Looking at the measured samples confirmed this hypothesis.

The differences between the red and the blue boxes show, how much overhead is added when KA1 is introduced that has to send messages to KA2 which is running the *measurement service*. The measured additional delay is around 36ms at 1 GBps speed of the communication network between the nodes.

A notification delay of above 200ms in the third column ((2)+(3)) indicates that the operations that happen on the destination KA take more than 100ms so that the notification interval 100ms after the first notification is missed. This assumption is supported by the measured delays for `get` and `set` operations on the local KA (red). The operations in the delay phases (2) + (3) are `set`, `notify`, `get`, setting the value on the device, `set`, `notify`. Their expected latencies are shown in Fig. 9.4. The delay of the notification is measured in Sec. 9.3.5.

The measurement shows that local `get` and `set` operations on a VSL node take 35ms average. The Virtual Node operations take 78ms each. The added delay for a remote access to another KA is about 36ms.

9.3.4 Measurement II: Service - Service - Device Simulator

Setup Modifications

In the previous measurement the one-way delay of the two `set` operations cannot be measured as the clocks on the Smart Device and on the host running the *measurement service* are not synchronized. Therefore the setup is modified for the second measurement scenario by replacing the Smart Device with an emulator that runs on PC2.

All processes on PC2 have access to the same clock. The time is therefore synchronized for all services on this machine. For the distributed measurement, where the services run on distributed machines, the Network Time Protocol (NTP) is used to synchronize the hosts as shown in Fig. 9.6.

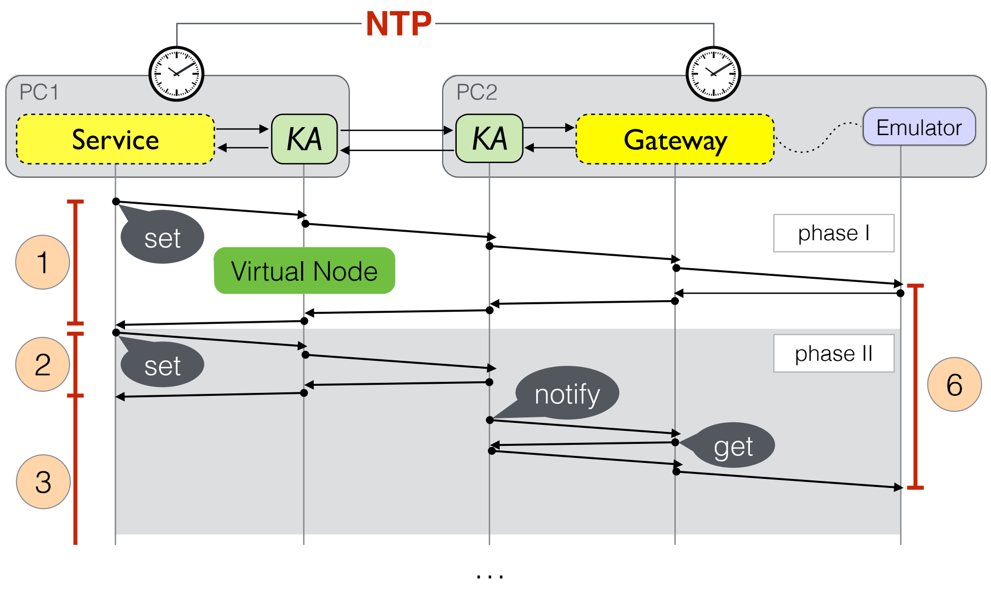


Figure 9.6: Time-sequence diagram for confirming the estimated one-way delay.

The goals of this second measurement are to measure the one-way delay that it takes when a service invokes something in another service, and to find out how much delay is caused by the communication to the Smart Device.

Hypothesis

All VSL operations that are used in the measurement are symmetric, meaning that the time until a request reaches its destination should be about half of the time until it returns at the *measurement service* ($\frac{(1)}{2}, \frac{(2)+(3)}{2}$). A significant change of the measured latency shown in Fig. 9.5 is not expected by exchanging the Smart Device with the emulator. The Smart Device is locally connected via Transmission Control Protocol (TCP) resulting in a low delay (Sec. 9.2.3).

Result

Fig. 9.7 shows the results of the second measurement.

The measured delays for accessing regular VSL nodes are similar to the first measurement (see Fig. 9.5). The access to the Virtual Nodes is about 10ms faster than in the case with an actual device (Fig. 9.5). This shows that the additional latency that is introduced by accessing the physical Smart Device (see Sec. 9.2.1) is about 10ms.

On the right two new result columns are added compared to Fig. 9.5. They show the one-way delay when accessing (`set`) a Virtual Node (labeled “vset”), and when setting

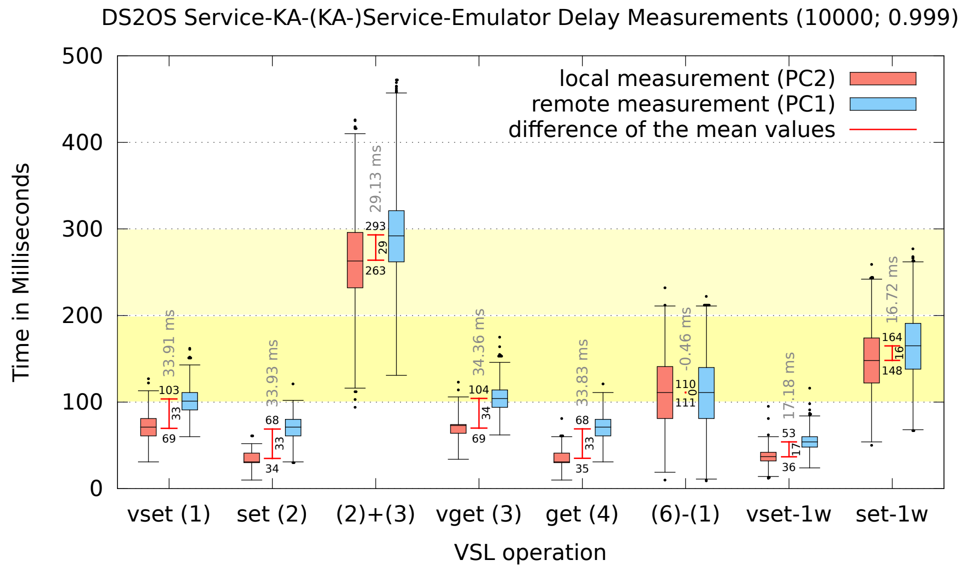


Figure 9.7: Delay Measurement Result of the second Measurement: Service <-> KA (<-> KA) <-> Service <-> Smart Device Emulator.

a regular VSL node value. The second measured delay ("set-1w") includes the `set` of the VSL node, the notification, and the retrieval of the node value with a `get` call.

The one-way set delay for the `set` on the Virtual Node confirms the hypothesis that it takes approximately half of the time until the `set` operation returns to the caller (1) for reaching the destination. The last row shows that even when using the slower asynchronous coupling mechanism via the VSL notifications it is possible to provide a user with good reactivity (Sec. 9.3.1).

The latency that is introduced by the notification mechanism depends on the configured notification interval. In the previous measurements, this interval is set to 60ms which is the default value. The reason for this choice is explained in Sec. 9.3.5. To assess the impact of this interval, the following measurement III is done.

9.3.5 Measurement III: NotificationDelay

The difference in communicating a value to a service via a `set` on a Virtual Node (denoted as `vset`) and via a `set` on a regular VSL node is that in the former case the value is sent to the service directly while in the latter the service receives a notification and retrieves the value then. As the service has to retrieve the value, the latency for the retrieval is always added in the asynchronous case.

In addition to the retrieval time, a notification delay is added in the asynchronous coupling case. The VSL sends notifications periodically only in fixed intervals. This is done to save bandwidth and Central Processing Unit (CPU) time (Sec. 9.3.3). During each interval all notifications are aggregated per user and sent out as aggregated message. The former saves network bandwidth when many changes are done within short time which is the case when a parent node is subscribed and many children are changed. The latter saves bandwidth as the message header has to be sent only once for all aggregated notifications, and it saves CPU time as only one message has to be processed at the sender and at the receiver.

The notification delay depends on the amount of VSL context repository structure changes, and the periodicity that is configured as notification interval on a KA. The trade-offs for the notification interval are low latency, when the interval is small, versus higher saving in bandwidth and CPU time when the interval is big.

To identify the influence of the notification periodicity on the overall delay of a `set`, a third measurement is done with different notification delays configured.

Setup Modifications

The setup is identical with measurement II (Sec. 9.3.4). The network delays and the KA processing delay are known to be around 36ms from the previous measurements (Fig. 9.5, Fig. 9.7). Therefore only a local measurement is done.

The notification delays in the KA are changed from 10ms to 100ms in steps of 10ms. Each setting is run 10000 times for collecting samples.

Hypothesis

The lower bound of the result is expected to be around half of the time the `set` operation takes to return plus the time it takes to retrieve the value. Taking the mean values from Fig. 9.7 this sum is 17ms+35ms=52ms.

In addition, a KA has to look up the *subscriberIDs* in the metadata of the context node (Sec. 5.2.1), get the callback binding via the connector address of the service (Sec. 6.4.1), and send the callback call to the connector of the service. The measurement will show how long this approximately takes.

As the measurement distribution should be uniform, adding 10ms of delay is expected to increase the measured overall average delay around 5ms. Some notification are expected to be sent without delay. Others are expected to have up to 9.99ms delay.

Result

Fig. 9.8 shows the one-way set delay for different notification delays.

The plotted delays contain the one-way delay for a `set` request, a notification, the two-way delay of the `get` on the node to obtain the value, and the tasks for the notification that are described in the hypothesis above.

Inside the whiskers 99.9% of the samples are plotted as in the other figures. Each bar comprises 10000 measurement samples. At 10ms delay that operations take around 125ms. Assuming an equal distribution of the 10000 samples, 5ms can be expected as average added delay (mean of the probabilities of when an event happens within the added 10ms interval). Removing the 5ms average delay, the operations take 120ms without notification delay. Subtracting the 52ms from the hypothesis, 68ms remain. The 68ms are probably the time it takes the KA to look up the subscribers and send the notifications.

A 10ms increase of the notification interval leads to an increase of the measured delay mean of about 5ms. This indicates that the samples are equally distributed. The growth is almost linear.

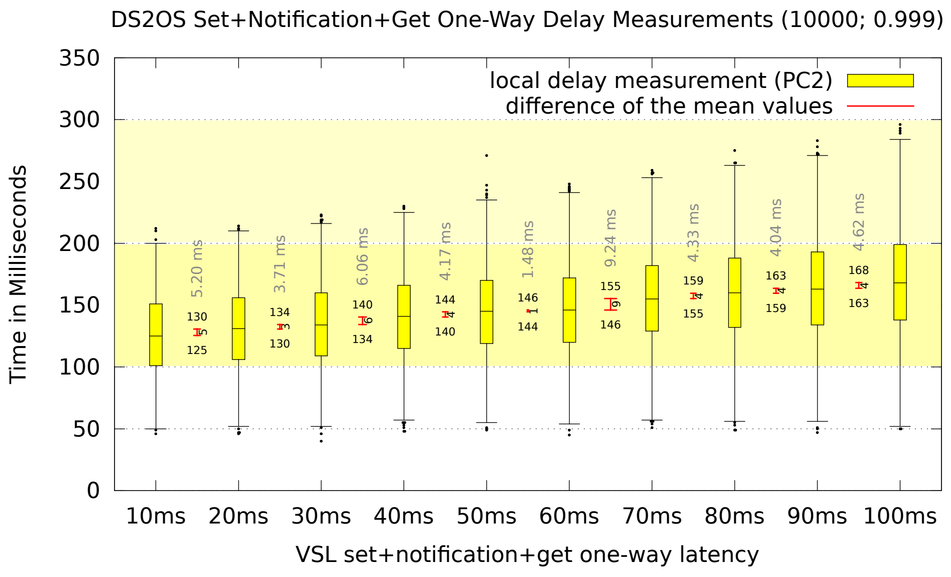


Figure 9.8: Delay measurement result of the third measurement: Service to Service One-Way-Delay at Different Notification Intervals with one KA.

For 10ms configured *notification delay*, only 10 outliers are above the 200ms interactivity boundary (Sec. 9.3.1). At 100ms configured *notification delay*, 75% of the samples are within the 200ms boundary. The mean delay of all shown delays is between 100ms and 200ms resulting in an interactive user experience. Even the worst case samples at 100ms notification delay are within the 300ms interactivity boundary (Sec. 9.3.1).

The step between 50ms and 60ms is exceptionally small. This value was confirmed in a second measurement with 10000 samples. As 60ms provide a good compromise between delay, and computational and network load (see above), 60ms are set as default notification delay in the KAs and used in the other measurements.

9.3.6 Latency Measurements Conclusion

The results of the measurement I with the Smart Device (Fig. 9.5), and the measurement II with the Smart Device emulator (Fig. 9.7) show that applying the evaluation criteria given in Sec. 9.3.1, the prototypical implementation of DS2OS offers latencies that are low enough to be perceived as highly interactive by humans.

The measurement results confirm that the asynchronous coupling returns faster to the caller in both cases, `set` and `get`. The advantage of the asynchronous coupling, that the VSL takes care of storing and providing the context, comes at the cost of increased latency until a value arrives at the destination service. The one-way delay of a VSL `set` combined with the notification and a `get` on the changed value is bigger than the delay of a `set` on a Virtual Node (Sec. 9.3.4).

The results show that using the `set` on Virtual Nodes allows coupling up to four services using one KA to remain below the 300ms boundary (Sec. 9.3.1). This result is relevant for cascaded services that were described in Sec. 5.6.6 for instance. Up to three services can be coupled over two different KAs with a resulting delay around

300ms. As described in Sec. 9.2.3, a coupling over two KAs reflects the coupling of any two distributed services in a VSL site (with equal network connections).

Measurement II shows that even with the slower asynchronous coupling, it is possible to reach one-way latencies between 100ms and 200ms as shown in Fig. 9.8.

The measurements show that the prototypical implementation of the VSL allows an interactive user experience according to the criteria from the literature given in Sec. 9.3.1. Optimal values below 200ms could typically be measured. The 300ms perception boundary was only not met in rare outliers for all tested operations.

9.4 Scalability Measurements

Petit poisson deviendra grand.

Jean de La Fontaine, French fabulist, (1621-1695), "Le petit Poisson et le Pêcheur", Fables, V., 3

The latency measurements done in Sec. 9.3 are relevant for determining the delay that can be expected when coupling services over the VSL. The result provides information about the degree of perceived interactivity that can be reached when implementing pervasive computing scenarios using DS2OS.

Another factor that is important for real-world deployments of DS2OS is scalability. Future Smart Space can be expected to consist of multiple distributed nodes running many services (Sec. 3.2.4). To handle a big amount of computing nodes and services, DS2OS must scale. The service management functionality of DS2OS is fully distributed and as such not expected to become a performance bottleneck as each node is managing its services locally, coordinated by the Site Local Service Manager (SLSM) (Sec. 7.2).

A critical factor for scalability is the performance of the VSL under high load as it is the basis for all services, providing the connectivity, data exchange, and data storage. The goal of this test scenario is evaluating the performance of a VSL deployment. It consists of multiple computation nodes that run multiple services under high load.

Instances of VSL context models consist of self-contained tuples (Sec. 5.2). The back-end of the implementation is a scaling database (Sec. 6.3.5). Therefore the amount of context that is stored in a VSL instance is not expected to influence the performance significantly. Primarily the amount of parallel accesses to context is expected to limit the VSL performance. Therefore it is measured in this section.

The VSL context synchronization provides overhead (Sec. 6.4.4). Sec. 9.5 shows that the implemented context synchronization mechanism uses incremental updates. Therefore not the mount of context but the amount of changes determines the overhead that is needed to provide the implemented VSL type search functionality. In this test, the context is not changing after the initialization phase. Therefore the synchronization overhead between the KAs is low.

As before, the term "vget" is used to describe a VSL `get` API operation on a Virtual Node. The term "vset" is used to describe a VSL `set` API operation on a Virtual Node (Sec. 5.3.1).

9.4.1 Assessment Criteria

The most important assessment criterion from the perspective of a service is the latency per request on the client. It determines the delay each service experiences. For this latency, the criteria described in Sec. 9.3.1 apply.

The overall throughput in handled requests is interesting as it gives information about how a VSL KA node behaves under high load. If the measured throughput remains at an upper level and does not decrease significantly when more load is added, this indicates a high dependability of the VSL under high load.

9.4.2 Setup

Fig. 9.9 shows the setup for this measurement. The topology consists of 36 machines that are connected over 1 GBit/ s Ethernet forming one IP subnet that is switched over two Gigabit switches per isle as described in Sec. 9.2.

Three different service roles are involved: a *control service*, a *server service*, and many *client services*. The *control service* manages the measurement. It triggers the different measurement runs on the *measurement clients* and the *measurement server*, and it collects the log data. The *server service* is the service that is accessed by the *client services*. It answers the `set` and `get` requests on the regular and Virtual Nodes of the VSL. The *client services* access the *server service* and measure the latency during a measurement run. For a test, a start time is communicated to all participating *measurement services*. At that time all services start accessing the *measurement server* simultaneously.

Services are coupled pairwise as in the previous measurement (Sec. 9.3).

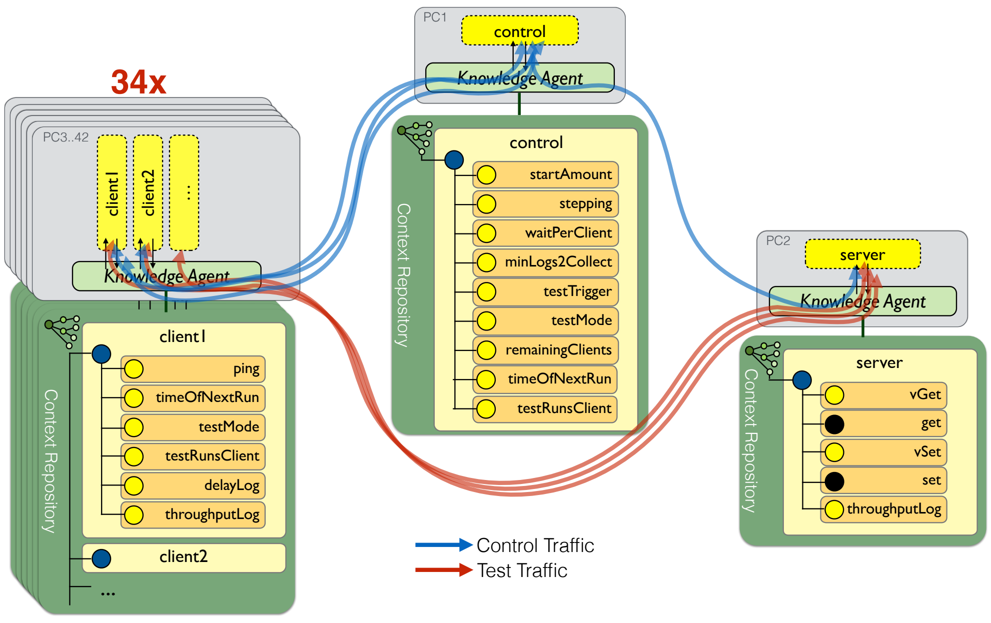


Figure 9.9: Setup of the Scalability Measurement with the VSL context models.

Each machine, named PC1 to PC36, runs a Knowledge Agent (KA). PC1 runs the *control service* and the CMR service (Sec. 6.6.2). PC2 runs the *server service*. PC3-PC36 run 50 *client services* each, resulting in 1700 *client services* distributed on 34 machines that can be used for accessing the *server service* in parallel.

Context Model

All relevant parameters of each service are exposed as Virtual Nodes over the VSL. Only the two test nodes for the `set` and `get` tests are regular VSL nodes in the *server service* as shown in Fig. 9.9 on the right in black. The Virtual Nodes are used to configure the services[47].

[47]The test showed how exposing the current configuration of a service over the VSL facilitates the debugging of services (Sec. 5.6.5).

Control service: The *control service* context node `startAmount` is used to set the starting amount of *client services* that should be used for the test run. The `stepping` context node defines the interval, the control service should step down to zero during the test. The `waitPerClient` is the time span the *control service* should add to the current time per client service to determine the start time for the measurement that is communicated to the clients to initiate the measurement.

The amount of collected logs varies with the amount of participating clients per test. One log is collected per client per test. The `minLogs2Collect` node defines how many logs should at least be connected per test round. If less logs are collected the test round is repeated until this minimum is reached.

The `testTrigger` is a Boolean variable that tells the control service to start the measurement. It can be set from outside, e.g. via the *console service* that was used throughout this document for showing the content of the context repositories.

The VSL node `testMode` contains the current test mode identifier out of the set {`vset`, `set`, `vget`, `get`}. This variable is queried by each client service when it receives a start time to find out which test mode it should run at the start time. The `remaining-Clients` VSL node contains the Identifiers (IDs) of all participating *client services*. It is set to 100 in this test. The `testRunsClient` contains the amount of continuous requests, a client service should issue in the next test round.

When a client service finishes its test run, it issues a `set` with its own *serviceID* on the context node `remainingClients` on the *control service* to report that it finished. When all participating *client services* have indicated that they finished, the *control service* can continue with its test workflow. This results in an asynchronous coupling between the *measurement client services* and the *measurement control service*.

Client service: Each *client service* contains a `ping` VSL node that returns "pong" to indicate that the service is still responsive to the requester. The `timeOfNextRun` node is used to set and query the currently set start time for the next test run. The `testMode` node is used to `set` and `get` the test mode ID that is used for the next test run. The `testRunsClient` node is used to `set` and `get` the amount of continuous requests that will be issued in the next test run at `timeOfNextRun`.

The `delayLog` node returns the delay log data of the *client service* from the last test run. The `throughputLog` node returns the throughput log data of the *client service* from the last test run. The latter nodes are used by the *measurement control service* to collect the logs from the *measurement client services* after a run.

The *server service* contains the four nodes that are used for the four test modes, `vGet` for testing the `get` access to a Virtual Node, `get` to test the `get` access to a regular VSL node, `vSet` for testing the `set` access to a Virtual Node, and `set` to test the `set` access to a regular VSL node. The `throughputLog` node returns the throughput log data of the *server service* from the last test run. As the service only uses the VSL, it can only provide throughput measurement data for the requests on Virtual Nodes as the regular node requests end up in the VSL and do not reach the service.

A test run consists of synchronized bursts of 100 requests that are sent from all participating *client services* in parallel. To synchronize the burst, all machines are synchronized via NTP as in the previous measurement (Sec. 9.3.4).

During a test the delay of the four operations, `set` and `get` on regular and virtual VSL nodes, is measured on each client service for each of the 100 requests per test round. The *client services* also measure the amount of requests they get through per second. This throughput is also measured on the *server service*. As the KA implementation is not changed for the measurement, only the throughput for the access to the Virtual Nodes can be monitored as the requests to the regular VSL nodes are directly handled by the KAs. For getting measurement results for these accesses, the aggregated throughput of all participating *client services* is used.

Description of a Test Run

A test run is characterized by the amount of participating clients. Each test run consists of four rounds for measuring the `set` and `get` access to the regular and virtual nodes of the server service's exposed context.

To determine the available *client services*, the control service runs a VSL type search on the type `performanceMeasurement/measurementClient` (Sec. 5.7.2). This type is the type of the context root node of each client (blue in Fig. 9.9). The VSL type search returns the addresses of all available client service instances.

The *control service* contains an outer loop that counts down from the `startAmount` in intervals of `stepping`. Inside the loop is another loop that iterates over the four test modes. Inside those two loops the test rounds are started.

At the beginning of a test round, the *control service* randomly selects participating *client services*. Available *client services* are identified by a VSL type search. The selection of services is randomized to emulate random activity inside a DS2OS site. As they are chosen randomly, it can happen that certain computation nodes have exceptionally high (many services used) or low load (few services used) during a test round.

The `ping` node of each selected service is accessed to check if it is available for the test. If it is not available, another client service instance is chosen. When the required amount of participating client service instances is collected, the *control service* sets the `timeOfNextRun` in each client service. Before returning the request, the client service queries the `testMode` for the run and the amount of requests to be used in the next test round (`testRunsClient`).

The *control service* informs the *server service* about the start time of the next test round. This time is used as relative time 0 in each throughput log file. The *server service* confirms its return immediately. It writes its ID to the `remainingClients` node. This makes the *control service* add the *server service* to the list of services to collect logs from at the end of the test round.

The value of the context node `testMode` defines the test. At the `timeOfNextRun` all *client services* issue `testRunsClient` many sequential requests towards the *server service* according to the current test. During the resulting request burst, they collect the request delay and request throughput statistics. The statistics are logged relatively to the start time of the test round (`timeOfNextRun`). After finishing with the configured amount of requests for the test round, each client sends its serviceID to the `remainingClients` VSL node of the *control service*.

When all used *client services* returned by setting their ID to the `remainingClients` Virtual Node, the control service starts collecting the log data from each node that participated.

If logs from at least `minLogs2Collect` client services were collected, the next test round with the next test mode is run. If not, the test round is repeated with new client service instances until logs from `minLogs2Collect` many clients are collected. The amount of collected logs depends on the amount of participating clients as described above.

When all test modes are run, the outer loop decrements the amount of participating clients and the test mode loop starts with the first test mode. When the then decreased amount of participating clients is below zero, the test ends.

Fig. 9.10 shows the four test rounds in the time-sequence diagram.

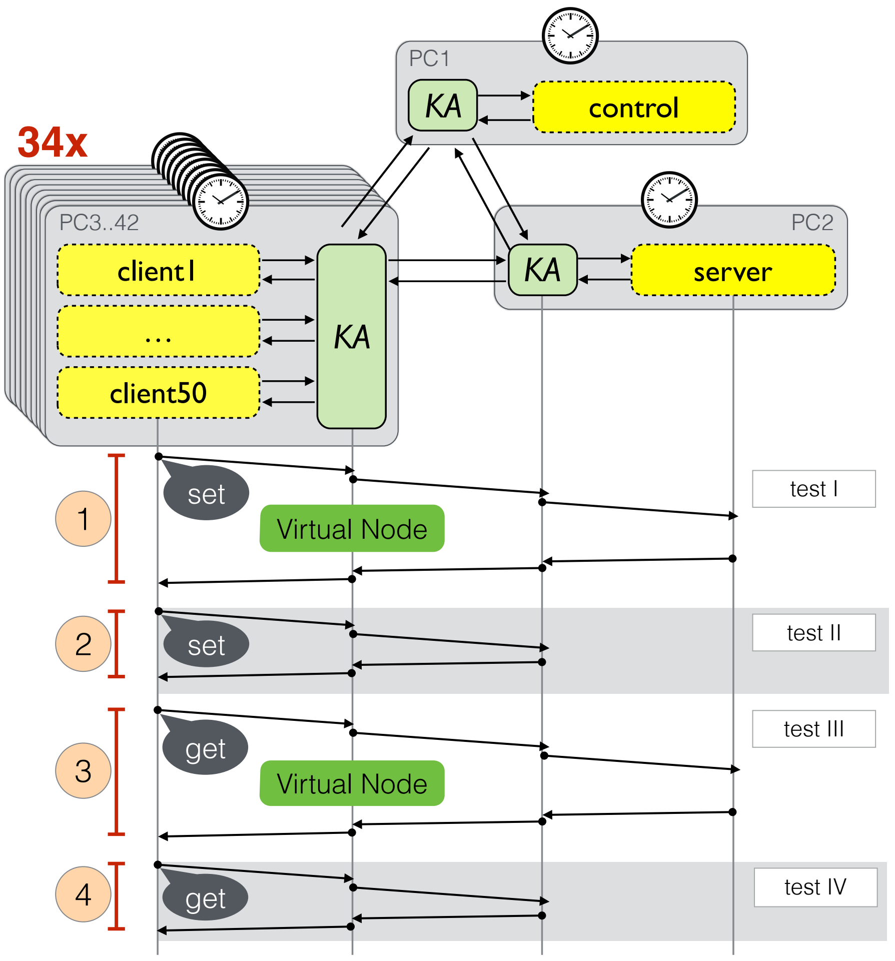


Figure 9.10: Setup of the Latency-Between-Services measurement.

Worst-Case Scenario

DS2OS is designed as distributed system to make use of the distributed resources that are likely to be present in future Smart Spaces. The DS2OS VSL is designed as distributed system that consists of identical networked KA peers (Sec. 6.3). The distribution of context and the resulting expected distribution of the parallel context accesses make the performance of the VSL scale up automatically with adding more KA peers. Adding a new computing node and running a KA on it results in more storage, network bandwidth, and computational power for accessing the context.

The VSL middleware manages the distribution transparently (Sec. 5.6.1). This scaling applies to the DS2OS service management architecture the same way as each added physical computing node offers additional computational power to run services in a transparent way as the management is done by DS2OS.

The test setup of this measurement is a worst-case scenario. It assumes that all *client services* are accessing a single *server service*. This is orthogonal to the described design of DS2OS as distributed system of peers (Sec. 6.3). While the setup is good for a stress test of the VSL, it can be expected that a real-world deployment will be significantly more distributed. Services on distributed KAs are likely to access each other more distributed. This leads to fewer parallel requests on one KA and lower delays compared to this test as distributed accesses do not interfere in the VSL (Sec. 9.4.3).

The result of this test is meaningful for more distributed service access patterns as well since it shows the performance of a KA (server service) under different parallel loads. All KAs serve requests independent from each other. Therefore, the expected delay when using the KA on a certain node primarily depends on the current concurrent access to this node. Only the load to synchronize between the KAs is additional overhead (Sec. 6.4.9).

Generalizing, the result of this measurement shows the delay that can be expected from a service (*server service*) under a load of *n client services* that sequentially access the target service 100 times each. For example, the measurement result for 26 concurrent accesses, shown in this section, is representative for any KA node that has 26 concurrent accesses over a period of 100 requests.

With more KAs in a DS2OS site, the synchronization overhead increases. As described in Sec. 6.4.9, the overhead is mainly proportional to changes in the VSL and not to the amount of KA instances running. Besides a hash over its structure, each KA is only distributing the structure changes in its local Knowledge Object Repository (KOR) since the last *alive ping*. Most processing effort for incoming *alive pings* is required for reflecting contained structure updates of a remote context repository into the local KOR. If there are no structure updates contained, the processing of an *alive ping* is basically limited to comparing one table entry (Sec. 6.4.9).

The test results with one *client service* corresponds to the results in the remote setup in the previous measurement (Sec. 9.3.4). An increased delay compared to in Fig. 9.7 shows how much overhead delay the synchronization of additional KAs introduces.

The *client services* collect the delays per request as in the latency measurement (Sec. 9.3). As this experiment consists of 100 consecutive requests per client, the *client services* collect the throughput in requests per second from the beginning of the request burst they send until they received all responses (Fig. 9.12).

The *server service* collects the throughput in answered requests per second since the start of a test round. The KA is not changed. Therefore, the *server service* can only measure the throughput for the Virtual Node requests. Requests to regular VSL nodes end up in the KA and not in the *server service*, not producing any load in it that could be measured by the service.

9.4.3 Measurement I: Big Setup

Hypothesis

If the implementation scales, the expected delay measurements will ideally show a linear growth of the delay with a growing number of concurrent node accesses. While a linear growth indicates good scalability, an exponential growth indicates a bad scalability.

The serving of context via Virtual Nodes is expected to perform worse under high load than the serving of context via regular nodes. The reason is that the used implementation of the connector is single threaded, allowing to process concurrent requests only sequentially. In addition, it requires additional look-ups, and a socket connection from the KA to the service that registered the virtual node (Sec. 6.3.3).

The throughput on the server side is expected to be proportional to the amount of parallel requests until saturation is reached. It is expected to increase with an increasing amount of parallel requests until a maximum amount of requests that can be served in parallel is reached. Ideally it will not decrease from that maximum throughput value when the load increases further.

For the client the opposite throughput pattern can be expected. With an increasing number of client services that access a server service in parallel, the per-service throughput is expected to decrease.

Resulting Plots

The data shown on the following plots are explained in Sec. 9.3.3. The resulting plots for the different kinds of requests are ordered in the same way for the first two plots. The `set` operations are on the left side of Fig. 9.11 and Fig. 9.12. The `get` operations are on the right side. The access to regular nodes is shown at the top of the pages, the access to Virtual Nodes is shown on the bottom. This order is chosen for comparability between the `set` and `get` operations.

In the next two plots (Fig. 9.13, Fig. 9.14), the throughput of the aggregated clients and the *server service* is compared. Therefore the order is changed, showing the `set` requests at the top and the `get` requests on bottom. The left side shows the throughput on the server. The right side shows the aggregated throughput of the clients.

Delay per Client

Fig. 9.11 shows the delay plots. The linear growth indicates that the implementation scales linear with the amount of participating clients. Each additional client increases the delay of regular node access by about 1ms. Each additional client increases the delay of Virtual Node access by about 2.5ms. As described before in this section, reasons for the increased delay include the single threaded implementation of the used connector, and the additional computational effort at the KA for identifying the receiver and invoking the callback.

Sec. 9.3.1 described that a delay of 2s is at the upper limit of delays that can be perceived as interactive by humans. Fig. 9.11 shows how many parallel requests per

KA can be handled within a certain timeframe. By adding up the shown delay of the VSL and the execution time of the application logic of a service, the amount of parallel operations within a specific timeframe such as the given 2s can be estimated.

For services that use regular nodes and notifications one `get` and one `set` operation can be carried out in about 1s under a load of 400 simultaneous requests per KA instance. This leaves one second for other operations to remain below the 2s limit (Sec. 9.3.1). As Virtual Nodes directly deliver their value to a requester via service callback, 300 parallel requests on a Virtual Node can be handled within 2s.

The sample distribution shows that the tendency in the measurements is towards lower delays. 25% of the samples are significantly better than the mean value. Analyzing them could give indications for improving the prototype.

Time outs are represented as dots below the x-axis of the plots ($<\frac{1}{1000}$ of the samples). The gray number above the amount of clients shows the amount of samples that were taken into account for the plot. The expected amount of samples is the number of clients multiplied by 100 as each client runs 100 sequential requests per test.

The box including the second and third quartile is narrow. This shows that the standard deviation of the sample set is low.

Throughput per Client

Fig. 9.12 shows the throughput in requests per second per client. Though it could be computed as inverse of the delay, the throughput was measured independently to ensure that the *measurement services* are sending their requests out continuously and not sparsely.

Different colors represent different measurements. In the legend, the numbers in parenthesis next to the amount of participating clients show the amount of collected samples that were used for the plot. As described in Sec. 9.4.2 the *control service* is configured to collect at least 500 samples (*minLogs2Collect*). When the amount of collected samples is below 500 another test round is started. In case of 400 *client services* this results in 800 samples for instance as each test round collects 400 samples.

The plot confirms that the requests were sent as a bulk, and not sparsely one request over a longer period. Fig. 9.12 shows that the client throughput is continuously at a similar level for the access to regular nodes (top). The throughput graph for the Virtual Nodes shows higher deviations. This could indicate that the single TCP connection to the single threaded connector is a bottleneck. Further investigation is future work.

Fig. 9.11 and Fig. 9.12 show the overall time of a measurement round. The measured throughput values for the 100 requests per client service (Fig. 9.12) are approximately the inverse of the delay (Fig. 9.11) as expected for continuous probing.

Fig. 9.12 shows a lower boundary. The throughput does not go below this boundary. This indicates that the implementation can handle a high load of parallel requests. The low variation between the sample values when saturation is reached confirms a linear growth of the delay/ a linear decrease of the throughput with a growing amount of parallel requests.

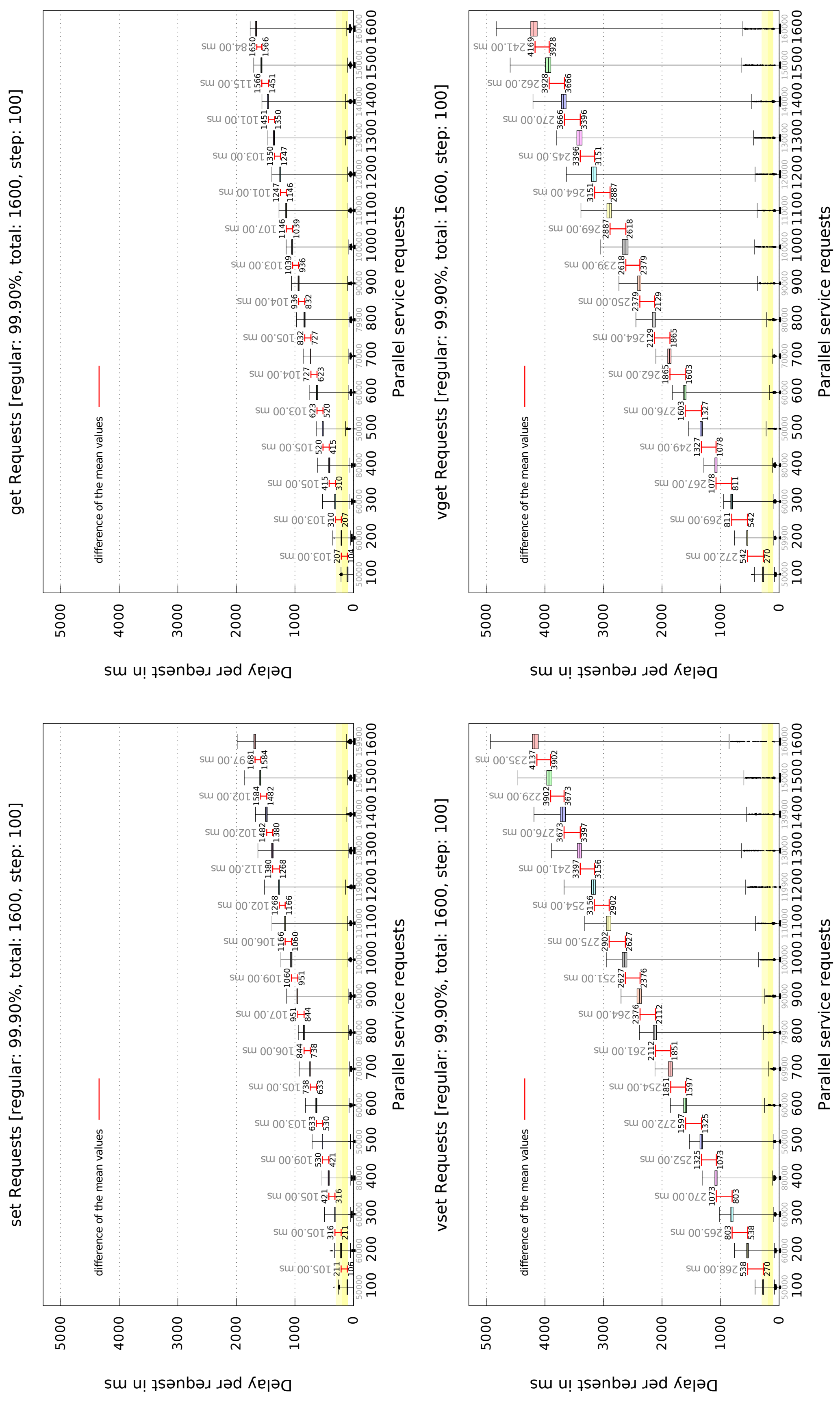


Figure 9.11: Measured delay in ms/request of 100 sequential requests sent in parallel from 100-1600 client services randomly selected from 34 hosts with 50 services on each KA.

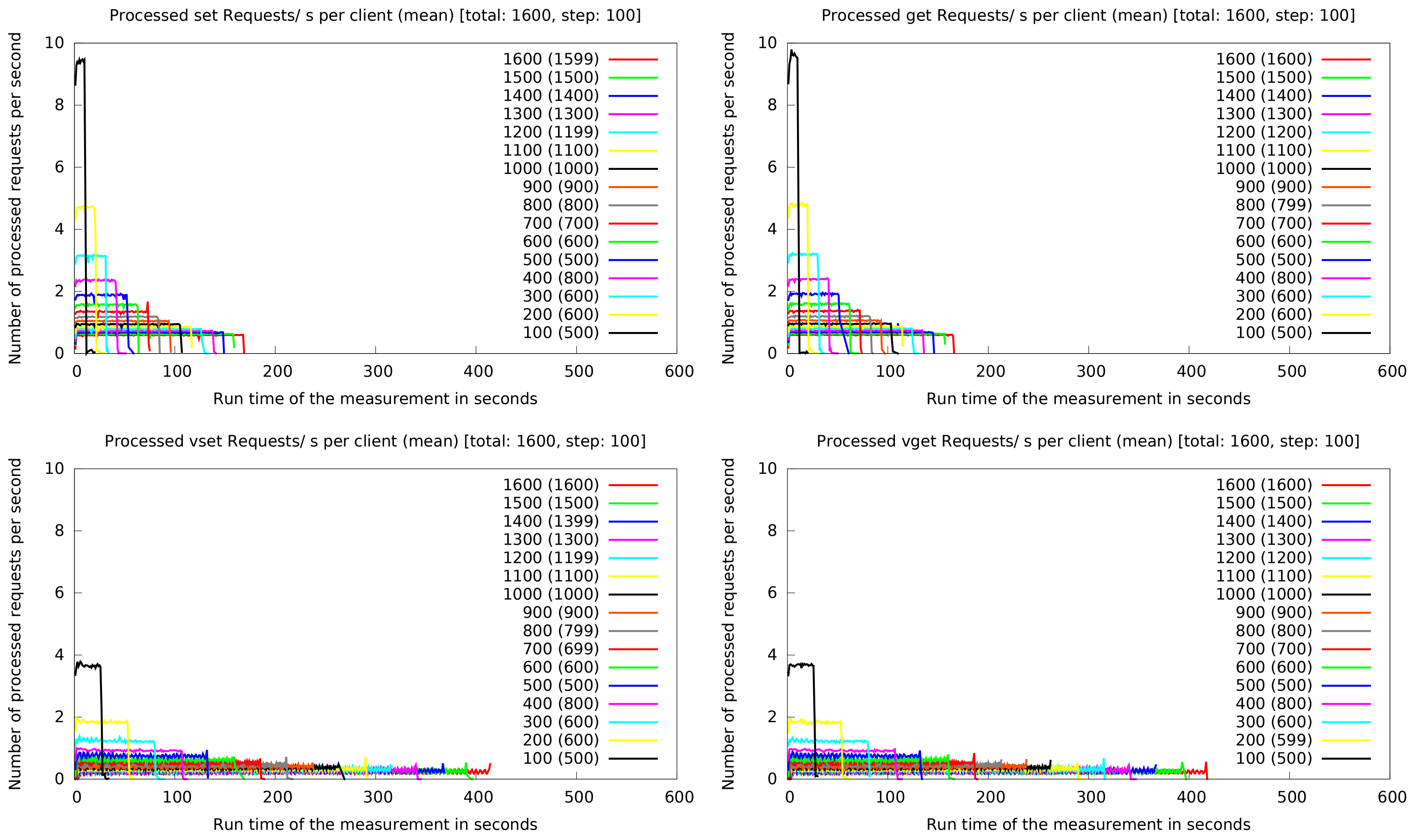


Figure 9.12: Throughput in requests/s of 100 sequential requests sent in parallel from 100-1600 client services randomly selected from 34 hosts with 50 services on each KA..

Overall System Throughput

The *overall system throughput* shows the amount of requests one VSL KA can handle per second.

The four plots in Fig. 9.13, and Fig. 9.14 have a different order than those before. This is done as the access to the regular VSL nodes (Sec. 5.3.2) cannot be measured in a service. It is transparently handled by the KA. As the KA was not changed for this measurement, the aggregated number of *client service* requests is plotted. To confirm that the aggregate shows a valid amount of requests, the aggregated throughput from all clients and the measured throughput for Virtual Node access are plotted next to each other in Fig. 9.14.

Fig. 9.13 shows the evaluation for *regular nodes*. As *regular nodes* are served directly via the KA, no measurement results are available on the left (server side). The aggregated client throughput is plotted on the right. It shows that the *measurement server* service can continuously serve around 950 requests per second. The different lengths of the tests follow directly from the previous plots as more overall requests are sent.

The spikes are probably introduced either by the processing in the network, at the receiving KA, or in the *measurement client*. The plot in Fig. 9.14 shows that the server serves continuously while some clients see a spike.

Fig. 9.14 shows the throughput for the Virtual Node access. As Virtual Node requests end in the *measurement server* service, its measured values are plotted on the left.

For comparison the corresponding aggregated *measurement client* throughput is plotted on the right. It confirms that the aggregated throughput is closely correlated to the throughput at the *measurement server*. The comparison confirms that the previous aggregated plot (Fig. 9.13) is likely showing representative throughput values for the server.

The delay at the beginning of the aggregated *measurement client* side shows that it takes a short time until the answers arrive. It also shows that the clients start synchronously. The spikes on the aggregated *measurement client* side show that some services had irregular answer behavior as described before. That the spikes are not shown in the left plots expresses that the server continuously serves around 400 Virtual Node requests per second.

The plots show that the served requests' delay varies around 400 requests/s with a growing amount of parallel requests. This could be an indicator that the single connection to the service for serving the Virtual Node requests via callback (Sec. 5.3.2) is a limiting factor. This assumption is strengthened by the lower variation that is shown in Fig. 9.13. In contrast to the current connector implementation (Sec. 6.3.3), the KAs are multi-threaded and accept multiple parallel TCP connections.

9.4.4 Measurement II: Small Setup

A real-world DS2OS deployment may be smaller than the testbed used for obtaining the results in Sec. 9.4.3. In this second scalability measurement, the test remains as described in Sec. 9.4.2 but the size of the testbed is reduced.

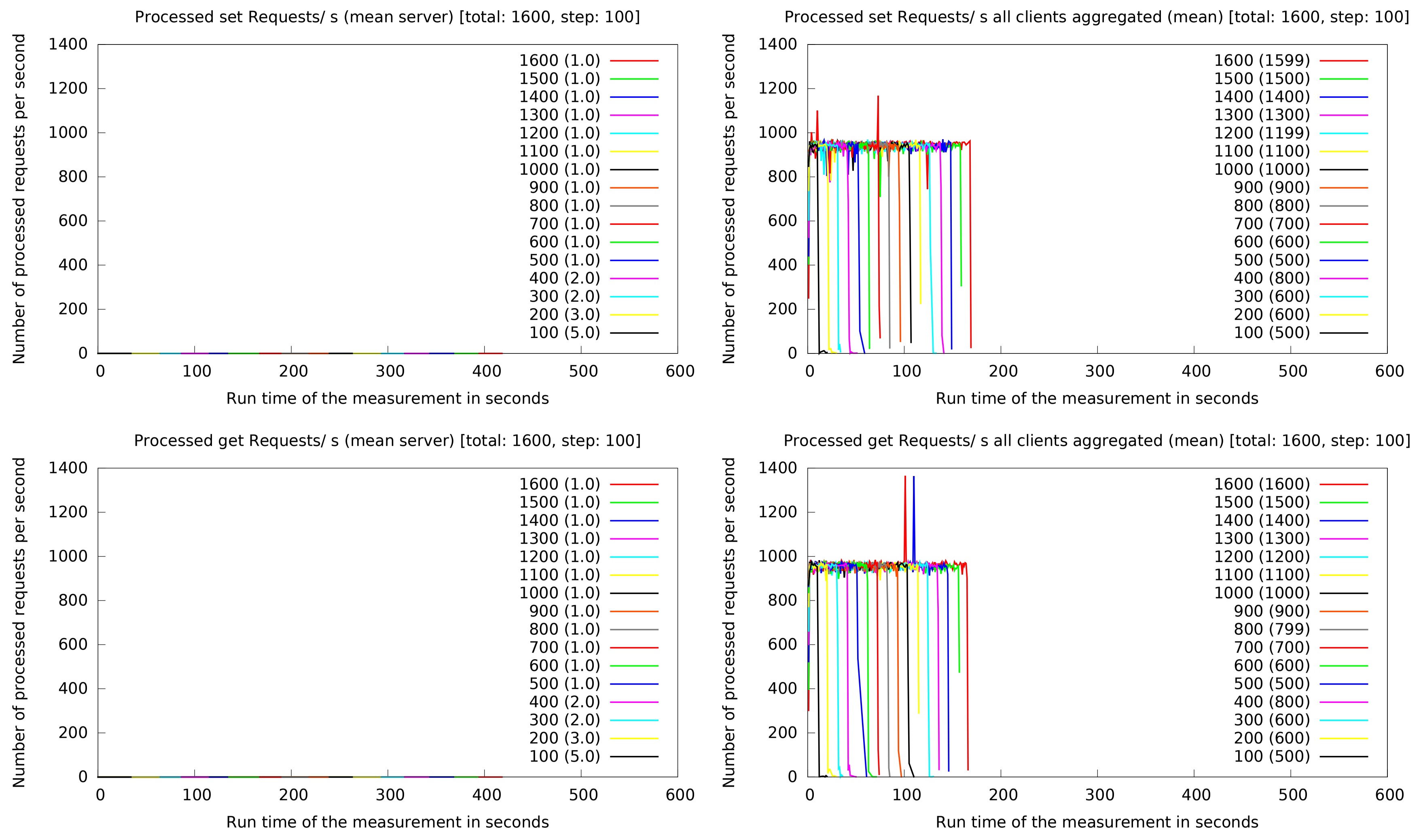


Figure 9.13: Server throughput for **regular VSL node access** in requests/s of 100 sequential requests sent in parallel from 100-1600 client services randomly selected from 34 hosts with 50 services on each KA. These requests do not end up in the server service. Therefore it cannot provide measurement results on the left.

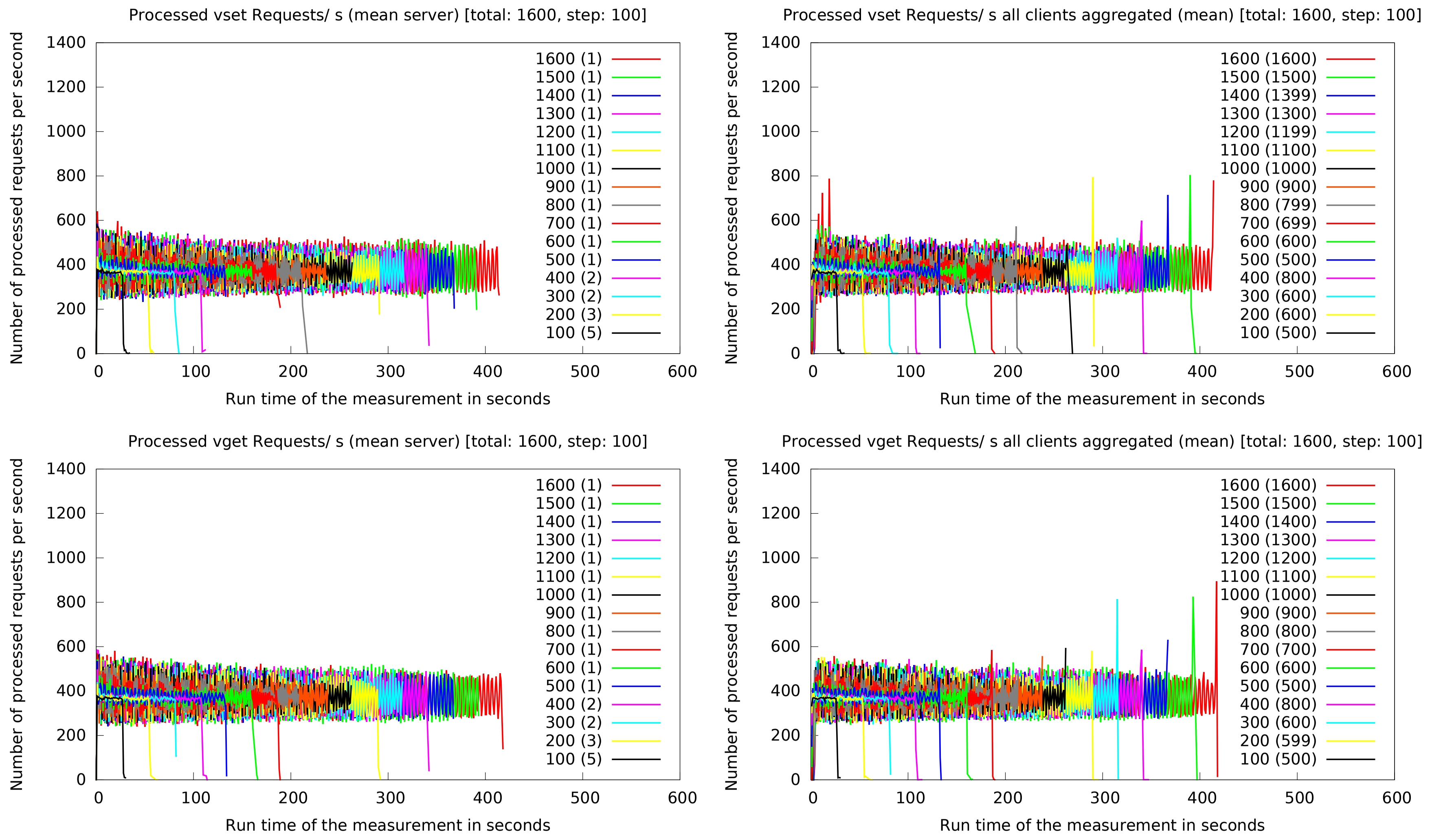


Figure 9.14: Server throughput in **Virtual Node access** in requests/s of 100 sequential requests sent in parallel from 100-1600 client services randomly selected from 34 hosts with 50 services on each KA. Left: server; right: aggregated client throughput for comparison.

The results of the previous measurement show the behavior of the VSL under high load. This test examines the delay and throughput with fewer computing nodes in the lower range (1-226 parallel requests) in more detail. Such a setup can be considered closer to a real world Smart Space setup. The measurement setup is analog to the one before. All client services send their requests to one service and one KA only.

Like before, the measurements in this section measure the behavior of a single node under load. The VSL is designed as system of distributed peers (Sec. 6.3). Using multiple distributed KA nodes can be expected to provide a significantly higher performance. It will be about the sum of the independent KAs multiplied with their requests per second.

This measurement shows the performance of a single node. Depending on the access patterns of services, and the topology of KA nodes, all KAs can be considered independent. As a result, each KA can ideally handle the amount of requests/s shown in Fig. 9.17. Running the experiment against ten *distributed measurement server service* will approximately multiply the overall measurable throughput by ten as the *distributed measurement server service* are independent and the synchronization overhead is low (Sec. 9.5).

Setup Modifications

The setup is modified to run within one isle (see Sec. 9.2). Instead of 34 KAs for hosting client services, only 5 computing nodes are used. See Fig. 9.15.

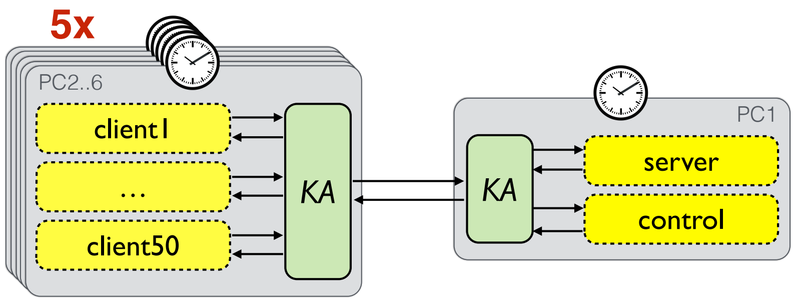


Figure 9.15: Small setup for the Latency-Between-Services measurement.

The server service and the control service run on the same machine and not on separate ones as in Fig. 9.9. On each KA that hosts client services, 50 services are started as before. This results in 250 services and 6 KAs running. The KAs synchronize every 30 seconds throughout the test as before (Sec. 9.5).

Hypothesis

Even though the size of the hardware setup is reduced, similar results to the previous test are expected. The performance is expected to be a bit worse than in the previous run for the same amount of participating client services as the services are distributed over fewer computing nodes resulting in statistically more client services sharing the same KA.

That the server service and the control service run on the same machine is not expected to cause a significant performance decrease as the control service only receives the finish signals from the distributed client services during a test run.

Result

The measurement results start with the delay plots of the four tests, `set` and `get` against regular and Virtual Nodes. See Fig. 9.16. For an explanation of the boxplot refer to Sec. 9.3.3.

The gray numbers at the bottom of each boxplot show how many samples contributed to the plot. This number is always around the next multiple of the amount of participating client * 100 that is above 50000 (see `minLogsToCollect` in Sec. 9.4.2), e.g. 226 (clients) * 100 (requests) * 3 (test runs) = 67800 (samples) which is more than 50000 samples.

The black dots at the x-axes represent time outs. Time outs happened in rare cases as shown. They are handled as outliers in the boxplot as they fall inside the interval of $\frac{1}{1000}$ of samples that was configured as outlier. For calculating the mean the outlier values are used regularly as 10s which was the TCP socket timeout in the setup.

As expected, the delays for one access are comparable to the results of the performance measurement (Sec. 9.3). The operation of the server service is comparable to that of the Smart Device emulator in Sec. 9.3.4. The results to compare with are those for the remote requests (blue box) in Fig. 9.7.

The plots show that the access to the regular VSL nodes is typically within the 300ms interactivity boundary (see Sec. 9.3.1) for up to 226 parallel requests. The mean of the delays is higher for the Virtual Nodes. Up to 100 Virtual Node requests can typically be handled within 300ms until the requester receives the result.

The gradient of the Virtual Node requests is about double compared to that of the regular node requests. This could have different reasons as described before.

- Virtual Node handling requires a lookup and a socket connection towards the connector of a service to be done at a KA.
- The server service communicates with its KA over the regular VSL protocol messages (Sec. 5.3.1) while the KA accesses its database directly without using a second socket and packet-based communication internally.
- The server service is only connected via one socket while the KA accepts an unrestricted amount of parallel socket connections.
- The server service implementation in the test is single-threaded while the KA is implemented as multi-threaded component with as many independent threads as cores are available on the system.

That the single threaded implementation of the *server service* causes the higher delays is unlikely as the response task is very simple and not expected to create high load. Most probably the significantly stronger performance decrease of the Virtual Node access comes from the single-threaded and single socket implementation of the connector that the service uses, and from the additional delay that comes from encoding end decoding the information for the exchange between the KA and the server service.

The second result set shows the mean throughput in requests per second that each client service experiences. See Fig. 9.17. The number in brackets in the legend shows the amount of log files that contributed to the respective line. Again the number is the next multiple of the participating clients that is over 500. Each log contains 100 requests distributed over the runtime of the test round.

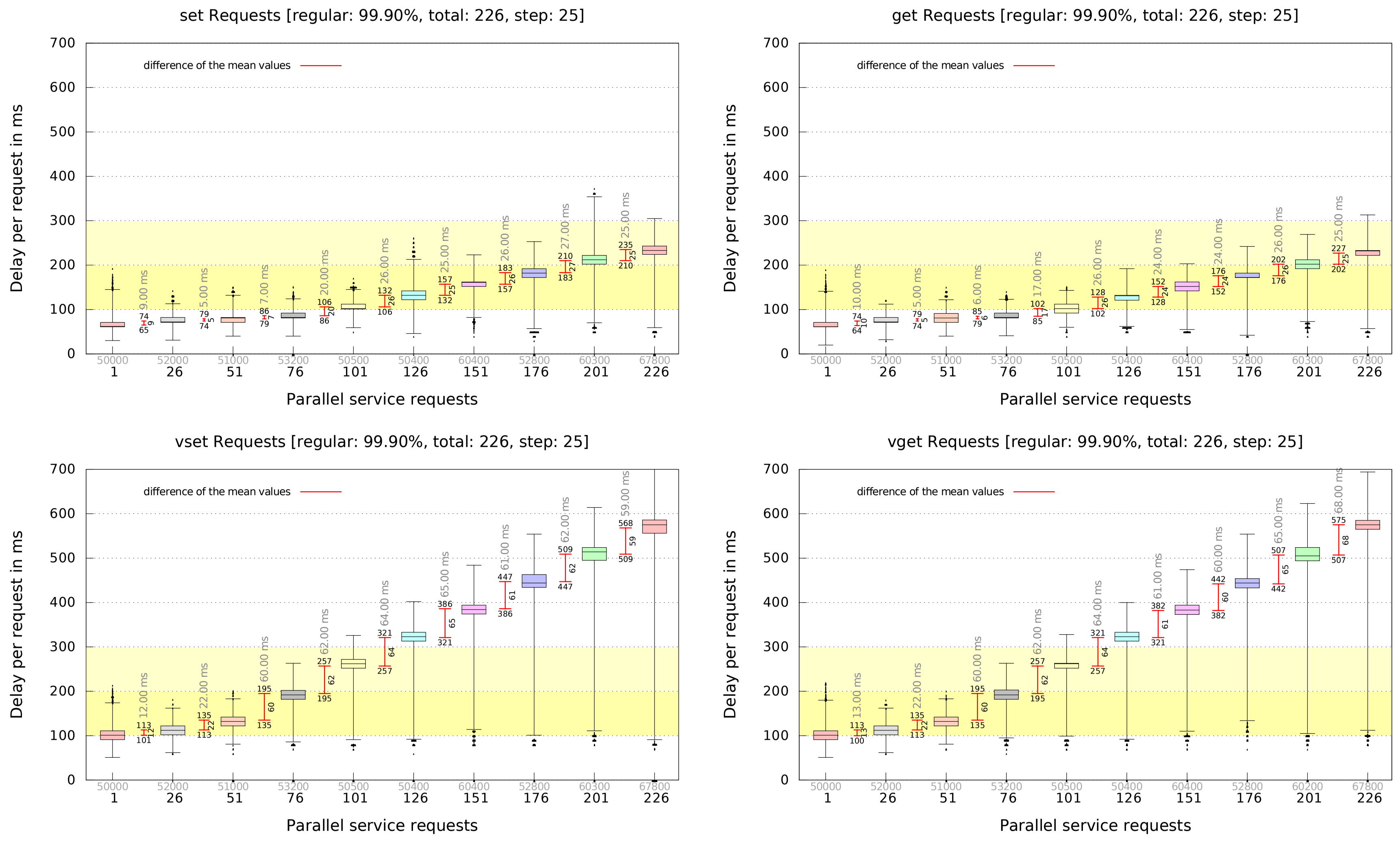


Figure 9.16: Mean delay in ms per client of 100 sequential requests sent from 1..226 randomly selected client services from 5 KA with 50 services on each KA.

Even though measured independently, the throughput is about the inverse of the delay from Fig. 9.16 as explained in the previous measurement already (Sec. 9.4.3). The added value of Fig. 9.17 is that it shows the throughput over the time of the entire 100 request burst (see Sec. 9.4.2). As can be seen, the performance remains roughly constant during the entire time of each burst showing the dependability of the VSL implementation.

In addition, the plot shows more than the previous one that the lines move closer to each other with more requests meaning that the throughput converges to a minimum which is likely to be the maximum of requests the server service can handle in a second. This becomes more clear in the next plot.

The last plots for this measurement contain the throughput in answered requests per second during the test round from the perspective of the server service. Fig. 9.18 shows the throughput for the requests on regular VSL nodes. The participating log files are again listed in brackets. The number on the server side is the multiplier for the amount of participating client service to reach the numbers in brackets on the right. This is clear as the server service only creates one log per test while each client service contributes one log to the measurement. The top entry in the legend shows for example that the test with 226 client services was run three times resulting in 3*226 = 678. That the number on the right is 677 shows that one client service log was empty or could not be collected as discussed above.

From the design of DS2OS it is clear that requests to regular nodes are answered by the VSL directly independent of the server service (Sec. 6.3). Therefore the server service reports a throughput of zero as expected.

To see how many requests are processed by the VSL KA which hosts the server service, the aggregated mean throughput of all participating client services is shown on the right. The plot shows that the KA is able to answer up to 1000 parallel requests per second as the results in Fig. 9.13 also showed. This limit is reached up from 100 parallel requests. Below 100 parallel requests the destination KA is not the limiting factor but the sending KAs and their client services.

For the requests towards the Virtual Nodes, the server service can provide throughput data as shown in the left plots in Fig. 9.19. Even though not necessary for the result, the aggregated client service throughput values are plotted here as well to show that the aggregation is showing the situation on the server in Fig. 9.18. As can be seen the limit for Virtual Node requests is about 400 answered requests per second and reached up from 50 parallel requests. The reasons why the throughput is lower for the Virtual Node requests was discussed at the delay plot (Fig. 9.16) above.

9.4.5 Scalability Measurement Conclusion

The stress-test shows that the prototypical implementation of the VSL can practically host a big amount of services while remaining responsive.

As expected, the design of self-contained tuples (Sec. 5.2.4) in combination with using a database back-end scales. With the tested prototype implementation up to 1000 requests can be served by a KA per second for regular nodes over the VSL.

The Virtual Node mechanism could probably be improved though it serves about 400 requests already. Likely limiting factors are the use of a single TCP connection and

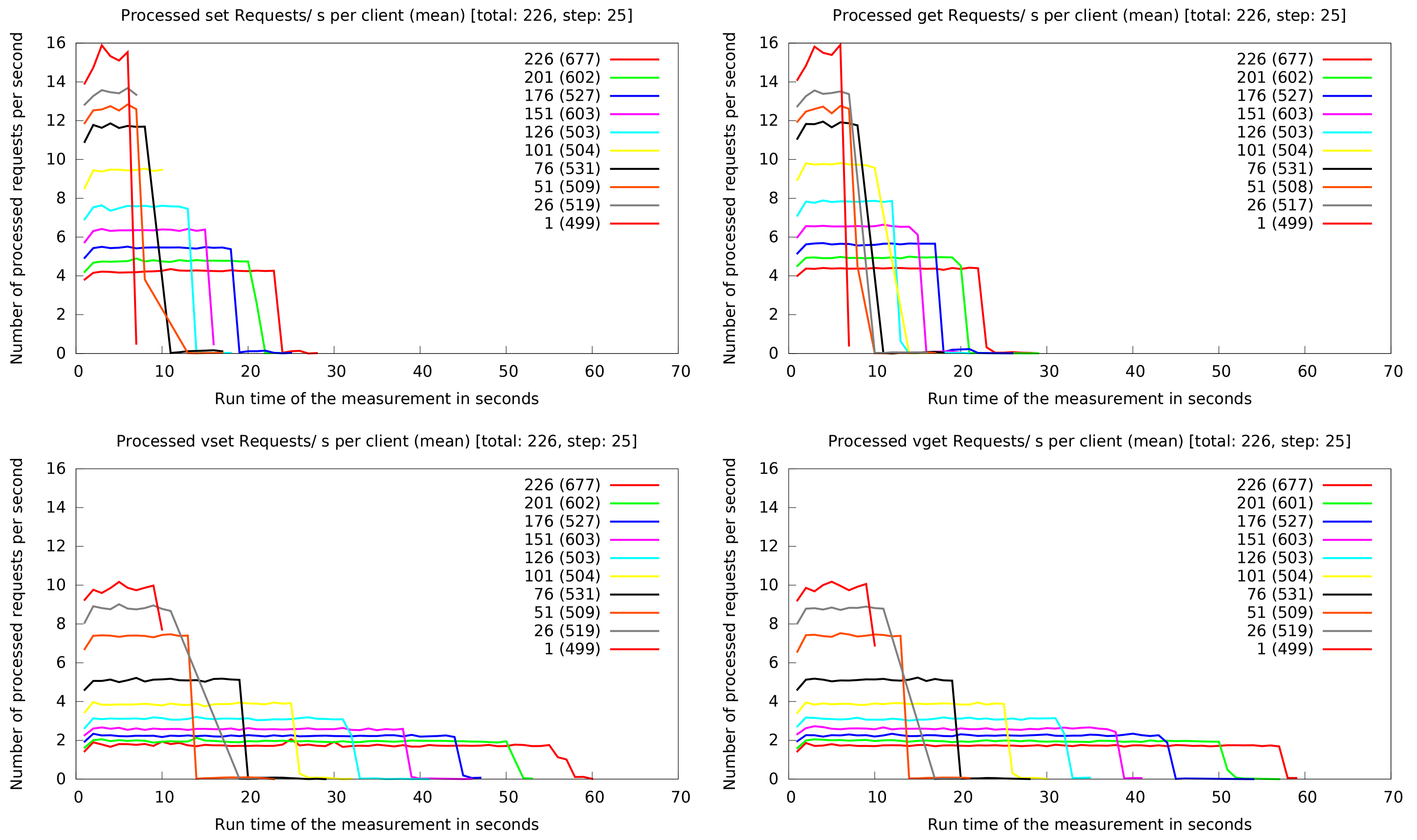


Figure 9.17: Mean throughput in requests per second per client of 100 sequential requests sent from 1..226 randomly selected client services from 5 KA with 50 services on each KA.

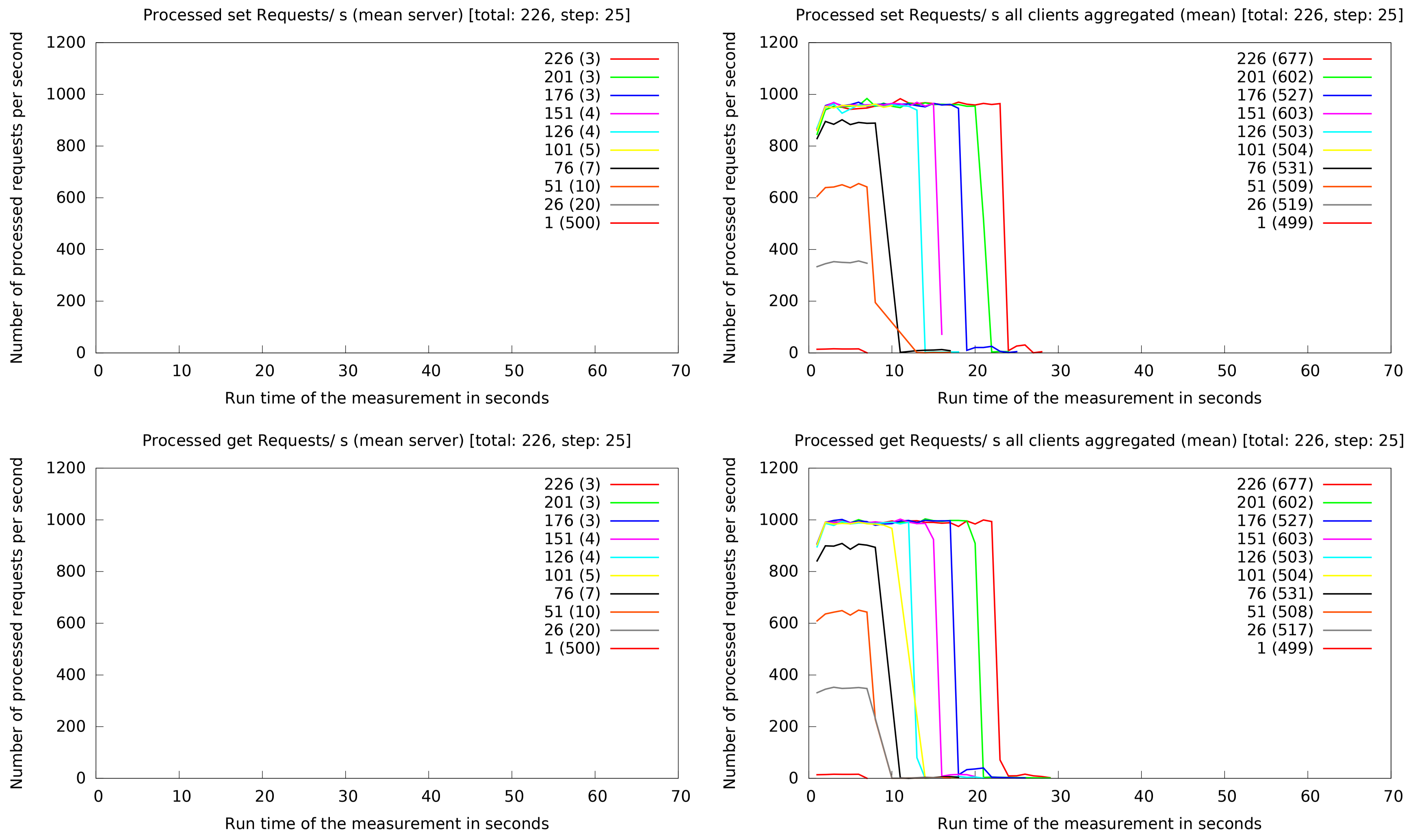


Figure 9.18: Mean server throughput in **regular VSL node access** requests per second of 100 sequential requests sent from 1..226 randomly selected client services from 5 KA with 50 services on each KA. These requests do not end up in the server service and therefore it cannot provide measurement results on the left.

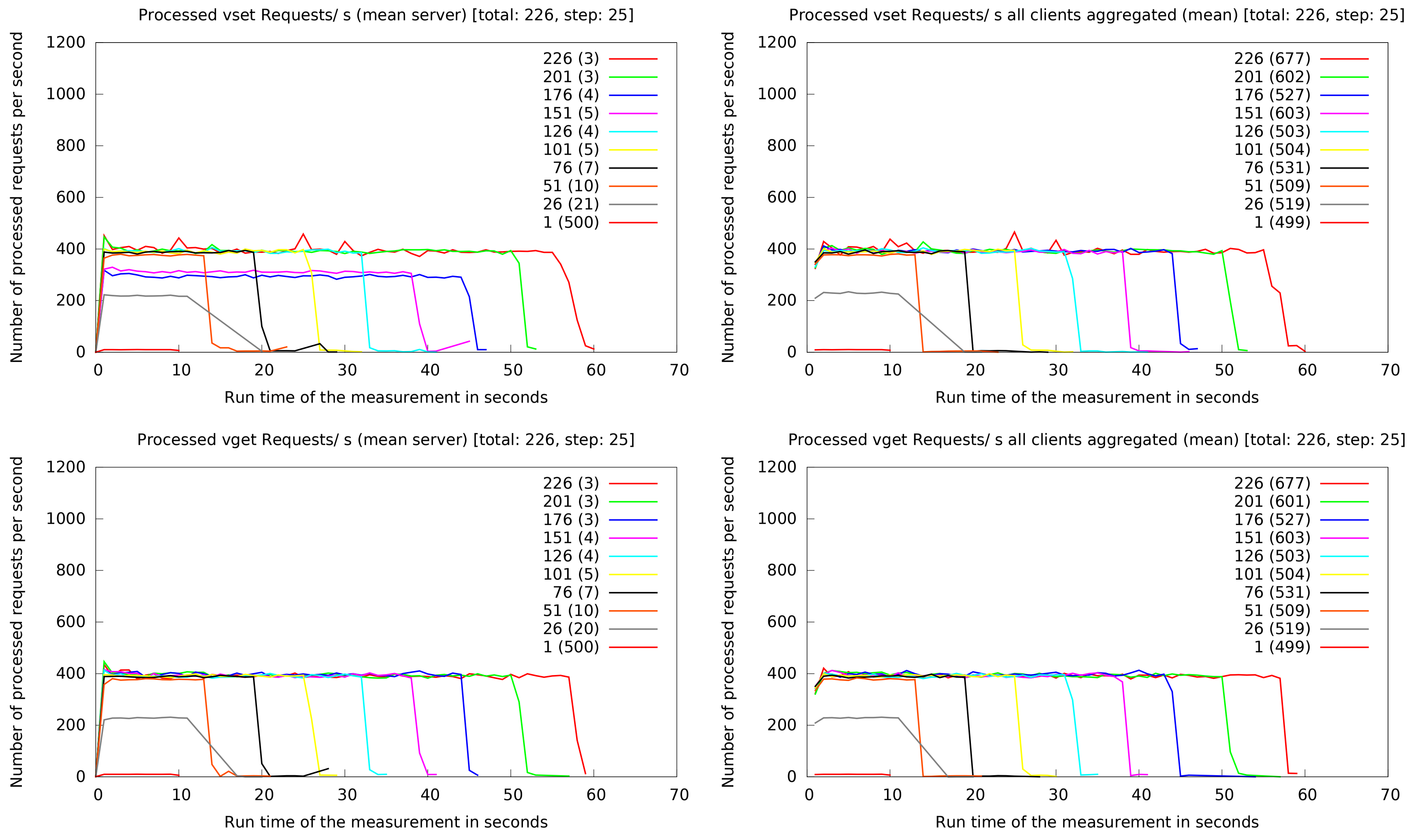


Figure 9.19: Mean server throughput in **Virtual Node access** requests per second of 100 sequential requests sent from 1..226 randomly selected client services from 5 KA with 50 services on each KA.

the single-threaded connector (Sec. 6.3.3). Further investigation and improvement of the implementation are future work.

The results from the small testbed show that under conditions that can be expected to be realistic future DS2OS Smart Spaces, the VSL allows implementing *interactive* services (Sec. 9.3.1) when multiple services work in parallel. The case of one *measurement service* corresponds to the remote KA access measurement shown in Sec. 9.3.

The results are generalizable to setups where many services on different KAs interact simultaneously as the VSL is implemented as P2P system. The chapter shows that adding additional KA peers does not result in high management overhead (Sec. 9.5), but it distributes the load. The measurement results that were obtained during the measurements can approximately be multiplied with the number of independently accessed KAs to obtain the overall throughput of a VSL instance (Sec. 9.4.4). Communications between different KAs are only limited by the network bandwidth as they happen independent from the KAs that do not participate.

The convergence measurement in Sec. 9.5 shows the amount of context that was used for the test. As described above, the design of the VSL and its implementation in the KAs make it rather independent of the amount of managed context. The only relevant factor is the amount of structure changes between two *alive pings* (Sec. 6.4.9) as shown next.

9.5 Agent Convergence

Beautifully synchronized, don't you agree?

Truman Burbank in "The Truman Show", Peter Weir 1998

As described in Sec. 6.4.9, the KAs synchronize by exchanging periodic *alive-pings*. This measurement evaluates the synchronization process between 36 independently started KAs.

Setup

The setup is the same as in the previous measurements (see Sec. 9.4.2). For this measurement 36 KAs are running on 36 computing nodes. The KAs are started one after another from PC1.1 to PC6.6 (see Sec. 9.2). All KAs are started independently and sequentially in groups of 6 KAs (Sec. 9.2). Between the launch of each group is a slightly longer delay. It is caused by the interactive setup script that requires a password to be entered for remote login on the physical machine the KA and the services should be started on.

The test setup "big setup" (Sec. 9.4.3) is reused for this test. As before, 50 services are started each of the 34 KAs. The remaining two KAs are used for the *control service* (PC1.1), and the *measurement service* (PC1.2). The services are started one per second 15s after the start of the KA. On one KA the control service is running, and on one KA the server service is running.

The alive ping interval is set to 30s. After starting 50 services per KA, each context repository has 11782 structure changes to be propagated to all other agents. The context repository on the remaining two PCs (PC1.1, PC1.2) have fewer changes as they provide fewer context. See Fig. 9.9.

Hypothesis

All hosts run the same sequence. They start one service every second after an initial waiting period of 15s. The *alive ping* period is set to 30s. It is expected that the plot will contain steps with a length of 30s until all services are started and all KAs are synchronized (Sec. 6.4.9).

The update intervals will not be synchronized between all hosts by design as the *alive ping* timer is started individually per host. This is done to distribute the *alive ping* dissemination in time and not to flood the network from all KAs at the same time.

Bigger step heights express that more updates happen in a time period. When all KAs start all their local services with the same rate, the height of each step since the first synchronization is expected to be the same for all KAs. The first synchronization will happen at 30s. If all KAs synchronize at the first possibility, their initial steps will have the same height.

After all KAs started all local client services, no changes to their context repositories should happen anymore. All KAs should be synchronized at some point after that time. All hosts run the same setup of clients. Therefore the version numbers of all remote KORs should be identical after the convergence phase. The version number for PC1.2 is likely to be low as it only hosts the server service. It is not expected to change after that service is initialized. There will be no data for PC1.1 as it is hosting the measurement and it does not have to synchronize with itself.

Result

Fig. 9.20 shows the result of this measurement from the perspective of PC1.1.

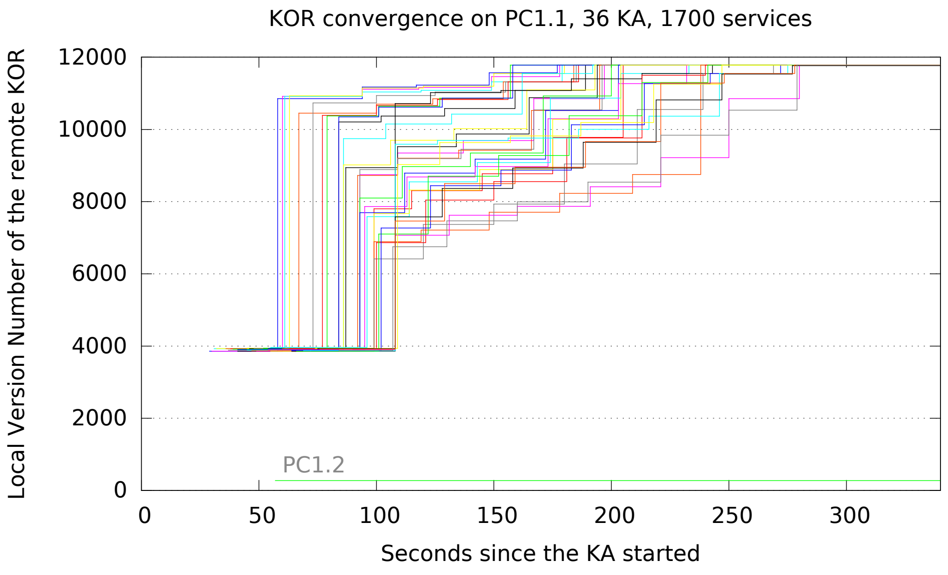


Figure 9.20: KOR synchronization of 36 KAs measured on PC1.1.

The plot shows that many KAs synchronize at the first *alive ping* exchange that happens within the time 30s and about 120s on the plot. It depends on the start time of each KA that is different as described before.

All services start at about 4000 changes. This indicates that all hosts get the same amount of services running until the first *alive ping* synchronization. Most of the hosts do similar progress until the next synchronization about 30s later which is shown in similar step heights.

Longer step lengths are explainable with different processing delays in the sender or receiver as the timer tasks are not prioritized. Listing 9.2 confirms this as it shows that the updates are not arriving and getting processed strictly regular at the receiver.

```
t old new version number
64   0 3879
102 3879  7271
123 7271  8442
153 8442  8873
183 8873  10128
214 10128 11268
242 11268 11549
272 11549 11782
```

Listing 9.2: Log for KOR updates from PC1.1. The first column shows the time after start, the second the old structure version, and the second row the new structure version.

As expected no unplanned updates are visible. All KAs are connected over a high speed and high bandwidth network (see Sec. 9.2) which makes loss improbable (Sec. 6.4.8).

The different step heights show that the distributed hosts start their measurement clients at different pace. This can be caused by the used setup script that uses different delays.

Before 300s all client services are running and the structures of all remote KAs are synchronized on PC1.1. As can be seen, all remote KAs had the same operations and run all 1700 services as they all end up at a version number 11782. This is confirmed by issuing a search on the client services as shown in listing 9.3. It confirms that 1700 instances of the VSL context model `/perfmeasure/measurementClient` are initialized.

```
system@pc1.1: / % get /search/?/perfmeasure/measurementClient
Result for type /perfmeasure/measurementClient in address pc1
1       /pc1.3/mCl1
2       /pc1.3/mCl10
3       /pc1.3/mCl11
[\ldots]
1698    /pc6.6/mCl7
1699    /pc6.6/mCl8
1700    /pc6.6/mCl9
system@pc1.1: / %
```

Listing 9.3: Console output of the search for the type /perfmeasure/measurementClient

PC1.2 is synchronized with PC1.1 at around 55sec. It does not have structure updates on the context repository after that time. Therefore is remains at the same version throughout the entire test.

9.5.1 Convergence Conclusion

The convergence measurement confirms that the *alive ping* mechanism leads to a fast synchronization of the remote KORs (Sec. 6.4.9).

The step heights in Fig. 9.20 show the amount of structure updates that have to be done by the KA on PC1.1. The measurement confirms that the processing effort depends on the amount of structure changes per *alive ping* period.

Structure changes only happen when new services are started on a KA, or when entities are added to or removed from *lists* (Sec. 5.2.10). The assumption for this implementation of KA synchronization is that structure changes are significantly less frequent than value changes.

The assumed behavior can be seen in the previous measurements in Sec. 9.3 and Sec. 9.4 where many value changes are done but no structure updates have to be disseminated. In such case the VSL implementation for synchronization (Sec. 6.4.9) results in low overhead in network capacity and computation time.

9.6 User Study

"Think simple" as my old master used to say - meaning reduce the whole of its parts into the simplest terms, getting back to first principles.

Frank Lloyd Wright, American architect, (1867 - 1959).

The overall objective of this thesis is to

<O.0> Design a programming abstraction for Smart Spaces that *structures and facilitates the development* of pervasive computing applications *at large*, and that can be *used in the real world.*

Its subsidiary objective <O.4> includes the development of software by users. The DS2OS framework provides tools for the distribution and the management of software (Ch. 7). The presented methods are expected to be suitable for a real world implementation since they are closely oriented at the design of App stores for smartphones, which proved to work (Sec. 2.3.3).

The major novelty presented in this thesis is the VSL programming abstraction. The following user study assesses the usability of the programming abstraction.

9.6.1 Setting

The study was done as part of a regular computer science course at Technische Universität München. As side result of this thesis, the presented content became part of the regular curriculum of the student education in the described course. The course material can be found online [Pah14a, Pah14b]. In Sec. A.3 two screenshots of the course material are shown.

A preliminary study was done in the summer term 2013. The results presented in this section were collected in winter term 2013/ 2014. The workspace that was used by the participants is shown in Fig. 9.1. It is explained in Sec. 9.2.

The user study is part of an experiment about Smart Space Orchestration. It consists of three parts:

- A *lecture* that gives an introduction into ubiquitous computing and Smart Space Orchestration.
- An individual *theoretical preparation* phase that introduces the background.
- A *hands-on part* that is done in a team of two.

The *lecture* gives an overview of ubiquitous computing similar to chapter 2 of this document. It introduces the basic concept of the VSL (Ch. 5), and its implementation (Ch. 6). It also introduces the proposed modularization of the program code from Sec. 8.2. Finally it introduces the Arduino platform (Sec. 3.2.3). The *lecture* took about one hour.

The *theoretical preparation* is done individually by the participants. It details content that was introduced in the lecture. The heterogeneity of Smart Devices (Sec. 3.2) is shown. The automation systems HomeOS (Sec. 4.5.5), and BOSS (Sec. 4.5.7) are briefly introduced. Then programming with the VSL programming abstraction is introduced by giving the connector interface (listing 6.1), and the abstract interfaces for the *notification callback* (listing 5.3) and the *Virtual Node callback* (listing 5.5). Finally details about building a DIY Smart Device based on the Arduino platform are given. This preparation took the participants between one to four hours.

The *hands-on part* is used for the following assessment. It is described in Sec. 9.6.4.

9.6.2 Design of the Evaluation Tools

The entire course is based on a web-based eLearning system [Pah09]. This system was extended for the user study. The extension measures the time, participants spend for doing the parts of the study. The different parts are described in Sec. 9.6.4.

The eLearning system structures content into learning blocks with inline questions the participants had to answer. The time tracking mechanism registers each time a learning block is opened and when a question is answered. The tracking happens in the background without influencing the participants.

9.6.3 Study Group

The user study was done with eight students in four teams. The participants of the study were computer science master students. One participant was female, seven participants were male.

The study is not generally representative. The number of participants is low. The background of participants is biased as they were Computer Science students. Keeping these limitations in mind, the results concerning the *time it took to get familiar with the system*, the *time it took for implementing complex tasks*, the *Lines Of Code (LOC) of Smart Space Orchestration services*, and the *individual feedback* received give an impression of the real world usability of the VSL programming abstraction.

The targeted audience of the VSL programming abstraction are enthusiastic hobby developers. Their pre-knowledge may be similar to that of the study participants concerning the Java programming background, and the Smart Space Orchestration background. The participants had background in the used Java programming language (Sec. 6.3.1) as shown in Fig. 9.24 (2). The participants did not have significant background with Smart Spaces as shown in Fig. 9.24 (1).

9.6.4 Course of the User Study

The study consists of twelve parts:

1. The first part is the "*Setup*". It shows the setup in Fig. 9.21 that is the targeted scenario of the *hands-on part.*
2. The *Web-based User Interface* part is about setting a VSL up by configuring the KAs, looking at the *agent synchronization* via *alive pings* (Sec. 6.4.9) and the *type search*, and looking at the *access rights*. In this part the *web-based User Interface (UI)* (Sec. 8.3.5) is setup.

3. The third part is "*Performance Measurements in a Complex System - the Beginning*". It consists of running the measurement that is described in Sec. 9.4.4.
4. The next part, "*Building our own smart device*", consists of building a DIY Smart Device.
5. Part five is the creation of the code that runs on the Smart Device. It is called "*Writing the Smart Device "Firmware"*".
6. The sixth part is "*Planning Your Scenario*". In this part the participants think about an *orchestration scenario* they want to implement with their Smart Device.
7. In the part "*Collecting the Logs of the First Performance Measurement*" the log files of part three are collected.
8. Part eight is about creating the context model (Sec. 5.2) to the DIY Smart Device that was created in part four. It includes distributing the model between different KAs via the *CMR service* (Sec. 6.6.2).
9. The ninth part is about "*Programming DS2OS*". It includes practical exploration of the demo service introduced in Sec. 8.2. The main part is writing the *adaptation service* that interfaces the DIY Smart Device interface that was programmed in part five.
10. Part ten is about "*Orchestrating our Smart Device with our Orchestration Logic*". Introduces ARSs (Sec. 8.3.2) for *modularization* of functionality. It contains the implementation of the pervasive computing scenario, the participants invented in part six.
11. The eleventh part is about "*Performance evaluation*". The measurement from part three is evaluated (Sec. 9.4.4). In addition to the measurement presented in Sec. 9.4.4 of this thesis, the measurement from Sec. 9.3.3 is done and evaluated.
12. Part twelve gives a summary.

As the course shows, the user study comprises many parts of this work. In addition to the described study, the user study is used to independently validate the measurement results from before (Sec. 9.3.3, Sec. 9.4.4) and to validate their reproducibility (Sec. 9.2.2).

The user study is embedded in a lab course and therefore it requires deeper understanding of the things that are done. 82 questions that have to be answered during the study ensure that the participants successfully master the parts described above.

The questions go beyond pure usage of the VSL. Therefore the measured times that it took the participants to go through a part depend not only on working with the VSL (mainly part 8-11) but also on the difficulty of the questions.

The questions are listed next. For more details see [Pah14a, Pah14b]. The number between the hyphens "-1-" at the beginning of a questions shows to which of the parts described above the question belongs.

1. -1- Give some reasons why it could make sense to distribute the described tasks over the network.

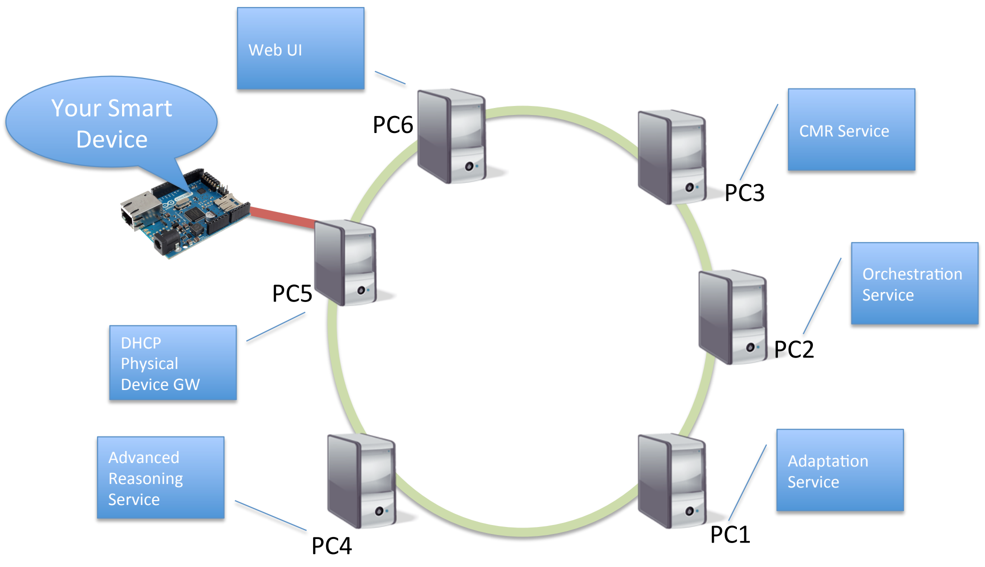


Figure 9.21: Targeted setup of the user study.

2. -2- What do the lines of the config do? To support your answer paste the relevant output lines from netstat.

3. -2- What happens when you add "/*" after agentRegister in the command?

4. -2- Paste the answer packet's TCP payload text that answers your remote get request.

5. -2- Paste an alive ping message.

6. -2- How do the agents synchronize their directories?

7. -2- What is the difference between the context subtree (local Knowledge Object Register, KOR) of the local agent and that of a remote agent?

8. -2- Concerning synchronization, what is the advantage of the observed differentiation?

9. -2- Paste the URL you called and the output. (Please use an XML beautifier [search the web for an online site doing that] to paste well indented XML into the text box.)

10. -2- What happened with the context data when you dropped the node?

11. -3- After running the script what is running on each host? Please continue the list: PC1: a KnowledgeAgent, MeasureControl, and a MeasureServer are running.

12. -3- How can you query from the VSL shell how many measurement client services are running? How many services are running?

13. -3- Paste the last five services the search reveals.

14. -3- What does location transparency mean concerning the access to the measurement control service?

15. -3- From observing the log output of the measurement control service (mcs) and your knowledge about the variables from above: describe the steps of one measurement that the mcs does.
16. -4- Why does it make sense to connect the pin 13 directly with the LED and not with the plus line of the breadboard?
17. -4- Give the formula how to calculate the resistance of the temperature sensor R2 when the resistor R1, V_in and V_out are given.
18. -4- Why is the button not changing the LED on every click?
19. -4- Please complete the following list of monitoring points (sense, input) and control points (actuate, output). Name their direction as "out" for an actuator and "in" for a sensor. Additionally attach a picture of your Smart Device showing your setup.
20. -5- Paste your configuration lines from the dhcpd.conf.
21. -5- Paste the nmap command line including the grep call and its output.
22. -5- Shortly describe how the feature you just tested could be used to implement auto discovery of your Smart Devices.
23. -5- What is the use of this code fragment in the button implementation?
24. -5- Why does it make sense to add button_state++;? (Hint: Think on asynchronous polling.)
25. -5- What is the relation between PIN_OUT_COUNT, enum pin_id_e, and int pin_out_table? Is the order of the entries relevant?
26. -5- Paste the output of your @hello call.
27. -5- Please complete the following list of monitoring points (sense, input) and control points (actuate, output). Name their direction as "out" for an actuator and "in" for a sensor. Additionally attach a picture of your Smart Device showing your setup.
28. -5- Paste the output of your @hello call.
29. -5- Paste the diff part for the enum pin_id_e.
30. -5- Paste the diff part for the int pin_out_table.
31. -5- Paste the diff part for the parserReadGet(void *ptr) function.
32. -5- Paste the diff part for the parserReadSet(void *ptr) function.
33. -5- Paste the diff part for the void sendValue(int id) function.
34. -5- Paste the diff part for the int setValue(void *ptr, int id) function.
35. -5- Upload your entire Arduino code here.
36. -5- Paste the telnet output of your successful test.
37. -5- Which are advantages of the used protocol design?

38. -5- Which are disadvantages of the used protocol design?
39. -6- Describe the orchestration workflow that you want to implement in this lab in your own words first (no code here, see next question...).
40. -6- Specify your intended orchestration workflow in pseudocode. (You will see your answer later for your implementation again.)
41. -7- Attach the archive with your log data.
42. -8- Paste the model for your smartDevice here.
43. -8- Paste the relevant protocol payload from wireshark (follow stream).
44. -8- Describe what happens from the wireshark observation.
45. -8- What is different? Why?
46. -8- Which are advantages and disadvantages of the observed behavior?
47. -8- How is the Model Repository discovered? (Hint: explore the knowledge tree of PC3 on PC4.)
48. -8- Execute the discovery commands in the DS2OS shell and paste the command and its result.
49. -9- Explain the signatures of the methods in the IVirtualNodeHandler interface. When are the methods called? What do the parameters contain?
50. -9- How is the correlation between the isLaughing node and the tickle node?
51. -9- Compare VirtualNodes and regular VSL nodes. List one advantage and one disadvantage of each.
52. -9- What does transparent access to data mean?
53. -9- In which dimensions does the VSL provide transparency? What is transparent to the developer?
54. -9- Paste your subscription wirings (registerSubscriptions()).
55. -9- Paste your Virtual Node wirings (registerVirtualNodeHandlers()).
56. -9- Attach the source code of your SmartGateway wiring java file. (aprox. 25 LOC)
57. -9- Paste the handler code you wrote for your SmartGateway and attach the source code java file. (approx. 150 LOC)
58. -10- Paste the model of your Advanced Reasoning Service and attach the source file.
59. -10- Why is it important to use meaningful and unique data types (model names)?
60. -10- Why is it enough to have a unique type for the parent node of the abstract interface (model) of a service?
61. -10- Paste the wiring code of your service.

62. -10- Paste the handler code of your service.
63. -10- Paste the source files of your Advanced Reasoning Service.
64. -10- Why does it make sense writing Advanced Reasoning Services instead of putting the whole functionality directly into an Orchestration Service?
65. -10- Paste the model for your service and attach the source file.
66. -10- Paste the wiring code of your service.
67. -10- Paste the handler code of your service.
68. -10- Paste the source files of your service.
69. -10- Paste the wiring code of your orchestration service.
70. -10- Paste the handler code of your service.
71. -10- Paste the source files of your Orchestration Service.
72. -10- What is the advantage of introducing a middleware framework like DS2OS?
73. -11- Explain all different elements you see in the boxplot in general (independent of the concrete measured values). Which element of the plot shows what (e.g. quantiles)?
74. -11- Upload all resulting plots as PDF and qualitatively describe the observed result.
75. -11- How is the coupling between what happens on the Arduino and the LatencyToDeviceMeasurer implemented?
76. -11- Match the numbers in the figure with the variables in the code. Make a list.
77. -11- Which additional delay do you expect in this third measurement?
78. -11- Does the time-diagram (figure TS1, Fig. 9.4) show the measurement from the perspective of PC2 or PC5. Explain your answer.
79. -11- From PC5: Paste the last 5 lines of your measurement from the log file and attach the log file with the measured data.
80. -11- From PC2: Paste the last 5 lines of your measurement from the log file and attach the log file with the measured data.
81. -12- Attach the combined boxplot PDF that gets created by the GNUplot script above from the data of PC5 and PC2.
82. -12- Describe what you can see on the plots and what is the difference between them.

Fig. 9.22 shows an example DIY Smart Device that was built by participants during the study. The picture illustrates the complexity of the functionality that was interfaced by the *adaptation service* the participants developed in the study.

All teams succeeded in all tasks without additional instructions. Listing 9.4 shows the context model, team2 created for the Smart Device shown in Fig. 9.22.

```
<smartDevice>
  <temperature type="/ilab/temperature" reader="*"></temperature>
  <button type="/ilab/button" reader="*"></button>
  <ledRed type="/ilab/led" reader="*">
    <desired reader="*" writer="*"></desired>
  </ledRed>
  <ledYellow type="/ilab/led" reader="*">
    <desired reader="*" writer="*"></desired>
  </ledYellow>
  <ledGreen type="/ilab/led" reader="*">
    <desired reader="*" writer="*"></desired>
  </ledGreen>
  <displayBacklight type="/ilab/led" reader="*">
    <desired reader="*" writer="*"></desired>
  </displayBacklight>
  <photo type="/ilab/lightSensor" reader="*"></photo>
  <heartbeat type="/ilab/heartbeat" reader="*">
    <desired reader="*" writer="*"></desired>
  </heartbeat>
  <timer0 type="/ilab/triggerTimer" reader="*">
    <desired writer="*"></desired>
  </timer0>
  <timer1 type="/ilab/triggerTimer" reader="*">
    <desired writer="*"></desired>
  </timer1>
  <vTimer0 type="/ilab/vTimer" reader="*" writer="*"></vTimer0>
  <vTimer1 type="/ilab/vTimer" reader="*" writer="*"></vTimer1>
</smartDevice>
```

Listing 9.4: The context model for the DIY device shown in Fig. 9.22.

The following original code snippets show the *notification wiring* (listing 9.5, Sec. 5.3.4) and the *Virtual Context wiring* (listing 9.6, Sec. 5.3.2) of two teams. The code shows that the VSL programming abstraction can be understood, and that it allows to write "elegant" code (Sec. 5.8).

```
ISubscriber desiredCallback = new ISubscriber() {
  @Override
  public void notificationCallback(String address) {
    try {
```

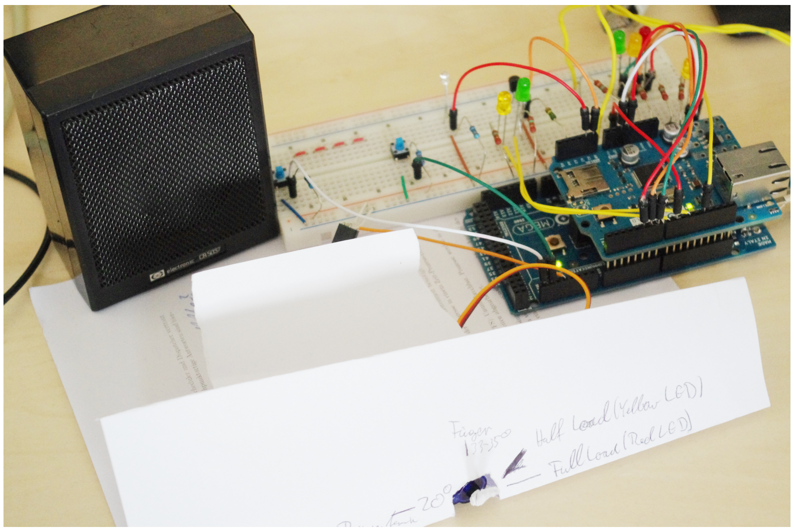


Figure 9.22: DIY Smart Device that was created by participants during the study.

```
      h.desiredCallback(address);
    } catch (VslException e) {
      LOGGER.warn("Error retrieving {}: {}", address,
          e.getLocalizedMessage());
      e.printStackTrace();
    }
  }
};

c.subscribe(myKnowledgeRoot + "/ledRed/desired", desiredCallback);
c.subscribe(myKnowledgeRoot + "/ledYellow/desired", desiredCallback)
    ;
c.subscribe(myKnowledgeRoot + "/ledGreen/desired", desiredCallback);
c.subscribe(myKnowledgeRoot + "/displayBacklight/desired",
    desiredCallback);
c.subscribe(myKnowledgeRoot + "/heartbeat/desired", desiredCallback)
    ;
c.subscribe(myKnowledgeRoot + "/timer0/desired", desiredCallback);
c.subscribe(myKnowledgeRoot + "/timer1/desired", desiredCallback);
```

Listing 9.5: The notification wiring for the context model shown in listing 9.22.

```
public static String[] ids_virt = {"speaker", "vTimer0", "vTimer1"};
public static Map<String, Integer> map = new HashMap<String, Integer
    >();

public void registerVirtualNodeHandlers() throws VslException {
    for (int i = 0; i < ids_virt.length; i++) {
        map.put(myKnowledgeRoot + "/" + ids_virt[i], i);
        c.registerVirtualNode(myKnowledgeRoot + "/" + ids_virt[i],
            new IVirtualNodeHandler() {
            @Override
            public void set(String address, String value, String
                writerID) {
                // System.out.println("Setting " + ids_virt[map.get(
                    address)]);
                h.set(ids_virt[map.get(address)], new String(value))
                    ;
            }

            @Override
            public String get(String address, String readerID) {
                if (!map.containsKey(address))
                    return ("Could not access value, invalid/unknown
                         address specified: " + address);
                return h.get(ids_virt[map.get(address)]);
            }
        });
    }
}
```

Listing 9.6: The Virtual Node wiring of another team.

9.6.5 Quantitative Evaluation

Fig. 9.23 shows the measured times that the eLearning-system logged (Sec. 9.6.2). The left y-axis shows the minutes that each part took the participants to answer on all questions and to implement all tasks. It belongs to the colored bars. The x-axis contains the parts that are introduced in Sec. 9.23. The right y-axis belongs to the lines. The lines accumulate the overall time.

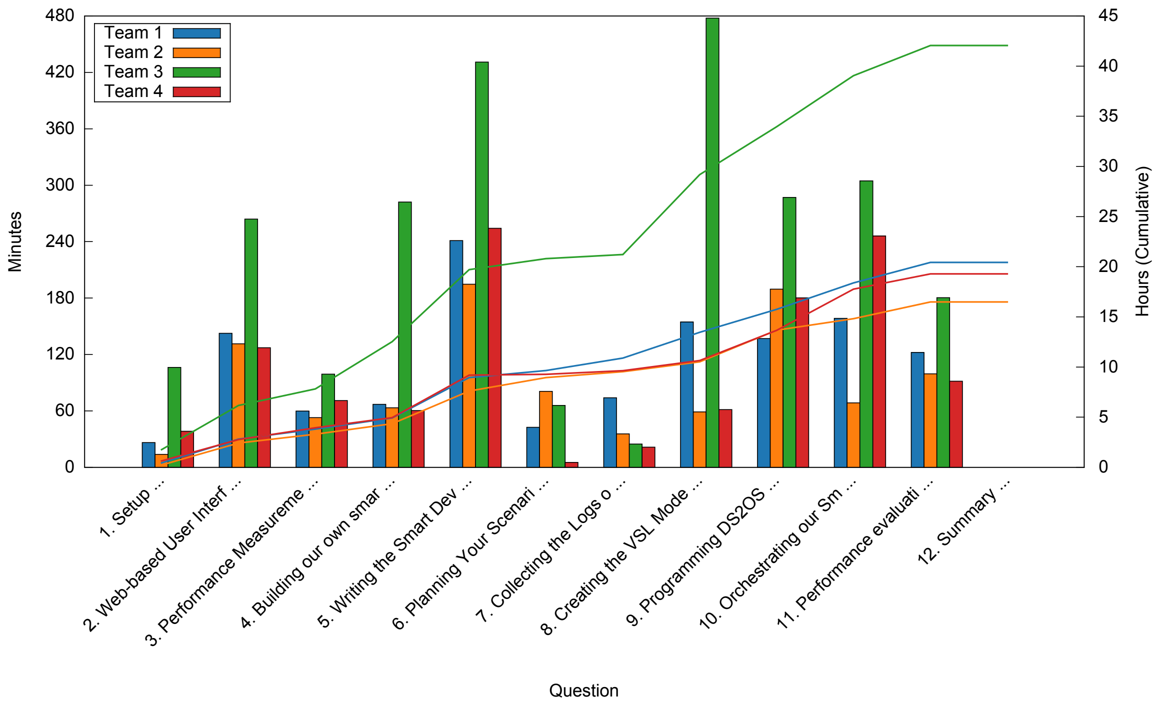


Figure 9.23: Absolute (left) and cumulative (right) time four student teams spent working on the parts given on the x-axis.

The evaluation algorithm is very basic. It assumes that participants work on one part the long they do not click on another part. The results give a rough idea how long each part (Sec. 9.6.4) took. The overall evaluated time correlates with the information the students gave on the feedback sheets. Three times "around 15h" was returned.

The teams were not supervised. They could come into the lab room when they wanted. They had between three and four weeks to complete all parts. They did not work continuously.

That the time measurements of team 3 are so much higher than the logged amount of time of the other teams can be explained by the observed behavior of team 3. The participants used the possibility not to work continuously but to make pauses during the study (e.g. for eating or going to lectures). Unluckily for the measurement, they did not log out for their pauses. This misled the described evaluation algorithm. It assumes that the team was continuously working while the team was pausing. In addition, team 3 was going through their answers several times.

The time spent in each part depends on the amount and the difficulty of the questions of the part. The questions are listed in Sec. 9.23. For creating DS2OS services it is not necessary to have the deep understanding of the underlying mechanisms that is required to pass the presented study material [Pah14a, Pah14b].

Therefore the shown times can be seen as an upper boundary that none of the teams needed to achieve the goal of creating the context model, writing the adaptation service, or the orchestration service.

At the end of the exercise, each team had their DIY Smart Device running, connected to the VSL, and created an orchestration workflow. An example workflow that a team invented is: "*We want to use the photo and temperature sensor as well as a button as* *input and some LEDs and a speaker as output. The basic flow is that the 3 traffic light LEDs show the relative current load during the performance measurement, but only if the light is turned off in the room. The speaker is used to warn the user when the temperature rises over a certain threshold, but only if the user has pushed the button before that either permits the audio feedback or completely denies it.*"

The most relevant times are:

Development of...	Time
Context model (8)	90min
Adaptation service (9)	180min
Orchestration service (10)	120min
Σ	390min = 6.5h

Table 9.1: Upper limits per task from the user study.

It can be expected that it took more time at the beginning (part 8, context model) to get used to the system.

The authors of *One.World* (Sec. 4.5.13) also made a user study [GDLM04]. It consists of programming a text and audio messaging system, and a manager that scans the system for applications and users. The authors state that these tasks are not "*overly complex*" [GDLM04].

In the user study made in this thesis, the participants had to connect a newly developed DIY Smart Device with a system they did not know before and write a relatively complex orchestration workflow (see before).

The complexity of the user tasks, and the assessed groups seem comparable between the *One.World* study and this study [GDLM04]. The time it took to implement the tasks described above in *One.World* is 256h [GDLM04]. The times it took the participants to do the described tasks (Sec. 9.6.4) was about 6.5h (table 9.1).

If the assumption holds true that the task difficulties are comparable, it shows that the VSL programming abstraction brings good usability. Even if it is not comparable, the user study shows that the VSL programming abstraction allows to implement complex tasks within short time, and with few lines of code.

9.6.6 Qualitative Evaluation

The quantitative analysis gives an impression that all participants succeeded in the tasks, and that it took not too long. It does not reflect how the users experienced working with the VSL programming abstraction. To capture this impression, a questionnaire was filled out by all participants some days after they had done the hands-on part.

The questions are:

1. How much experience with smart spaces did you have before this lab?

2. How much experience with Java did you have before this lab?
3. How difficult was it to implement the connectivity of your smart device with the VSL?
4. How difficult was it to implement the Advanced Reasoning Services?
5. How difficult was it to implement the Orchestration Service that implements your pervasive computing scenario?
6. How complex to use is the programming abstraction of DS2OS (with the hierarchically structured context nodes as representation of the real world and as abstract interfaces to services)?
7. How suitable is DS2OS for orchestrating future smart spaces?
8. How familiar are you with the concepts of DS2OS after the Lab?
9. How suitable is it to use DS2OS for beginners? (e.g. can it be used without understanding everything entirely? Are the abstractions easy-to-understand?)

The participants could always answer on a scale from 1 to 5. Fig. 9.24 shows the results. The result at the top is the best possible answer, the result on bottom is the worst. The size of the circles and the written numbers show how many participants crossed which value. The connection between the table and the explanatory text below, is represented by the identifiers in parenthesis. The identifier "(q1)" is used in the following text as link to question number 1 in the enumeration above. The corresponding evaluation in the Fig. 9.24 is marked with the number of the corresponding question "1".

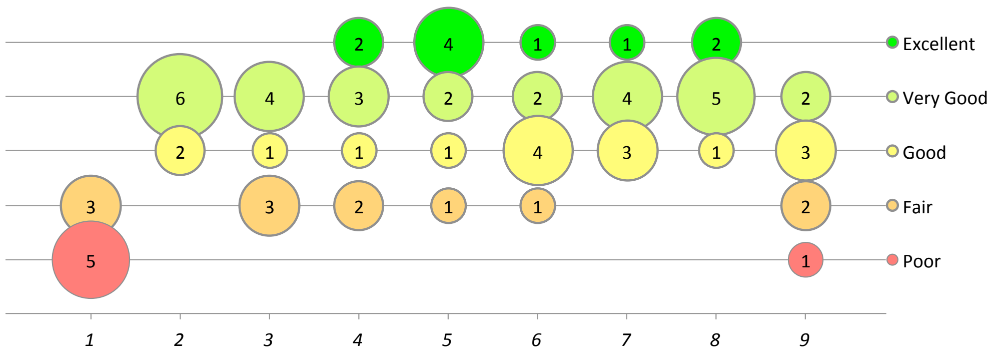


Figure 9.24: Subjective impression of the study group.

As the result shows, the participants had almost no background in Smart Space Orchestration (q1). At the same time they had very good background in Java (q2). This matches with the targeted developer group of enthusiastic DIY makers (Sec. 2.5).

The *advanced reasoning service* (called Smart Gateway in the experiment) is the most difficult service as it connects the Smart Device to the VSL. It implements bidirectionally reflection of state between the VSL and the DIY Smart Device (Sec. 8.3.1). Four participants found the task quite simple, three found it difficult, and one found it okay (q3).

The *Advanced Reasoning Services (ARSs)* derive higher-level context (Sec. 8.3.2). Knowing the system better, the perceived difficulty shifted towards very simple (q4) which was crossed by two participants.

The *orchestration service* implements the pervasive computing scenario. Having the helper services developed, most participants answered to question 5 that it was *very easy* to implement the pervasive computing scenario (Sec. 9.6.5 for an example scenario that was implemented by a team).

Explicitly asked about the usability of the VSL programming abstraction in question 6, most participants answered *good* with a tendency towards better. For this result it has to be taken into account that it was the first contact of the participants with the novel programming model. A hand-written comment here was "*When you got into it, it is quite nice and "easy", but the first "dive" into the system is quite hard.*".

In addition it has to be considered that the participants created an *adaptation service*, *ARSs*, and an *orchestration service*, and worked with the *UI service*. With its diversity, orchestration of Smart Spaces is complex. Especially when never done before (q1). This is reflected in the next answer.

Asked for the *suitability* of the VSL programming abstraction as tool to create software for future Smart Spaces, most participants answered *very good* (q7). A hand-written comment here was "*From the lab it seems quite suitable because it provides a nice layer of abstraction and a very easy to understand api that makes it well suited to create advanced services.*".

Question 8 asked for the *familiarity* with the concepts of the VSL programming abstraction after having gone through the exercise. Most participants feel *very familiar* with the system after this short time. The answer shows that the chosen approach is apparently experienced as *intuitive*.

The last question asked for the *suitability for beginners* (q9). Though the previous answers show that all participants got along well with the VSL programming abstraction, they still perceive it difficult to use the system for beginners. This is probably interrelated with the short discussion above, that Smart Space Orchestration is a new concept. Users and developers are not used to it yet as it is not reality in 2014 (Sec. 2.4). A suitable programming abstraction for Smart Spaces that *structures and facilitates the development* of pervasive computing applications *at large*, and that can be *used in the real world* (<O.0>) is missing (Sec. 2.5).

9.6.7 Conclusion

The user study shows that this thesis' goal of designing a simple-to-use and powerful programming abstraction for Smart Space Orchestration was accomplished for the participants of the study.

The quantitative analysis shows that within 15h a deeper understanding of Smart Space Orchestration can be created with the VSL programming abstraction. The program code of the participants shows that the VSL, and combination with the modularization that is proposed by the service template (Sec. 8.2) structure the programming of Smart Spaces. The code shows that the VSL enables the creation of complex functionality with little code.

The qualitative analysis shows that the VSL programming abstraction is simple-to-use and has a good learning curve.

The measurements that were presented in Sec. 9.3 and Sec. 9.4 are partly contained in the user study. The participants proved the reproducibility of the measurements (Sec. 9.2.2).

As part of the regular curriculum at Technische Universität München the instruction material that was used for the study teaches computer science students about Smart Space Orchestration.

9.7 Chapter Conclusion

This chapter evaluates the VSL from different dimensions.

The delay measurements in Sec. 9.3 show that the VSL programming abstraction allows fast context access. This confirms that the self-contained tuples can be processed efficiently as expected (Sec. 3.9, <R.40>). *<R.40>*

Using the VSL, services can be coupled to implement an interactive user experience (Sec. 9.3.1). Sec. 8.3.2 discusses the relevance of modularization for facilitating development (<R.26>, <R.1>). *<R.26>*

<R.1> The VSL implementation is not optimized for scalability but the VSL design is. The test with up to 1600 parallel requests from 34 distributed KAs showes that the implementation is robust (<R.30>). *<R.30>* The amount of participating entities shows that the self-management scales (<R.50>). *<R.50>*

As discussed in Sec. 9.4.4, the P2P architecture is chosen to distribute the load. The second measurement showes that for up to 101 parallel requests per node a coupling between two services over either regular nodes or Virtual Nodes is possible within the 300ms boundary for *very interactive* user perception (Sec. 9.3.1).

As this limit relates to a single node and the KA peers are independent, this amount can be multiplied by the amount of independently interacting KAs (Sec. 9.4.5). This makes the VSL architecture scale (<R.50>).

The evaluation of the self-management overhead in Sec. 9.5 confirms fast convergence and low load on the network and the participating KAs via the chosen structure synchronization approach (Sec. 6.4.9, <R.3>). *<R.3>*

The user study (Sec. 9.6) shows that the VSL programing abstraction enables integrating Smart Devices, and implementing diverse pervasive computing scenarios within short time (<20h, Sec. 9.6.5). The basis of this fast implementation is the context-awareness support of the VSL (<R.5>). *<R.5>*

The participants of the user study implemented a complex scenario. They created a DIY Smart Device first, and integrated it into the VSL. This shows that in 2014 with the available hardware (e.g. Arduino) and the VSL on the software side, a maker culture for pervasive computing can technically emerge (Sec. 2.5, Sec. 10.4, <R.4>, *<R.4>* <R.48>). *<R.48>*

The subjective impression of the participants of the user study confirms the simplicity-to-use of the VSL programming abstraction (Sec. 9.6.6, <R.1>). *<R.1>* Especially the simplicity in the use of the central context abstraction is confirmed (<R.29>). *<R.29>*

The code that was produced during the user study shows how the VSL programming abstraction structures and facilitates the implementation of complex scenarios. Better code fosters service dependability (<R.30>), and helps enhancing the security of Smart Space services (<R.49>). *<R.30>* *<R.49>*

That the components of the VSL were not modified for any of the tests shows the real world usability of the approach (<O.4>).

Part IV

Conclusion

10. Conclusions & Future Directions

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

Edsger Wybe Dijkstra, Dutch computer scientist, (1930-2002)

From the first moment I came in contact with programming in the 1990s, the act of developing programs had something magic for me. Programming comprises abstracting a problem, thinking about a solution, logically formalizing the solution into a computer program, and refining it until the problem is solved[48]. Starting a program makes the logic alive – thoughts become actions in a magic way.

As described in Sec. 2.5, in the 1990s the physical shell of a computer was limiting the sphere of actions. I remember well, buying my first soundcard, a *soundblaster* from the company *Creative Labs* with an ISA bus. This device is fascinating as it extends the range of a Personal Computer (PC). Before installing it, program actions were limited by the screen. With audio the effects of programs were directly coming into my room – out of the shell into my world. Having audio output was a significant extension of the range of computer programs into the world of their users. The interaction between the world inside the computer and the outside user was significantly enhanced for me.

Over the years, I followed the development of computer games. Having computer games in 2014 that provide high quality video and audio giving an impression very close to real world footage amazes me. In 2014, head-mounted displays such as the *Oculus Rift*[49] are about to extend the impression of reality that can be provided through a computer. The described form of games pulls the human user into the virtual world that the game creates (*Virtual Reality (VR)*).

With new devices such as Google glass[50], the *real world* can be augmented with computing (*Augmented Reality (AR)*). AR fascinates me as it extends the *reality* with computing. While games pull me into their *VR*, AR pushes computing into my reality.

The described development goes from

[48]Often a problem becomes more clear in the process as logic abstraction provides better understanding, and different access to problems.

[49]http://en.wikipedia.org/w/index.php?title=Oculus_Rift&oldid=597991496

[50]http://en.wikipedia.org/w/index.php?title=Google_Glass&oldid=598804606

- computer programs that *control* a *limited virtual world* that exists inside the PC screen to
- computer programs that control (theoretically) *unlimited virtual worlds* that humans can perceive over many senses to
- computer programs that *augment* the *real world.*

Pervasive computing continues this evolution. The presented Virtual State Layer (VSL) programming abstraction (Ch. 5) and the foundation services (Ch. 8) enable computer programs to *control* the *real world.* The interaction with the *real world* via Smart Device sensors and actuators (Ch. 2, Ch. 3) enables Smart Space Orchestration. The limiting shell of today's computers is extended. It covers the entire world now. The operating range of VSL software is only limited by the available distributed Smart Devices (Sec. 10.4).

Starting a program makes the logic alive – thoughts become actions in a magic way.

This evolutionary step opens up numerous possibilities – and hazards.

I'm afraid. I'm afraid, Dave. Dave, my mind is going. I can feel it. I can feel it. My mind is going. There is no question about it. I can feel it. I can feel it. I can feel it. I'm a...fraid. Good afternoon, gentlemen. I am a HAL 9000 computer. I became operational at the H.A.L. plant in Urbana, Illinois on the 12th of January 1992. My instructor was Mr. Langley, and he taught me to sing a song. If you'd like to hear it I can sing it for you.

Quote from "2001 A Space Odyssey", Stanley Kubrick, (1928-1999), American film director, screenwriter, producer, cinematographer, and editor.

10.1 Assessment

When we understand knowledge-based systems, it will be as before- except our finger-tips will have been singed.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

This thesis started with the challenge to provide a programming abstraction that enables the shift from "*programming environments to programming environments*" [Abo12]. The challenge was formalized into objective <O.0> that was divided into four subordinate objectives. To reach the objectives, 50 requirements were identified.

The fulfillment of the *functional requirements* was shown in the chapters 5ff with the markers in the text (<R.1>). Te fulfillment of the requirements leads to a fulfillment of the objectives as shown in Fig. 3.15. *<R.1>*

The subjective requirement *usability* for the VSL programming abstraction was considered by taking results from psychology into account (Sec. 3.5). The intuitive use was demonstrated in the chapters 7ff, and assessed via the user study in Sec. 9.6.

The Distributed Smart Space Orchestration System (DS2OS) framework implements elements from the App economy for smartphones and provides identical usability (Ch. 7). As the App economy is apparently suitable for *real world use* from a usability perspective, DS2OS is too. In particular the simple-to-use service deployment, and security features were discussed (Ch. 7). The scalability of the VSL was assessed in Ch. 9.

Ch. 3 showed the feasibility of Smart Space Orchestration with the VSL. The crowd-sourced standardization process (Sec. 7.4.2) is fundamental for real world deployment of the solution. It provides interface portability of services. Interface portability enables the execution of one service implementation in different Smart Space instances without change (Sec. 5.6.3).

This solves objective <O.4>, to

<O.4> Design a programming abstraction that can be *used by regular users* for implementing pervasive computing in the(ir) real world, and not only by researchers in laboratories. This includes empowering users to develop their own programs that implement their desired pervasive computing scenarios (Sec. 2.5.2).

The diversity of pervasive computing use cases is supported by the μ-middleware concept (Sec. 6.2). Its seamless extensibility enables transparent coupling of distributed services over descriptive interfaces (Sec. 5). The support of divers communication modes (Sec. 5.4.3) enables the implementation of diverse scenarios. In particular it allows to provide the functionality of the assessed pervasive computing middleware solutions from Ch. 4 via regular VSL services. Ch. 7, and Ch. 8 showed how the VSL programming abstraction facilitates the implementation of such services.

This solves objective <O.3>, to

<O.3> Support *diverse* pervasive computing scenarios and not only few use cases.

The VSL programming abstraction does not differentiate between service classes (Ch. 8). All services use the identical mechanisms and offer the same interfaces. This allows composing all services in a Service Oriented Architecture (SOA). The resulting modularization of functionality facilitates (Sec. 3.5) the software development. Existing functionality can be reused to mash up new functionality (Sec. 8.4).

The VSL allows providing *coupled* interfaces for the same functionality on different levels of abstraction (Sec. 8.3.2). This facilitates the development of programs as program logic can be transparently coupled on different levels of abstraction (Sec. 3.5).

The separation of service logic from service state structures VSL services. It allows sharing context in a controlled way over the VSL middleware independent of services (generative communication).

The use of a descriptive interface facilitates the development of services as descriptive knowledge is more intuitive for humans than procedural knowledge (Sec. 3.5). Using context models as abstract service interfaces provides the inheritance and validation mechanisms of the VSL meta model (Sec. 5.2). This supports developers in creating abstract interfaces (e.g. inheritance makes the interface in listing 5.2.11 short). The (enforced) sharing of context models (abstract service interfaces) over the Context Model Repository (CMR) enables reuse.

This solves the objectives <O.2>, and <O.1>, to

<O.1> Structure and facilitate the development of so-called *adaptation services.*

<O.2> Structure and facilitate the development of *orchestration services.*

Having all the subsidiary objectives solved, the overall goal of the thesis is solved, to

<O.0> Design a programming abstraction for Smart Spaces that *structures and facilitates the development* of pervasive computing applications *at large*, and that can be *used in the real world.*

The differences to the state of the art were addressed in Sec. 4.5. They are summarized in table 4.1, and in the difference of the existing combined reference architecture in Fig. 4.3 to the corresponding DS2OS architecture in Fig. 7.6.

10.2 Conclusion

This thesis introduces a middleware paradigm that is called μ-middleware. It enables transparent extension of a middleware core with functionality at run time. Characteristic for the VSL is the use of a descriptive data structure as functional interface for services that can be accessed via fixed methods. This contrasts the typically used method-signature based service interfaces.

The introduced Virtual Context concept enables the use of static interfaces for dynamically generated content. It enables a direct coupling of services over a descriptive data structure. As shown in Sec. 5.4.2 with the Virtual Subtrees, the resulting interface can be used like an interface of a functional programming language. A difference is that the VSL can additionally be used like a distributed database that stores and provides context. This duality combines the dynamics of a SOA with the context-awareness support of context provisioning middleware.

A side-effect of using the VSL for service coupling is the enabling of security-by-design. All communication is controlled by the VSL, either by routing requests to service callbacks (Sec. 5.4.2), or by entirely serving the requests (Sec. 5.3.2). Security is important for the implementation of pervasive computing. Security and safety critical areas in the immediate human surrounding are inevitably touched by the new technology. The need for security and safety is expressed by the lower quote in the introductory text to this chapter, and in Sec. 2.6.2.

The introduction of the CMR to share and converge context models (=abstract service interfaces) solves a fundamental problem of SOAs and Smart Spaces: it fosters *portability*. Introducing crowdsourced methods for implementing a self-convergence of the resulting ontology provides the necessary scaling standardization.

The introduced hybrid VSL meta model provides the necessary simplicity for enabling crowdsourced development of context models. The resulting self-containment of context tuples provides the necessary processing efficiency for larger installations, and devices with limited resources[51].

The DS2OS framework could directly be used to create an App economy for Smart Spaces (Sec. 10.3). Ch. 7 showed how the complexity of Smart Space Orchestration can be lowered to the complexity that is provided by the App economy in 2014 (Sec. 3.10). At the same time it showed, how users can be effectively empowered to control the access to functionality in their Smart Spaces.

Similar to the App economy, DS2OS enables user-based software development. DS2OS provides an infrastructure for sharing user-developed services. The previously discussed portability enables running the same service in all VSL-operated Smart Spaces. The functionality of a service only depends on the locally available VSL context models that define the locally available functionality (Sec. 7.5).

Ch. 8 demonstrated the feasibility of the approach for supporting existing or custom built devices (Sec. 3.2.3). It showed how future User Interfaces (UIs) could look like. The connectivity of distributed Smart Spaces could become interesting in the future for sharing hardware (e.g. wind sensor), and experiences (e.g. heating curve)

[51]The instantiation of a context model can be offloaded from a device. It is transparent for a device if it receives a fully specified context model, or a context model that requires recursive resolution of dependencies.

[PNKC13]. The mash up of functionality (Sec. 8.4) showed how users can implement complex pervasive computing scenarios only by configuring Event-Condition-Action (ECA) rules (e.g. via a graphical interface). This empowers users to benefit from the new functionality without programming their own services.

In 2014, there is an emerging maker culture for Smart Devices (Sec. 2.5.1, Sec. 3.2.3). With its simple-to-use powerful programming interface, the presented solution fosters the emergence of a complementing *maker culture for software* for Smart Spaces. Such development would result in a real world implementation of pervasive computing, making the last piece of ubiquitous computing become reality (Ch. 2).

10.3 Future Directions

Dealing with failure is easy: Work hard to improve. Success is also easy to handle: You've solved the wrong problem. Work hard to improve.

Epigrams on programming [Per82], Alan Perlis, American computer scientist, (1922 - 1990), 1st Turing award winner 1966.

Different future directions were already mentioned in the text. The most important next step is the development and the release of the Smart Space Store (S2Store), and the Smart Space Service management (S2S) service management infrastructure. Both activities have started already. The release allows real users to use the technology and to contribute. Evaluating the results is expected to become interesting research. In particular the emergence of the crowdsourced ontology is interesting as it does not come out of the research community like the current approaches but out of the real world.

More concrete tasks that were mentioned in the text are:

- Automated generation of *service stubs* for further simplification of the VSL service development (Sec. 5.4.1).
- Evaluation of the chances to implement formal service validation (Sec. 5.5.6).
- Evaluation of context caching mechanisms (Sec. 6.4.5). With caching and the locator-id-split, the VSL implements not only a delay-tolerant network but also a content-centric network [JST+09]. Results from this research area might be directly applicable for the caching. In the other direction, the VSL may also be an interesting implementation of a content-centric network/ information-centric network.
- Evaluation of the automated connector generation based on an Extensible Markup Language (XML)-Remote Procedure Call (RPC) Interface Definition Language (IDL) Sec. 6.3.2.
- Implementation of more optimization strategies in the Site Local Service Manager (SLSM) (Sec 7.2.4).
- Evaluation of the use of applying more migration techniques in the SLSM that are known from cloud computing (Sec. 4.3).
- Further development of the concept for automated integration of new Smart Devices (Sec. 8.3.1).
- Further development of the concept for automated generation of adaptation services for unsupported Smart Devices (Sec. 8.3.1).
- Implementation of a multi-threaded connector that uses multiple connections to its Knowledge Agent (KA) to make the Virtual Node handling scale better (Sec. 9.4.3).

10.4 Towards Smart Spaces

Neo, sooner or later you're going to implement just as I did that there's a difference between knowing the path and walking the path.

Morpheus, The Matrix, 1999

At the beginning of 2014, Smart Spaces are not part of everyday life. With the solutions that this thesis provides, this may change. This section gives a visionary outlook how such change may happen. Fig. 10.1 illustrates the envisioned development.

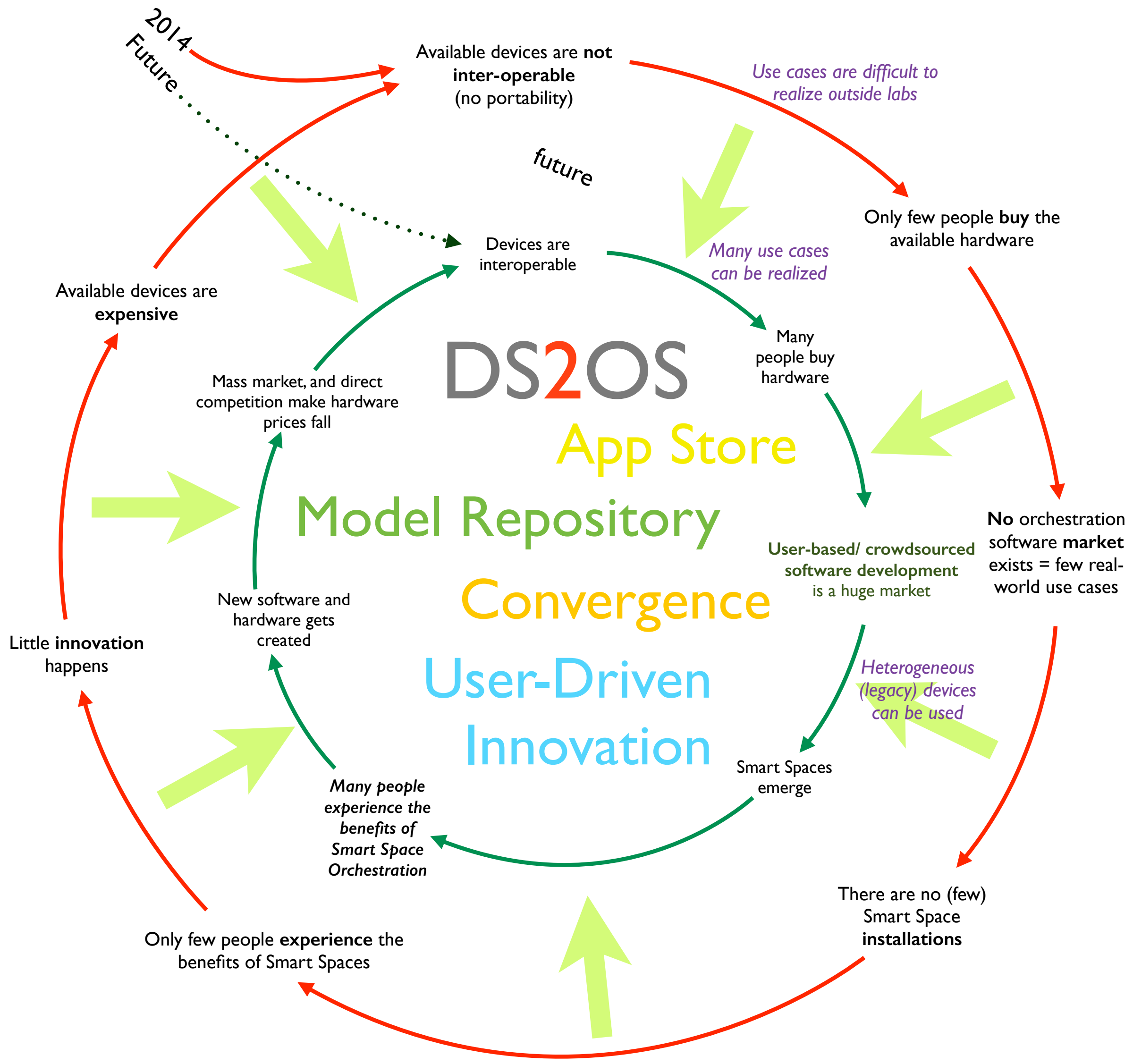

2014
Future
Available devices are not inter-operable (no portability)
Use cases are difficult to realize outside labs
Only few people buy the available hardware
No orchestration software market exists = few real-world use cases
There are no (few) Smart Space installations
Only few people experience the benefits of Smart Spaces
Little innovation happens
Available devices are expensive
future
Devices are interoperable
Many use cases can be realized
Many people buy hardware
User-based/ crowdsourced software development is a huge market
Heterogeneous (legacy) devices can be used
Smart Spaces emerge
Many people experience the benefits of Smart Space Orchestration
New software and hardware gets created
Mass market, and direct competition make hardware prices fall
DS2OS
App Store
Model Repository
Convergence
User-Driven
Innovation

Figure 10.1: The situation of Smart Spaces in 2014 in the outer loop, and the possible development with the presented DS2OS ecosystem in the inner loop.

In 2030, DS2OS drives many Smart Spaces installations. Following, a review from the year 2030 back to 2014 is given form the perspective of users, developers, and "the market".

10.4.1 User Perspective 2030

The growing amount of Smart Spaces led to mass production of Smart Devices. Prices dropped significantly. Almost all available hardware devices are Smart Devices, allowing Smart Space Orchestration.

Millions of Smart Space services are available over the S2Store (Sec. 7.2.6). The crowd-sourced developer community provides drivers for new devices within days. Users buy new Smart Devices,

- plug them to their VSL network,
- the *device discovery service* (Sec. 8.3.1) detects them,
- if a driver is available, it is automatically installed using the SLSM (Sec. 7.2.4),

and the device is usable.

New pervasive computing scenarios can simply be installed by finding them on the S2Store, and clicking on `install` in the SLSM interface (Fig. 7.5). The prices of Smart Space Services are similar to those of smartphone Apps in 2014 around 1$ per service.

Over the time, meaningful names and descriptions for the *access groups* in the S2Store emerged (Sec. 7.3.3). This facilitates setting the access rights. As many VSL services share the same access groups, the access rights configuration is typically done with one click as the SLSM suggests the rights based on previous selections.

Via Smart Space Orchestration, the behavior of a Smart Space can be changed by reconfiguring existing, or installing new Smart Space Services. In combination with new devices, such as active wallpapers that can display items like monitors did in 2014, the entire type of use of a Smart Space can be changed via software. Such change is simple and fast. Therefore physical environments are changed frequently. New usage patterns of physical spaces emerged. Literally the reconfiguration of a physical environment shifted from the hand to the mind.

10.4.2 Developer Perspective 2030

The market volume for Smart Devices and Smart Space Services is huge. Programming the reality is not science fiction anymore.

The propagation of 3D printers, and the availability of embedded system kits allow users to invent and create their own Smart Devices at home. More and cheaper Smart Devices than were produced in total in 2014 are getting produced by users in their homes. The sharing construction schemes for such Do-It-Yourself (DIY) Smart Devices is a huge market.

Thousands of new Smart Space Services are uploaded to the S2Stores daily. A crowd-sourced developer scene emerged similar to the App development scene for smartphone Apps in 2014.

The simple-to-use Application Programming Interface (API), the convergence mechanisms, and the automated feedback, that acts as bug reporting mechanism, made the development of high-quality Smart Space Service so simple that even programming beginners offer DS2OS Smart Space Services.

Adaptation services for new Smart Devices are typically in the S2Store within hours. Most often they are available before the devices are sold. Vendors release their interface context models, and corresponding emulators early. This fosters the development of Smart Space Services that use their new Smart Devices. In turn they can sell more devices.

The large scale of the market made Smart Space Service development a profitable business. At the same time the enabling of crowdsourced development led to a diversity of use cases that could not be imagined by the research community in 2014.

10.4.3 Market-Economic Perspective 2030

Smart Spaces are a huge business with hardware and software vendors as main actors. Fig. 10.1 shows how the market situation changed since 2014. The outer development is explained first, starting with the 2014 arrow on the top left. Then the inner circle is explained.

Without use cases, only few people invested in Smart Devices in 2014. The deployed Smart Devices were often incompatible, and only used to implement silo functionality. The silos complicated the implementation of pervasive computing scenarios in 2014.

The missing of a common abstract interface, an Operating System (OS) for Smart Spaces, made the reuse of Smart Space Services almost impossible (no portability). Services could only be used in the Smart Spaces they had especially been developed for. As a result, only few Smart Spaces existed. Only few people could experience the benefits of Smart Space Software Orchestration. The situation could be compared to the beginning of the smartphone era around 2008.

The low market volume kept vendors from investing in the development of Smart Devices. As is clear now, real innovation only started with enabling users to design, create, and share pervasive computing scenarios on their own. The maker scene for hardware and software (Sec. 10.4.2) became a huge and profitable business.

As only few Smart Devices existed in 2014, they were expensive. The high price prevented even early adapters from buying them.

Then DS2OS was released. What started with a few enthusiasts back in 2014 is installed almost everywhere today. Via Smart Space Orchestration, the S2Store, and crowdsourced development, almost all devices inside a Smart Space can be controlled by DS2OS services. While this is taken for granted today, the work of skilled users that developed adaptation services for their own Smart Device and shared the services over theS2Store was very important at the beginning. It enabled non experts that had the same devices at home to experience Smart Space Orchestration for the first time.

Millions of scenarios can be installed from the S2Store. As the possibilities of orchestrating the environment are huge, many people started developing Smart Space Services. The S2Store allows users to distribute their Smart Space Services. It manages the entire life-cycle of a service. In return, a share on the sales revenue of each service is kept by the S2Store operator – like App stores did in 2014.

The crowdsourced convergence mechanisms led to a self-standardization by the market (Sec. 7.4). It enabled vendor interoperability in a scalable way (portability).

The implemented and available use cases attract many consumers. Even people that do not have a Smart Space at home experience the technology in public spaces. Such spaces can be controlled by the people that are present in the space over public interfaces.

Companies and users produce Smart Devices and sell them via the Internet. Many consumers produce their Smart Devices via their 3D printers at home. The mass production led to a significant price drop, making Smart Devices affordable to everyone.

As people experience the benefits of Smart Spaces everywhere, many people buy Smart Devices resulting in a high penetration of physical spaces. The Smart Device market is huge. The third computing generation is reality for a long time.

A. Appendix

A.1 Requirements List

Following, a short list of the 50 requirements that were identified in the analysis section (Ch. 3) is given.

<R.1> *Simplicity-to-use* of the programming abstraction, and of the services using it.

<R.2> *Multi-user support.*

<R.3> *Self-management.*

<R.4> *User participation* in the creation of software.

<R.5> *Context-awareness support.*

<R.6> *Role-independent unified interfaces*, not distinguish between different service roles.

<R.7> *Portability of the implementation of the programming abstraction.*

<R.8> *Logical portability of services.*

<R.9> *Information loss* prevention.

<R.10> *Standardization* to overcome heterogeneity.

<R.11> *Distribution support.*

- <R.12> *Access transparency.*
- <R.13> *Location transparency.*
- <R.14> *Migration transparency.*
- <R.15> *Relocation transparency.*
- <R.16> *Concurrency transparency.*
- <R.17> *Failure transparency.*
- <R.18> *Persistence transparency.*

Coordination Modes

- <R.19> *Direct communication support.*
- <R.20> *Mailbox communication support.*
- <R.21> *Publish-subscribe communication support.*
- <R.22> *Generative communication support.*

<R.23> *Dynamic extensibility with device drivers and support functionality.*

<R.24> *Descriptive knowledge representation.*

<R.25> *Interaction on different levels of abstraction.*

<R.26> *Modularization support.*

<R.27> The *support of multiple programming languages.*

<R.28> *High expressiveness of the context representation.*

<R.29> *Simple-to-use context abstraction.*

<R.30> *Dependability.*

<R.31> *Loose coupling of services.*

- <R.32> *Service contracts.*
- <R.33> *Service autonomy.*
- <R.34> *Service abstraction.*
- <R.35> *Service reusability.*
- <R.36> *Service composability.*
- <R.37> *Service statelessness.*
- <R.38> *Service discoverability.*

<R.39> *Abstract interface validation support.*

<R.40> *Meta model with suitable properties.*

- <R.41> *Low complexity in the meta model.*
- <R.42> *Collaboration support* in the creation of context models.
- <R.43> *Dynamic extensibility of the context models at run time.*
- <R.44> Context validation.

<R.45> *Portable service executables.*

<R.47> Support of *emulators.*

<R.46> *App store.*

<R.48> *Crowdsourced development.*

<R.49> *Security-by-design.*

<R.50> *Scalability.*

A.2 Cascaded Retrieval of a Context Model

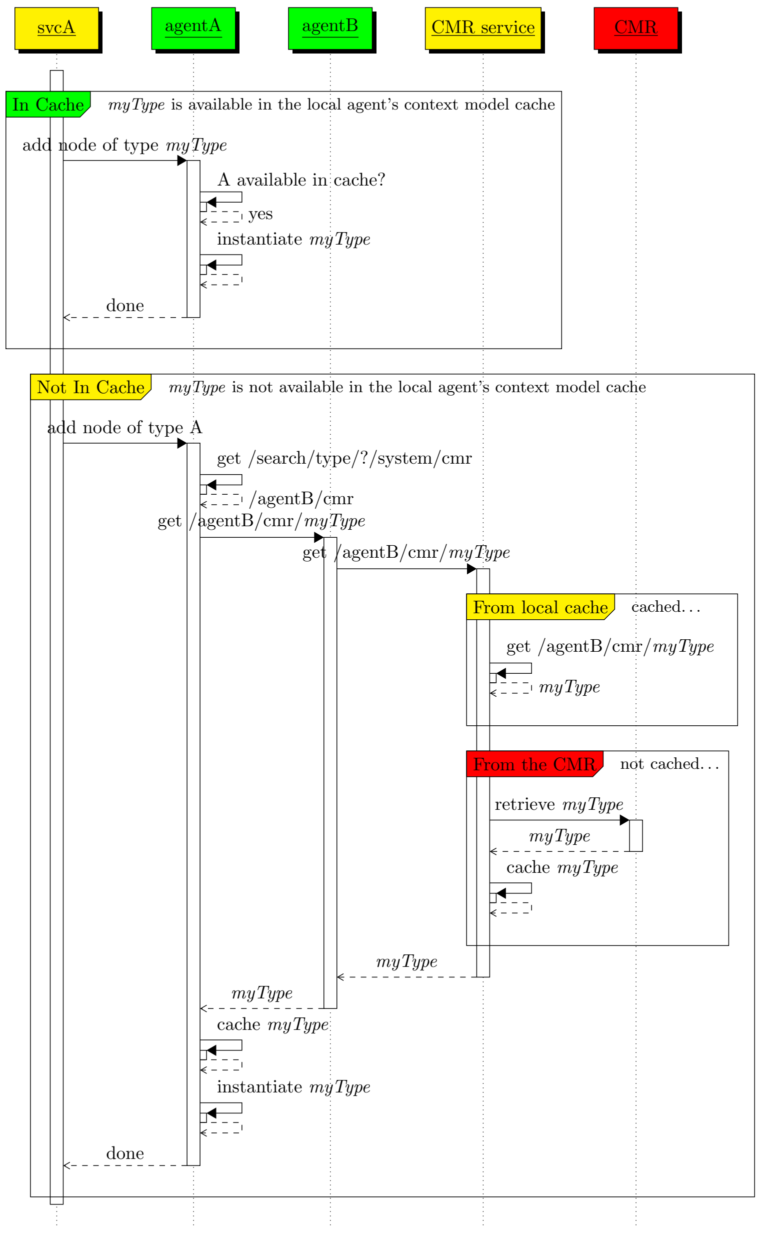


Figure A.1: SvcA is instantiating *myType* on agentA.

Fig. A.1 shows the instantiation of a context model *myType* via the CMR service as described in Sec. 5.7.1.

If the requested context model *myType* is available in the cache of the local Knowledge Agent (KA) *agentA* it is instantiated from there (upper part).
If it is not available, the CMR service (Sec. 5.7.1) is searched based on its type (lower part). If the CMR service has the context model *myType* in cache it is served from there. If not, the Context Model Repository (CMR) is contacted, the context model *myType* is loaded and locally cached. The KA *agentA* is also caching the context model for possible future retrieval as described in Sec. 5.7.1.

A.3 iLab Exercise for the User Study

Sec. 9.6 describes the user study that was conducted to evaluate the real world usability of the Virtual State Layer (VSL) programming abstraction. The entire study is based on the eLearning system *labsystem* [Pah09]. Fig. A.4 and Fig. A.5 show two screenshots from the instruction material that was provided for the user study in the *labsystem*.

Fig. A.4 shows a part of the preparatory material (Sec. 9.6.1). Fig. A.5 shows the Advanced Reasoning Service (ARS) creation part of the study (Sec. 9.6.4). The screenshot shows some of the free text inputs that were answered by the study participants.

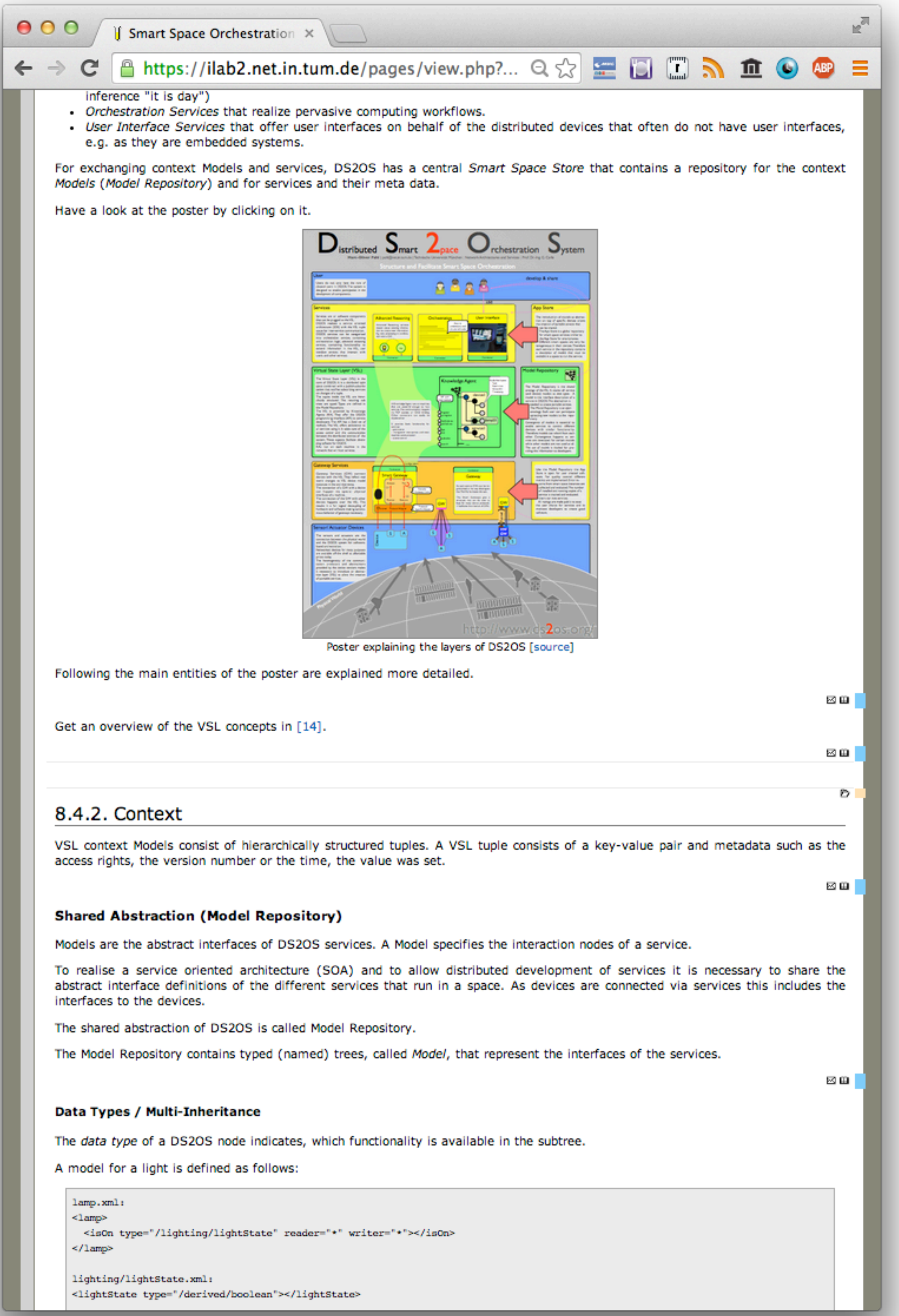
The teaching material that was created for the user study of this thesis is now part of the regular curriculum at Technische Universität München. It teaches future computer science students every semester about (future) Smart Space Orchestration.

The material is not printed here as it is about 160 pages. To get an idea how to use the VSL and to try it out, the material can be downloaded as ePub [Pah14b] Fig. A.2 shows a QR code that can be used to access the material in the epub format as electronic book.

Figure A.2: QR code to download the epub version of the user study material.

To get an idea how to use the VSL and to try it out, the material can be downloaded as web site [Pah14a]. Fig. A.3 shows a QR code that can be used to access the material as web site.

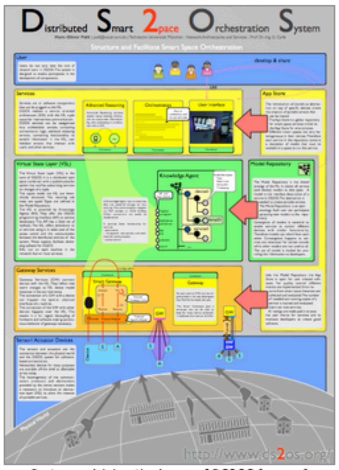
Figure A.3: QR code to download the web version of the user study material.


Smart Space Orchestration
https://ilab2.net.in.tum.de/pages/view.php?...

inference "it is day")
- *Orchestration Services* that realize pervasive computing workflows.
- *User Interface Services* that offer user interfaces on behalf of the distributed devices that often do not have user interfaces, e.g. as they are embedded systems.

For exchanging context Models and services, DS2OS has a central *Smart Space Store* that contains a repository for the context *Models* (*Model Repository*) and for services and their meta data.

Have a look at the poster by clicking on it.



Poster explaining the layers of DS2OS [source]

Following the main entities of the poster are explained more detailed.

Get an overview of the VSL concepts in [14].

8.4.2. Context

VSL context Models consist of hierarchically structured tuples. A VSL tuple consists of a key-value pair and metadata such as the access rights, the version number or the time, the value was set.

Shared Abstraction (Model Repository)

Models are the abstract interfaces of DS2OS services. A Model specifies the interaction nodes of a service.

To realise a service oriented architecture (SOA) and to allow distributed development of services it is necessary to share the abstract interface definitions of the different services that run in a space. As devices are connected via services this includes the interfaces to the devices.

The shared abstraction of DS2OS is called Model Repository.

The Model Repository contains typed (named) trees, called *Model*, that represent the interfaces of the services.

Data Types / Multi-Inheritance

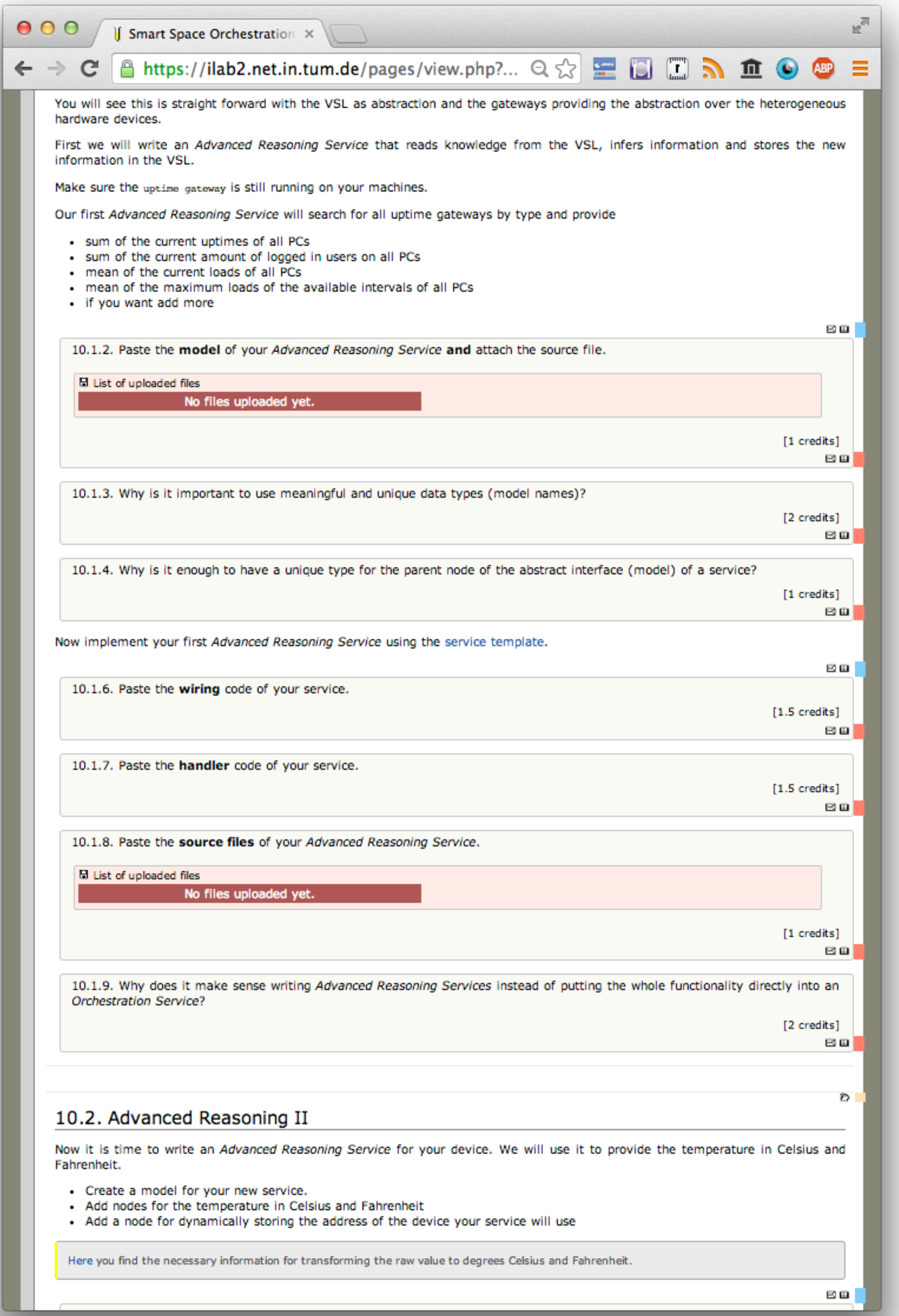
The *data type* of a DS2OS node indicates, which functionality is available in the subtree.

A model for a light is defined as follows:

```
lamp.xml:
<lamp>
  <isOn type="/lighting/lightState" reader="*" writer="*"></isOn>
</lamp>

lighting/lightState.xml:
<lightState type="/derived/boolean"></lightState>
```

Figure A.4: The screenshot shows the study material that was given to the users for their own preparation in the web-based eLearning system.



Smart Space Orchestration
https://ilab2.net.in.tum.de/pages/view.php?...

You will see this is straight forward with the VSL as abstraction and the gateways providing the abstraction over the heterogeneous hardware devices.

First we will write an *Advanced Reasoning Service* that reads knowledge from the VSL, infers information and stores the new information in the VSL.

Make sure the `uptime gateway` is still running on your machines.

Our first *Advanced Reasoning Service* will search for all uptime gateways by type and provide

- sum of the current uptimes of all PCs
- sum of the current amount of logged in users on all PCs
- mean of the current loads of all PCs
- mean of the maximum loads of the available intervals of all PCs
- if you want add more

10.1.2. Paste the **model** of your *Advanced Reasoning Service* **and** attach the source file.

List of uploaded files

No files uploaded yet.

[1 credits]

10.1.3. Why is it important to use meaningful and unique data types (model names)?

[2 credits]

10.1.4. Why is it enough to have a unique type for the parent node of the abstract interface (model) of a service?

[1 credits]

Now implement your first *Advanced Reasoning Service* using the service template.

10.1.6. Paste the **wiring** code of your service.

[1.5 credits]

10.1.7. Paste the **handler** code of your service.

[1.5 credits]

10.1.8. Paste the **source files** of your *Advanced Reasoning Service*.

List of uploaded files

No files uploaded yet.

[1 credits]

10.1.9. Why does it make sense writing *Advanced Reasoning Services* instead of putting the whole functionality directly into an *Orchestration Service*?

[2 credits]

10.2. Advanced Reasoning II

Now it is time to write an *Advanced Reasoning Service* for your device. We will use it to provide the temperature in Celsius and Fahrenheit.

- Create a model for your new service.
- Add nodes for the temperature in Celsius and Fahrenheit
- Add a node for dynamically storing the address of the device your service will use

Here you find the necessary information for transforming the raw value to degrees Celsius and Fahrenheit.

Figure A.5: The screenshot shows the study material that was used for the user study in the web-based eLearning system.

Bibliography

[ABB+86] M Accetta, R Baron, W Bolosky, D Golub, and R Rashid. Mach: A New Kernel Foundation For UNIX Development. 1986. Cited in sections 3.4.4 and 6.2.

[ABI12] ABI Research. 90 Million Homes Worldwide Will Employ Home Automation Systems by 2017 [online]. May 2012. URL: https://www.abiresearch.com/press/90-million-homes-worldwide-will-employ-home-automa. Cited in sections 2.1 and 2.4.4.

[Abo12] Gregory D Abowd. What next, ubicomp?: celebrating an intellectual disappearing act. In *UbiComp '12: Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM Request Permissions, September 2012. Cited in sections 1, 2.4.4, 2.5, 2.5.1, 2.5.1, 2.5.1, 2.5.3, 3.1, 3.2, 3.2.3, 3.12, 4.2, 4.5, 5.9, 6.3.3, 6.4, 7.2, and 10.1.

[AC90] Philip E Agre and David Chapman. What are plans for? *Robotics and Autonomous Systems*, 6(1-2):17–34, June 1990. Cited in section 3.6.4.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. Web Services: Concepts, Architectures and Applications, 1st edition. *Web Services: Concepts, Architectures and Applications, 1st edition*, 2004. Cited in sections 3.4, 3.4.2, 3.4.3, 3.4.4, 3.8, 3.8.1, 3.8.2, 3.8.2, 3.8.2, and 3.9.6.

[Ada79] Douglas Adams. *The Hitchhiker's Guide to the Galaxy*. The Hitchhiker's Guide to the Galaxy. Pan Books, 1979. Cited in section 3.3.4.

[AG89] Miklos Ajtai and Yuri Gurevich. Datalog vs. first-order logic. *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 142–147, 1989. Cited in section 4.5.5.

[AGG10] Ali Ziya Alkar, H Selcuk Gecim, and Muharrem Guney. Web Based ZigBee Enabled Home Automation System. In *2010 13th International Conference on Network-Based Information Systems (NBiS)*, pages 290–296. IEEE, September 2010. Cited in section 3.2.3.

[Ahl76] David Ahl. On Computer Languages. *The Best of Creative Computing*, 1, September 1976. Cited in sections 3.5.5 and 6.3.3.

[AHS01] Emile Aarts, Rick Harwig, and Martin Schuurmans. Ambient intelligence. *The invisible future*, November 2001. Cited in section 2.6.3.

[AKOSS+11] A M A H Al-Kuwari, C Ortega-Sanchez, A Sharif, V Digital Ecosystems Potdar, and Technologies Conference DEST 2011 Proceedings of the 5th IEEE International Conference on. User friendly smart home infrastructure: BeeHouse. In *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, March 2011. Cited in sections 3.2.2 and 3.2.3.

[AL86] A Avizienis and J C Laprie. Dependable computing: From concepts to design diversity. In *Proceedings of the IEEE*, pages 629–638, 1986. Cited in section 3.7.4.

[Ale13] Alexa Internet, Inc. Alexa Top 500 Global Sites [online]. November 2013. URL: http://www.alexa.com/topsites. Cited in section 2.3.3.

[AM00] Gregory D Abowd and Elizabeth D Mynatt. Charting past, present, and future research in ubiquitous computing. *Transactions on Computer-Human Interaction (TOCHI*, 7(1), March 2000. Cited in section 2.4.4.

[And] Android Open Source Project. Android Security FAQ | Android Developers [online]. URL: http://developer.android.com/guide/faq/security.html. Cited in section 7.3.

[Ant11] G. Anthes. Invasion of the mobile apps. *Communications of the ACM*, 54(9):16–18, 2011. Cited in section 3.10.4.

[AO04] P G Argyroudis and D O'Mahony. Securing Communications in the Smart Home - Springer. *Embedded and Ubiquitous Computing*, 2004. Cited in section 7.

[App10] Apple Incorporated. App Store Review Guidelines. Apple Inc., 2010. Cited in sections 3.10.7 and 7.4.2.

[App12] Apple Incorporated. iOS Security. Technical report, May 2012. Cited in sections 3.10.8, 5.5.5, 6.5.5, and 7.3.

[App13] Apple Incorporated. App Distribution Guide. Apple Inc., October 2013. Cited in sections 3.10.5, 3.10.8, 4.2, and 8.3.3.

[APV06] Brett Adams, Dinh Phung, and Svetha Venkatesh. Extraction of social context and application to personal multimedia exploration. In *the 14th annual ACM international conference*, page 987, New York, New York, USA, 2006. ACM Press. Cited in section 2.4.4.

[ASH95] ASHRAE. ANSI/ASHRAE standard 135-1995, BACnet. *BACnet A Data Communication Protocol for Building Automation and Control Networks*, 1995. Cited in section 4.4.2.

[Asi50] Isaac Asimov. *I, Robot.* Gnome Press, June 1950. Cited in section 5.5.6.

[ASZ+10] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50, April 2010. Cited in section 4.3.

[AY03] J Avery and J Yearwood. Dowl: A dynamic ontology language. In *ICWI 2003*, 2003. Cited in section 3.9.6.

[AZW06] Uwe Aßmann, Steffen Zschaler, and Gerd Wagner. Ontologies, Metamodels, and the Model-Driven Paradigm. In *Ontologies for Software Engineering and Technology*, pages 249–273. Springer, 2006. Cited in sections 3.12, 3.9.1, and A.3.

[Ban08] Massimo Banzi. Getting Started with Arduino, Ill edition. *Getting Started with Arduino, Ist edition*, October 2008. Cited in sections 2.4.2, 3.2.3, and 9.2.1.

[Bar05] Jakob E Bardram. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. In *Pervasive Computing*, pages 98–115. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. Cited in section 4.5.10.

[BBDR$^+$13] Sean Bechhofer, Iain Buchan, David De Roure, Paolo Missier, John Ainsworth, Jiten Bhagat, Philip Couch, Don Cruickshank, Mark Delderfield, Ian Dunlop, Matthew Gamble, Danius Michaelides, Stuart Owen, David Newman, Shoaib Sufi, and Carole Goble. Why linked data is not enough for scientists. *Future Generation Computer Systems*, 29(2), February 2013. Cited in section 9.2.2.

[BBH$^+$10] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180, April 2010. Cited in sections 3, 3.3.4, 3.9, 3.9.2, 3.3, 3.9.3, 3.9.5, 3.9.6, 4, 4.5.1, 5.1.2, 5.2.12, 8.3.2, and 8.3.6.

[BBS$^+$10] T Bläsing, L Batyuk, A.-D Schmidt, S.A Camtepe, and S Albayrak. An Android Application Sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 55–62, 2010. Cited in section 3.10.8.

[BCFF12] Paolo Bellavista, Antonio Corradi, Mario Fanelli, and Luca Foschini. A survey of context data distribution for mobile ubiquitous systems. *Computing Surveys (CSUR*, 44(4), August 2012. Cited in sections 2.4.1, 2.6.4, 3, 3.9, 4, 4.5.1, 4.5.16, and 4.3.

[BCG07] Fabio Luigi Bellifemine, Giovanni Caire, and Dominic Greenwood. Developing Multi-Agent Systems with JADE. *Wiley Series in Agent Technology*, April 2007. Cited in section 3.9.5.

[BCG12] Manfred Broy, María Victoria Cengarle, and Eva Geisberger. Cyber-Physical Systems: Imminent Challenges. In *Large-Scale Complex IT Systems. Development, Operation and Management.*, pages 1–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Cited in sections 2.4.4, 2.6.2, 3.3.1, 3.7.4, 5.5, 5.5.6, and 7.3.

[BCL12] Steven Bauer, David Clark, and William Lehr. The Evolution of Internet Congestion. February 2012. Cited in section 2.3.1.

[BCPR09] David F Bacon, Yiling Chen, David Parkes, and Malvika Rao. A Market-Based Approach to Software Evolution. In *Proceeding of the 24th ACM SIGPLAN conference companion*, page 973, New York, New

York, USA, 2009. ACM Press. Cited in sections 3.10.6, 3.10.7, 7.4, 7.4.2, and 7.4.3.

[BCQS07] C Bolchini, C A Curino, E Quintarelli, and F A Schreiber. A data-oriented survey of context models. *ACM Sigmod Record*, 36(4):19–26, 2007. Cited in sections 2.4.1, 3.9.2, 3.3, 3.9.3, 3.9.5, and 3.9.6.

[BD99] Bernd Bruegge and Allen A Dutoit. *Object-Oriented Software Engineering; Conquering Complex and Changing Systems*. Prentice Hall PTR, October 1999. Cited in section 6.3.5.

[BD07] Genevieve Bell and Paul Dourish. Yesterday's tomorrows: notes on ubiquitous computing's dominant vision. *Personal and ubiquitous computing*, 11(2), January 2007. Cited in sections 2.4.4, 2.5.3, 3.1, and 3.2.

[BDR07] M Baldauf, S Dustdar, and F Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2, April 2007. Cited in sections 3, 4, 4.5.1, and 4.5.2.

[Ber96] Philip A Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98, February 1996. Cited in sections 2.4.1, 2.4.4, and 5.6.3.

[BFK$^+$11] Sandford Bessler, Alexander Fischer, Eva Kühn, Richard Mordinyi, and Slobodanka Tomic. Using tuple-spaces to manage the storage and dissemination of spatial-temporal content. *Journal of Computer and System Sciences*, 77(2), March 2011. Cited in sections 3.4.2 and 5.2.13.

[BG04] Dan Brickley and R V Guha. RDF Vocabulary Description Language 1.0: RDF Schema, February 2004. Cited in section 3.9.4.

[BGJR92] D L Black, D B Golub, D P Julin, and R F Rashid. Microkernel Operating System Architecture and Mach. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992. Cited in section 3.4.4.

[BGRB11] Yérom-David Bromberg, Paul Grace, Laurent Reveillere, and Gordon S Blair. Bridging the interoperability gap: overcoming combined application and middleware heterogeneity. In *Middleware'11: Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag, December 2011. Cited in section 3.3.4.

[BHLM02] Staffan Björk, Jussi Holopainen, Peter Ljungstrand, and Regan Mandryk. Special Issue on Ubiquitous Games. *Personal and ubiquitous computing*, 6(5-6), January 2002. Cited in section 2.4.4.

[BHM$^+$04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture. Technical report, April 2004. Cited in sections 3.8.2, 3.8.2, and 3.8.2.

[BJMS12] A J Bernheim Brush, Jaeyeon Jung, Ratul Mahajan, and James Scott. HomeLab: shared infrastructure for home technology field studies. In *UbiComp '12: Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM Request Permissions, September 2012. Cited in sections 7.4.2 and 7.4.5.

[BLM$^+$11] A J Bernheim Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. Home Automation in the Wild: Challenges and Opportunities. In *the 2011 annual conference*, page 2115, New York, New York, USA, 2011. ACM Press. Cited in sections 2.1, 2.4.2, 2.4.4, 2.7, 3.2, 3.2.1, 3.2.2, 3.2.3, 3.2.5, 3.3, 3.3.1, 3.3.3, 3.3.4, 3.13.2, 4.2, 6.4, 7.2, and 7.2.6.

[BN84] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *Transactions on Computer Systems (TOCS*, 2(1), February 1984. Cited in section 3.4.2.

[BN91] Steven T Bushby and Michael Newman. The BACnet communication protocol for building automation systems. *ASHRAE Journal*, pages 14–21, April 1991. Cited in section 4.4.2.

[BPR99] Bernd Brügge, Ralf Pfleghar, and Thomas Reicher. OWL: An Object-Oriented Framework for Intelligent Home and Office Applications. In *CoBuild '99: Proceedings of the Second International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture*. Springer-Verlag, October 1999. Cited in sections 3.1.1, 3.2.1, and 3.3.3.

[BRa09] Yérom-David Bromberg, L Réveillère, and et al. Automatic generation of network protocol gateways. *Middleware 2009*, 2009. Cited in section 3.3.4.

[Bro86] R A Brooks. A Robust Layered Control System For A Mobile Robot. *Robotics and Automation*, 1986. Cited in section 3.6.1.

[Bro90] R A Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 1990. Cited in sections 3.6.1 and 3.6.

[Bro91a] R A Brooks. Artificial Intelligence - Intelligence without representation. *Artificial intelligence*, 1991. Cited in sections 3.4.2, 3.6.4, and 3.6.4.

[Bro91b] R A Brooks. Intelligence Without Reason, 1991. Cited in sections 3.6 and 3.6.1.

[BSG12] Francisco J Ballesteros, Enrique Soriano, and Gorka Guardiola. Octopus: An Upperware based system for building personal pervasive environments. *Journal of Systems and Software*, 85(7), July 2012. Cited in section 2.4.1.

[But11] M Butler. Android: Changing the Mobile Landscape. *Pervasive Computing, IEEE*, 10(1):4–7, 2011. Cited in sections 3.10.7, 3.10.9, and 7.2.1.

[Can76] Alexander B Cannara. Toward A Human Computer Language. *The Best of Creative Computing*, 1, September 1976. Cited in sections 3.5.5 and 6.3.3.

[Car95] C F Cargill. Standards Policy for Information Infrastructure. *Standards policy for information infrastructure*, 1995. Cited in sections 3.3.2 and 3.2.

[CD05] Diane J Cook and Sajal K Das. *Smart environments*. technologies, protocols, and applications. Wiley-Interscience, Hoboken, NJ, USA, 2005. Cited in sections 2.4.1, 3.4, 3.4.3, 3.4.4, 3.10.8, and 7.

[CD07] Diane J Cook and Sajal K Das. How smart are our environments? An updated look at the state of the art. *Pervasive and Mobile Computing*, 3(2):53–73, March 2007. Cited in sections 2.4.4, 2.5.1, 2.5.3, 3, 3.3, 3.3.1, 3.10, 4, 4.5.1, and 7.

[CD12] Diane J Cook and Sajal K Das. Pervasive computing at scale: Transforming the state of the art. *Pervasive and Mobile Computing*, 8(1):22–35, February 2012. Cited in sections 2.4.2, 2.4.4, 3, 3.1, 4, 4.5.1, 5.5, 5.5.6, 6.4, and 7.3.

[CDB$^+$12] Marco Conti, Sajal K Das, Chatschik Bisdikian, Mohan Kumar, Lionel M Ni, Andrea Passarella, George Roussos, Gerhard Tröster, Gene Tsudik, and Franco Zambonelli. Looking ahead in pervasive computing: Challenges and opportunities in the era of cyber–physical convergence. *Pervasive and Mobile Computing*, 8(1):2–21, February 2012. Cited in section 7.3.

[Cer88] V G Cerf. IAB recommendations for the development of Internet network management standards. Technical report, Internet Engineering Task Force, 1988. Cited in section 4.4.

[CES83] E M Clarke, E A Emerson, and A P Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Request Permissions, January 1983. Cited in section 5.5.6.

[CFK$^+$13] Tony Clark, Ulrich Frank, Vinay Kulkarni, Balbir Barn, and Dan Turk. Domain specific languages for the model driven organization. In *GlobalDSL '13: Proceedings of the First Workshop on the Globalization of Domain Specific Languages*. ACM Request Permissions, July 2013. Cited in section 3.3.1.

[CGH$^+$02] E Callaway, P Gorday, L Hester, J A Gutierrez, M Naeve, B Heile, and V Bahl. Home networking with IEEE 802.15.4: a developing standard for low-rate wireless personal area networks. *IEEE Communications Magazine*, 40(8):70–77, August 2002. Cited in sections 3.2.1 and 3.2.2.

[CGR11] M N Cortimiglia, A Ghezzi, and F Renga. Mobile Applications and Their Delivery Platforms. *IT Professional*, 13(5):51–56, 2011. Cited in section 3.10.3.

[Che04] S Chemishkian. Experimental Bridge LONWORKS® / UPnPTM 1.0. *Consumer Communications and Networking Conference*, 2004. Cited in section 3.3.4.

[Cho56] N Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956. Cited in sections 3.5.3, 3.5.5, and 6.3.3.

[Chu10] Autonomic computing technologies for cyber-physical systems. *ICACT 2010*, 2:1009–1014, 2010. Cited in sections 2.6.2 and 3.7.4.

[Cis12] Cisco. Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2011–2016 . Technical report, February 2012. Cited in sections 2.3.1, 2.3.1, 2.3.2, 2.3.3, 2.5.1, and 3.10.

[CJ08] Cisco and Johnson Controls. Building Automation System over IP (BAS/IP). Technical report, San Jose, August 2008. Cited in sections 2.7, 3.2.1, and 3.2.2.

[CK00] Guanling Chen and David Kotz. A Survey of Context-Aware Mobile Computing Research. *A Survey of Context-Aware Mobile Computing Research*, November 2000. Cited in section 2.4.1.

[CKKC07] Wonik Choi, Hyunduk Kim, Jinsu Kim, and Jinseok Chae. A Context-Aware Framework for Mobile Navigation Service. In *Computer and Information Technology, 2007. CIT 2007. 7th IEEE International Conference on*, pages 423–428, 2007. Cited in section 2.4.4.

[Cla93] R Clarke. Asimov's laws of robotics: implications for information technology-Part I. *Computer*, 26(12):53–61, 1993. Cited in section 5.5.6.

[CLZ00] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mobile-Agent Coordination Models for Internet Applications. *Computer*, 33(2):82–89, February 2000. Cited in sections 3.4.1, 3.2, and 3.4.2.

[Coe98] Michael H Coen. Design principles for intelligent environments. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*. American Association for Artificial Intelligence, July 1998. Cited in section 2.4.1.

[Com96] Commission of the European Communities. *Standardization and the Global Information Society: The European Approach*. Technical Report 96, Brussels, July 1996. Cited in section 3.3.2.

[CPC$^+$04] Harry Chen, Filip Perich, Dipanjan Chakraborty, Tim Finin, and Anupam Joshi. *Intelligent Agents Meet Semantic Web in a Smart Meeting Room*. IEEE Computer Society, July 2004. Cited in section 2.4.4.

[CPW99] M Coen, B Phillips, and N Warshawsky. Meeting the computational needs of intelligent environments: The metaglue system. In *Proceedings of MANSE*, pages 210–213, 1999. Cited in section 3.3.3.

[CSF$^+$08] D Cooper, S Santesson, S Farrell, S Boeyen, R Housley, and W Polk. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Technical report, Internet Engineering Task Force, 2008. Cited in sections 6.5.1 and 7.3.4.

[CSG07] Marshini Chetty, Ja-Young Sung, and Rebecca E Grinter. How smart homes learn: the evolution of the networked home and household. In *UbiComp '07: Proceedings of the 9th international conference on Ubiquitous computing*. Springer-Verlag, September 2007. Cited in section 3.3.1.

[Dar10] Sarah Darby. Smart metering: what potential for householder engagement? *Building Research & Information*, 38(5):442–457, October 2010. Cited in section 2.4.4.

[DAS01] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2), December 2001. Cited in sections 3.3.3, 3.3.3, and 5.6.5.

[DBB$^+$88] U Dayal, B Blaustein, A Buchmann, U Chakravarthy, M Hsu, R Ledin, D McCarthy, A Rosenthal, S Sarin, M J Carey, M Livny, and R Jauhari. The HiPAC project: combining active databases and timing constraints. *SIGMOD Record*, 17(1), March 1988. Cited in section 3.6.4.

[DDMS12] G Denker, N Dutt, S Mehrotra, and MO Stehr. Resilient dependable cyber-physical systems: a middleware perspective . *J Internet Serv Appl*, 3:41–49, 2012. Cited in sections 2.6.2, 3.2.5, 3.7.4, 5.5, 5.5.6, and 7.3.

[Dét02] Francoise Détienne. Software Design - Cognitive Aspects. 2002. Cited in sections 3.5.2 and 5.4.1.

[Dey01] A K Dey. Understanding and Using Context. *Personal and ubiquitous computing*, 5(1):4–7, 2001. Cited in section 2.4.1.

[DHJT$^+$10] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. sMAP: a simple measurement and actuation profile for physical information. In *SenSys '10: Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM Request Permissions, November 2010. Cited in section 3.10.9.

[DHK11] S Dawson-Haggerty and A Krioukov. Experiences Integrating Building Data with sMAP. *UC Berkeley Technical Report No. UCB/EECS-2012-21*, 2011. Cited in section 5.6.3.

[DHKT$^+$13] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. BOSS: Building Operating System Services. *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013. Cited in sections 2.4.1, 2.4.2, 3.2.1, and 4.5.7.

[Dis] Distributed Management Task Force. Distributed Management Task Force [online]. URL: http://dmtf.org/. Cited in section 4.4.1.

[DLG07] Maurice Dixon, Simon C Lambert, and Julian R Gallop. *Distributed Data Mining and Knowledge Management with Networks of Sensor Arrays*. Springer US, Boston, MA, 2007. Cited in section 3.2.

[DM12] P Dourish and S Mainwaring. Ubicomp's Colonial Impulse. *UbiComp '12: Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, 2012. Cited in sections 2.5, 2.5.2, 3.1, 3.3.2, 4.2, 4.5, 4.5.17, and 6.2.

[DMA$^+$10] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A J Bernheim Brush, Bongshin Lee, Stefan Saroiu, and Victor Bahl. The Home Needs an Operating System (and an App Store). In *the Ninth ACM SIGCOMM*

Workshop, pages 1–6, New York, New York, USA, 2010. ACM Press. Cited in sections 2.1 and 4.5.5.

[DMA+12] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A J Bernheim Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *NSDI'12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, April 2012. Cited in sections 2.4.1, 2.4.2, 2.4.4, 3.3.3, 4.5.5, and 5.6.3.

[Dou03] Paul Dourish. The Appropriation of Interactive Technologies: Some Lessons from Placeless Documents. *Computer Supported Cooperative Work*, 12(4), September 2003. Cited in section 6.4.

[DSFA99] Anind K Dey, Daniel Salber, Masayasu Futakawa, and Gregory D Abowd. An Architecture to Support Context-Aware Applications. *UIST'99*, 1999. Cited in section 4.5.8.

[DWJG02] Bert J Dempsey, Debra Weiss, Paul Jones, and Jane Greenberg. Who is an open source software developer? *Communications of the ACM*, 45(2):67–72, February 2002. Cited in section 4.2.

[EBDN02] W K Edwards, V Bellotti, A K Dey, and M W Newman. Stuck in the Middle: Bridging the Gap Between Design, Evaluation, and Middleware. 2002. Cited in sections 4.5, 5.6.7, and 8.3.3.

[EBM05] C Endres, A Butz, and A MacWilliams. A survey of software infrastructures and frameworks for ubiquitous computing. *Mobile Information Systems*, 2005. Cited in sections 3, 4, 4.5.1, and 4.5.2.

[EFGK03] Patrick Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(June 2003):114–131, June 2003. Cited in section 3.4.2.

[EG01] W Keith Edwards and Rebecca E Grinter. At home with ubiquitous computing: Seven challenges. *Ubicomp 2001: Ubiquitous Computing*, 2001. Cited in sections 3.2.2, 3.3, and 3.3.1.

[Ega05a] D Egan. The emergence of ZigBee in building automation and industrial control. *Computing & Control Engineering Journal*, 2005. Cited in section 3.2.1.

[Ega05b] D Egan. The emergence of ZigBee in building automation and industrial control. *Computing & Control Engineering Journal*, 16(2):14–19, 2005. Cited in section 3.3.1.

[EHS10] Basem S El-Haik and Adnan Shaout. Software Design for Six Sigma: A Roadmap for Excellence, 1st edition. *Software Design for Six Sigma: A Roadmap for Excellence, 1st edition*, November 2010. Cited in section 3.1.1.

[EK95] D R Engler and M F Kaashoek. Exokernel. *ACM SIGOPS Operating Systems Review*, 1995. Cited in section 3.4.4.

[Eri12] Ericsson. Emerging App Culture. Technical report, 2012. Cited in sections 2.3.3 and 3.10.4.

[Erl05] Thomas Erl. *Service-Oriented Architecture*. Concepts, Technology, and Design. Prentice Hall, August 2005. Cited in sections 3.8, 3.8.1, 3.11, and 3.8.2.

[ERS10] J Edmonds, L Raschid, and H Sayyadi. Exploiting Social Media to Provide Humanitarian Users with Event Search and Recommendations. *Social Media for Humanitarian Users and Event Search*, 2010. Cited in section 2.4.4.

[EWCS96] Y Endo, Z Wang, J B Chen, and M I Seltzer. Using Latency to Evaluate Interactive System Performance . *ACM SIGOPS Operating Systems Design and Implementation*, 1996. Cited in section 9.3.1.

[Fel00] R Fellows. Connecting BACnet® to the Internet. *Heating/piping/air conditioning engineering*, 72(3):65–71, 2000. Cited in section 3.2.1.

[FHA99] E Freeman, S Hupfer, and K Arnold. JavaSpaces: Principles, Patterns, and Practice. 1999. Cited in sections 3.4.1, 3.4.2, and 5.3.2.

[Fie02] RT Fielding. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2, 2002. Cited in sections 2.6.1 and 3.2.1.

[Fis98] G Fischer. Beyond “couch potatoes”: From consumers to designers. *Computer Human Interaction*, 1998. Cited in section 4.2.

[FLC$^+$00] Michael D Fabrlzio, Benjamin R Lee, David Y Chan, Daniel Stoianovici, Thomas W Jarrett, Calvin Yang, and Louis R Kavoussi. Effect of Time Delay on Surgical Performance During Telesurgical Manipulation. *Journal of Endourology*, 14(2):133–138, March 2000. Cited in section 9.3.1.

[Flo67] R W Floyd. Assigning Meanings to Programs. *Mathematical aspects of computer science*, 1967. Cited in section 5.5.6.

[FLR13] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. Mobile cloud computing: A survey. *Future Generation Computer Systems*, 29(1):84–106, January 2013. Cited in section 2.3.3.

[FMF$^+$12] Francois Fouquet, Brice Morin, Franck Fleurey, Olivier Barais, Noel Plouzeau, and Jean-Marc Jezequel. A dynamic component model for cyber physical systems. In *CBSE ’12: Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*. ACM Request Permissions, June 2012. Cited in section 7.2.1.

[Gas88] L Gasser. A Survey of Distributed Artificial Intelligence. *Communication*, 1988. Cited in section 3.6.5.

[GBRB12] P Grace, Yérom-David Bromberg, L Réveillère, and G Blair. OverStar: An Open Approach to End-to-End Middleware Services in Systems of Systems - Springer. *Middleware 2012*, 2012. Cited in sections 3.2.2 and 3.3.4.

[GC03] A G Ganek and T A Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003. Cited in sections 3.7.1, 3.7.3, and 3.7.3.

[GDH01] R Grimm, J Davis, and B Hendrickson. Systems Directions for Pervasive Computing. *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, 2001. Cited in sections 5 and 5.6.5.

[GDLM04] R Grimm, J Davis, E Lemar, and A Macbeth. System support for pervasive applications. *ACM Transactions on Computer Systems*, 22(4):421–468, 2004. Cited in sections 4.5.13, 5.6.5, and 9.6.5.

[GEBF04] L Garcés-Erice, EW Biersack, and PA Felber. Hierarchical Peer-to-Peer Systems . *Parallel Processing Letters*, 2004. Cited in section 6.4.6.

[Gel85] D Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1985. Cited in section 3.4.2.

[GG92] M Gien and L Grob. Micro-kernel Based Operating Systems: Moving UNIX onto Modern System Architectures . In *Proceedings of the UniForum*, San Francisco, 1992. Cited in sections 3.4.4 and 6.2.

[GGS09] Yannis Georgalis, Dimitris Grammenos, and Constantine Stephanidis. Middleware for Ambient Intelligence Environments: Reviewing Requirements and Communication Technologies. In *UAHCI '09: Proceedings of the 5th International on ConferenceUniversal Access in Human-Computer Interaction. Part II: Intelligent and Ubiquitous Interaction Environments*. Springer-Verlag, July 2009. Cited in sections 3, 4, and 4.5.1.

[GJ86] P Green Jr. Protocol conversion. *IEEE Transactions on Communication*, 1986. Cited in sections 3.3.1 and 3.3.4.

[GJ91] Narain H Gehani and H V Jagadish. Ode as an Active Database: Constraints and Triggers. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc, September 1991. Cited in section 3.6.4.

[GJP10] Katja Gilly, Carlos Juiz, and Ramon Puigjaner. An up-to-date survey in web load balancing. *World Wide Web*, 14(2):105–131, December 2010. Cited in sections 6.6.1, 7.2.6, 7.6.2, and 7.7.1.

[GM95] J Gosling and H McGilton. The Java language environment. *Whitepaper, Sun Microsystems, Inc.*, 1995. Cited in sections 3.10.3, 3.10.3, and 6.3.1.

[GN87] Michael R Genesereth and Nils J Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann Publishers, 1987. Cited in sections 3.5.4, 3.6.4, 3.8, 3.9.1, and A.3.

[Gol76] Marian Goldeen. Learning About Smalltalk. *The Best of Creative Computing*, September 1976. Cited in sections 3.5.5 and 6.3.3.

[Goo] Google Inc. Android Developer Reference [online]. URL: http://developer.android.com/reference/packages.html. Cited in sections 3.10.3, 4.2, 7.4.2, and 8.3.3.

[GPJB07] A Gupta, S Paul, Q Jones, and C Borcea. Automatic identification of informal social groups and places for geo-social recommendations - International Journal of Mobile Network Design and Innovation - Volume 2, Number 3-4/2007 - Inderscience Publishers. *International Journal of Mobile Network Design and Innovation*, 2007. Cited in section 2.4.4.

[GPP$^+$98] Michael P Georgeff, Barney Pell, Martha E Pollack, Milind Tambe, and Michael Wooldridge. The Belief-Desire-Intention Model of Agency. In *ATAL '98: Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*. Springer-Verlag, July 1998. Cited in sections 3.6.3 and 3.6.3.

[GPZ05] Tao Gu, Hung Keng Pung, and Da Qing Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, 28(1):1–18, January 2005. Cited in section 4.5.9.

[GR11] Mark H Goadrich and Michael P Rogers. Smart Smartphone Development: iOS versus Android . In *the 42nd ACM technical symposium*, pages 607–612, New York, New York, USA, 2011. ACM Press. Cited in sections 3.10.3, 3.10.5, and 8.3.3.

[Gri04] R Grimm. One.world: Experiences with a Pervasive Computing Architecture . *Pervasive Computing*, 2004. Cited in section 4.5.13.

[Grø09] Endre Grøtnes. Standardization as open innovation: two cases from the mobile industry. *Information Technology & People*, 22(4):367–381, 2009. Cited in section 3.3.2.

[Gru93] T R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 1993. Cited in sections 3.9.1 and A.3.

[GSSS02] D Garlan, D P Siewiorek, A Smailagic, and P Steenkiste. Project Aura: toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, April 2002. Cited in section 4.5.4.

[GWB10] Vânia Gonçalves, Nils Walravens, and Pieter Ballon. How about an App Store? Enablers and Constraints in Platform Strategies for Mobile Network Operators. In *2010 Ninth International Conference on Mobile Business and 2010 Ninth Global Mobility Roundtable (ICMB-GMR)*, pages 66–73. IEEE, 2010. Cited in sections 3.10, 3.10, 3.10.4, 3.10.5, and 3.10.6.

[GZ13] Iryna Gurevych and Torsten Zesch. Collective intelligence and language resources: introduction to the special issue on collaboratively constructed language resources. *Language Resources and Evaluation*, 47(1), March 2013. Cited in section 7.4.

[GZI10] Bin Guo, Daqing Zhang, and Michita Imai. Toward a cooperative programming framework for context-aware applications. *Personal and ubiquitous computing*, 15(3):221–233, August 2010. Cited in sections 4.5.12 and 8.4.4.

[HA02] R Harwig and Emile Aarts. Ambient Intelligence: invisible electronics emerging. In *Interconnect Technology Conference, 2002. Proceedings of the IEEE 2002 International*, pages 3–5, 2002. Cited in section 2.6.3.

[hAN99] Heinz-Gerd hegering, Sebastian Abeck, and Bernhard Neumair. Integrated Management of Networked Systems, 1999. Cited in sections 4.4.1 and 4.4.1.

[Har03] R Harper. Inside the Smart Home. 2003. Cited in sections 2.4.4 and 3.2.2.

[Har11] W Hardaker. Transport Layer Security (TLS) Transport Model for the Simple Network Management Protocol (SNMP). Technical report, Internet Engineering Task Force, 2011. Cited in section 8.3.1.

[HC03a] Christopher K Hess and Roy H Campbell. A Context-Aware Data Management System for Ubiquitous Computing Applications. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society, May 2003. Cited in section 4.5.3.

[HC03b] Christopher K Hess and Roy H Campbell. An application of a context-aware file system. *Personal and ubiquitous computing*, 7(6), December 2003. Cited in section 4.5.3.

[HCJG97] CHA Hsieh, M T Conte, T L Johnson, and J C Gyllenhaal. Optimizing NET Compilers for Improved Java Performance . *Computer*, 1997. Cited in section 6.3.1.

[Hel05] Sumi Helal. Programming pervasive spaces. *Pervasive Computing, IEEE*, 4(1), 2005. Cited in section 2.4.4.

[HGH96] CHA Hsieh, J C Gyllenhaal, and W W Hwu. Java bytecode to native code translation. *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 90–99, 1996. Cited in section 6.3.1.

[HI04] K Henricksen and Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, pages 77–86, 2004. Cited in section 4.5.11.

[Hil13] F Hillebrand. The creation of standards for global mobile communication: GSM and UMTS standardization from 1982 to 2000. *Wireless Communications, IEEE*, 20(5):24–33, 2013. Cited in section 3.3.2.

[HIM05a] K Henricksen, Jadwiga Indulska, and T McFadden. Middleware for Distributed Context-Aware Systems . *International Symposium on Distributed Objects and Applications (DOA)*, 2005. Cited in sections 2.4.1, 2.4.4, 2.6.4, 2.6, 3, 4, 4.5.1, 4.5.11, and 8.3.6.

[HIM05b] Karen Henricksen, Jadwiga Indulska, and Ted McFadden. Modelling Context Information with ORM. In *On the Move to Meaningful Internet Systems*, pages 626–635. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. Cited in section 4.5.11.

[Hin99] Debby Hindus. The Importance of Homes in Technology Research. In *Cooperative buildings. Integrating Information, Organizations, and Architecture*, pages 199–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. Cited in sections 3.2.1, 3.3.2, and 6.2.

[HIR02] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. In *Pervasive Computing*, pages 167–180. Springer Berlin Heidelberg, Berlin, Heidelberg, August 2002. Cited in sections 2.4.1 and 3.9.6.

[HJ02] Clyde W Holsapple and K D Joshi. A collaborative approach to ontology design. *Communications of the ACM*, 45(2), February 2002. Cited in sections 3.9.6, 5, 5.1.2, and 7.

[HKLC08] Yueh-Min Huang, Yen-Hung Kuo, Yen-Ting Lin, and Shu-Chen Cheng. Toward interactive mobile synchronous learning environment with context-awareness service. *Computers & Education*, 51(3), November 2008. Cited in section 2.4.4.

[HLH$^+$11] Kathryn Huberty, Mark Lipacis, Adam Holt, Ehud Gelblum, Scott Devitt, Benjamin Swinburne, Francois Meunier, Keon Han, Frank A Y Wang, Jasmine Lu, Grace Chen, Bill Lu, Masahiro Ono, Mia Nagasaka, Kazuo Yoshikawa, and Mathew Schneider. Tablet Demand and Disruption. Technical report, Global Technology, Media and Telecommunications Equipment Team, February 2011. Cited in section 2.3.2.

[HLM$^+$97] V. Hartkopf, V. Loftness, A. Mahdavi, S. Lee, and J. Shankavaram. An integrated approach to design and engineering of intelligent buildings—The Intelligent Workplace at Carnegie Mellon University. *Automation in Construction*, 6(5):401–415, 1997. Cited in section 3.2.1.

[HM08] Markus C Huebscher and Julie A McCann. A survey of autonomic computing—degrees, models, and applications. *Computing Surveys (CSUR*, 40(3), August 2008. Cited in sections 3.7.1, 3.7.2, and 3.7.4.

[HMEZ$^+$05] Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura, and Erwin Jansen. The Gator Tech Smart House: A Programmable Pervasive Space. *Computer*, 38(3), March 2005. Cited in sections 2.4.4, 3.3.3, 3.7.2, and 3.7.4.

[Hoa69] CAR Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969. Cited in section 5.5.6.

[Hoa78] C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. Cited in sections 3.3.4 and 5.5.6.

[HPJP06] Intark Han, Hong-Shik Park, Youn-Kwae Jeong, and Kwang-Roh Park. An integrated home server for communication, broadcast reception, and home automation. *IEEE Transactions on Consumer Electronics*, 52(1):104–109, February 2006. Cited in section 3.3.4.

[HPSvH03] Ian Horrocks, Peter F Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: the making of a Web Ontology Language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7–26, December 2003. Cited in section 3.9.6.

[HPW02] D Harrington, R Presuhn, and B Wijnen. An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. Technical report, Internet Engineering Task Force, 2002. Cited in section 4.4.1.

[HR88] B Hayes-Roth. A blackboard architecture for control. *Distributed Artificial Intelligence*, June 1988. Cited in section 3.4.2.

[HR06] Karen Henricksen and Ricky Robinson. A Survey of Middleware for Sensor Networks: State-of-the-Art and Future Directions . In *the international workshop*, pages 60–65, New York, New York, USA, 2006. ACM Press. Cited in sections 3, 4, and 4.5.1.

[HSK09] Jong-yi Hong, Eui-ho Suh, and Sung-Jin Kim. Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4):8509–8522, May 2009. Cited in sections 3, 4, and 4.5.1.

[HWN10] Patrick Holroyd, Phil Watten, and Paul Newbury. Why Is My Home Not Smart? In *Aging Friendly Technology for Health and Independence*, pages 53–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. Cited in sections 2.4.1, 2.4.4, and 3.3.1.

[II12] ISO, International Organization for Standardization and IEC, International Electrotechnical Commission. ISO/IEC Directives, Part 1. ISO/IEC, 2012. Cited in section 3.3.2.

[III98] International Organization for Standardization, International Electrotechnical Commission, and International Telecommunication Union ITU. ITU-T X.902 | ISO/IEC 10746-1, December 1998. Cited in section 2.4.

[ISO08] ISO, International Organization for Standardization. ISO/IEC 18000, 2008. Cited in section 2.1.

[ITU05] International Telecommunication Union ITU. ITU Internet Reports 2005: The Internet of Things 2005 . Technical Report 7, Geneva, November 2005. Cited in sections 2.6.1, 3.2, and 3.3.2.

[ITU11] International Telecommunication Union ITU. *Measuring the Information Society 2011*. Technical report, Geneva, September 2011. Cited in sections 2.3.1, 2.3.1, 2.5.1, and 3.10.

[ITU13] International Telecommunication Union ITU. ICT Facts and Figures. Technical report, Geneva, February 2013. Cited in sections 2.3.1, 2.5.1, and 3.10.

[JL93] Philip Nicholas Johnson-Laird. *The Computer and the Mind: An Introduction to Cognitive Science.* Fontana Press, London, 1993. Cited in sections 3.1.1, 3.5.1, 3.5.2, 3.5.3, 3.5.4, 3.5, 3.5.4, 3.5.5, 3.5.5, 3.5.5, and 6.3.3.

[JPSM13] R Jose, H Pinto, B Silva, and A Melro. Pins and posters: Paradigms for content publication on situated displays. *Computer Graphics and Applications, IEEE*, 33(2):64–72, 2013. Cited in section 2.5.1.

[JPW98] Kai Jakobs, Rob Procter, and Robin Williams. User participation in standards setting—the panacea? *StandardView*, 6(2), June 1998. Cited in sections 3.3.2, 3.2, and 3.3.2.

[JPW01] K Jakobs, R Procter, and R Williams. The making of standards: looking inside the work groups. *IEEE Communications Magazine*, 39(4), April 2001. Cited in section 3.3.2.

[JSB00] Julie A Jacko, Andrew Sears, and Michael S Borella. The effect of network delay and media on user perceptions of web resources. *Behaviour & Information Technology*, 19(6):427–439, January 2000. Cited in section 9.3.1.

[JST$^+$09] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *the 5th international conference*, page 1, New York, New York, USA, 2009. ACM Press. Cited in sections 5.2.13 and 10.3.

[KA06] S Karapiperis and D Apostolou. Consensus building in collaborative ontology engineering processes. *Journal of Universal Knowledge Management*, 2006. Cited in sections 3.9.6, 5, 5.1.2, and 7.

[Kay90] A Kay. User interface: A personal view. *The art of human-computer interface design*, 1990. Cited in sections 3.5.5 and 6.3.3.

[Kay96] Alan C Kay. The early history of Smalltalk. *History of programming languages—II*, January 1996. Cited in sections 3.5.5 and 6.3.3.

[KBY$^+$12] Ji Eun Kim, George Boulos, John Yackovich, Tassilo Barth, Christian Beckel, and Daniel Mosse. Seamless Integration of Heterogeneous Devices and Access Control in Smart Homes. In *IE '12: Proceedings of the 2012 Eighth International Conference on Intelligent Environments*. IEEE Computer Society, June 2012. Cited in sections 3.2.2 and 7.2.1.

[KC03] J O Kephart and D M Computer Chess. The vision of autonomic computing. *Computer*, 36(1), 2003. Cited in sections 3.7.1, 3.7.2, 3.7.4, and 6.4.10.

[KD07] N Kara and O A Dragoi. Reasoning with Contextual Data in Telehealth Applications. In *Wireless and Mobile Computing, Networking and Communications, 2007. WiMOB 2007. Third IEEE International Conference on*, page 69. IEEE Computer Society, 2007. Cited in section 2.4.4.

[KFKC12] Andrew Krioukov, Gabe Fierro, Nikita Kitaev, and David Culler. Building application stack (BAS). In *the Fourth ACM Workshop*, page 72, New York, New York, USA, 2012. ACM Press. Cited in sections 2.1, 2.4.2, 3.2, 3.2.1, 3.3.3, 3.3.3, 3.10.9, 4.4.2, 4.5.7, and 5.6.3.

[KG77] A Kay and A Goldberg. Personal Dynamic Media. *Computer*, 10(3):31–41, 1977. Cited in sections 2.3.3, 3.5.5, and 6.3.3.

[KHC10] Eunju Kim, Sumi Helal, and Diane Cook. Human Activity Recognition and Pattern Discovery. *IEEE Pervasive Computing*, 9(1):48–53, January 2010. Cited in section 8.3.2.

[Kim10] K Kimbler. App store strategies for service providers. In *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on*, pages 1–5, 2010. Cited in sections 3.10, 3.10, 3.10.4, and 3.10.6.

[Kjæ07] Kristian Ellebæk Kjær. A survey of context-aware middleware. *Proceedings of the international workshop on Middleware for sensor networks*, pages 148–155, February 2007. Cited in sections 3, 4, and 4.5.1.

[KK07] S Kalasapur and M Kumar. Dynamic Service Composition in Pervasive Computing . *IEEE Transaction on Parallel and Distributed Systems*, 2007. Cited in section 4.5.4.

[KK10] HJ Kim and I Kim. The Success Factors for App Store-Like Platform Businesses from the Perspective of Third-Party Developers: An Empirical Study Based on A Dual Model Framework. In *Proceedings of the Pacific Asia Conference on Information Systems 2010*, Taipei, Taiwan, July 2010. School of Business, Yonsei University, Seoul, South Korea. Cited in section 3.10.6.

[KKK06] Min-Soo Kang, Kyung Mi Kim, and Hee-Cheol Kim. A questionnaire study for the design of smart home for the elderly. In *e-Health Networking, Applications and Services, 2006. HEALTHCOM 2006. 8th International Conference on*, pages 265–268, 2006. Cited in section 2.4.4.

[KKR$^+$13] M Knappmeyer, S L Kiani, E S Reetz, N Baker, and R Tonjes. Survey of Context Provisioning Middleware. *Communications Surveys & Tutorials, IEEE*, 15(3):1492–1519, 2013. Cited in sections 2.4.1, 2.4.4, 2.6.4, 2.6, 2.6.4, 3, 3.3.3, 3.9, 3.9.2, 3.3, 3.9.3, 3.9.4, 3.9.5, 3.9.6, 4, 4.5, 4.5.1, 4.5.2, 4.5.9, 4.5.16, 4.3, 5.1.2, 8.3.2, and 8.5.

[KKS05] Swaroop Kalasapur, Mohan Kumar, and Behrooz Shirazi. Seamless service composition (SeSCo) in pervasive environments. In *MSC '05: Proceedings of the first ACM international workshop on Multimedia service composition.* ACM Request Permissions, November 2005. Cited in section 4.5.4.

[KLL$^+$97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing.* ACM Request Permissions, May 1997. Cited in section 6.4.6.

[KNSN05] W Kastner, G Neugschwandtner, S Soucek, and H M Newmann. Communication Systems for Building Automation and Control. *Proceedings of the IEEE*, 93(6):1178–1203, July 2005. Cited in sections 2.4.2, 3.2.1, 3.2.2, 3.3, 3.3.1, 3.3.4, and 4.4.2.

[KOA$^+$99] Cory D Kidd, Robert Orr, Gregory D Abowd, Christopher G Atkeson, Irfan A Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E Starner, and Wendy Newstetter. The Aware Home: A Living Laboratory for Ubiquitous Computing Research. In *Cooperative buildings. Integrating Information, Organizations, and Architecture*, pages 191–198. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. Cited in section 3.2.2.

[Kom00] Nikolas Kompridis. So We Need Something Else for Reason to Mean. *International Journal of Philosophical Studies*, 8(3):271–295, January 2000. Cited in section 3.6.2.

[KR90] Leslie Pack Kaelbling and Stanley J Rosenschein. Action and planning in embedded agents. *Robotics and Autonomous Systems*, 6(1-2), June 1990. Cited in sections 3.6.1, 3.6.4, and 3.6.5.

[KRM$^+$07] Kari Kostainen, Olli Rantapuska, Seamus Moloney, Virpi Roto, Ursula Holmström, and Kristiina Karvonen. Usable Access Control inside Home Networks. Technical Report NRC-TR-2007-009, Nokia Research Center, Helsinki, April 2007. Cited in section 7.

[Kru09] John Krumm. *Ubiquitous computing fundamentals.* Chapman & Hall/CRC, September 2009. Cited in sections 2.4.1, 2.4.4, 2.5.1, 2.5.1, 2.5.3, 3, 3.1, 3.2, 3.10.9, 4, 4.5.1, and 6.4.

[KS94] P Kalyanasundaram and A S Sethi. Interoperability issues in heterogeneous network management - Springer. *Journal of Network and Systems Management*, 1994. Cited in sections 4.4.1 and 4.4.1.

[KSD$^+$03] M Kumar, B Shirazi, S K Das, B Y Sung, D Levine, and M Singhal. PICO: a middleware framework for pervasive computing. *Pervasive Computing, IEEE*, 2(3):72–79, 2003. Cited in section 4.5.4.

[KSSR05] T Kindberg, B Schiele, A Schmidt, and K Rehman. What makes for good application- led research in ubiquitous computing? . *Pervasive 2005 Workshop*, May 2005. Cited in section 2.4.4.

[Kub68] Stanley Kubrick. 2001: A Space Odyssey. Technical report, Stanley Kubrick, GB/USA, 1968. Cited in section 5.5.6.

[KY12] Yung-Wei Kao and Shyan-Ming Yuan. User-configurable semantic home automation. *Computer Standards & Interfaces*, 34(1):171–188, January 2012. Cited in section 3.2.2.

[Lac09] Marc Lacoste. *Architecting Adaptable Security Infrastructures for Pervasive Networks through Components* , volume 56 of *Communications in Computer and Information Science1865-09291865-0937.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. Cited in section 7.

[Lam88] S S Lam. Protocol conversion. *IEEE Transactions on Software Engineering*, 14(3):353–362, March 1988. Cited in section 3.3.4.

[Lan64] P J Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 1964. Cited in section 3.9.6.

[Lan01] Marc Langheinrich. Privacy by Design - Principles of Privacy-Aware Ubiquitous Systems. In *UbiComp '01: Proceedings of the 3rd international conference on Ubiquitous Computing.* Springer-Verlag, September 2001. Cited in sections 5.5 and 5.5.6.

[Lee06] EA Lee. Cyber-physical systems-are computing foundations adequate. *NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, 2006. Cited in sections 2.6.2, 3.2.5, 3.3.4, and 3.7.4.

[LM05] Chang-Eun Lee and Kyeong-Deok Moon. Design of a universal middleware bridge for device interoperability in heterogeneous home network middleware. In *Consumer Electronics, 2005. ICCE. 2005 Digest of Technical Papers. International Conference on*, pages 371–372, 2005. Cited in section 3.3.4.

[LST13] V Ludovici, F Smith, and F Taglino. Collaborative ontology building in virtual innovation factories. In *Collaboration Technologies and Systems (CTS), 2013 International Conference on*, pages 443–450, 2013. Cited in sections 3.3.1, 3.9.6, 5, 5.1.2, and 7.

[LT03] Josh Lerner and Jean Tirole. Some Simple Economics of Open Source. *The Journal of Industrial Economics*, 50(2):197–234, March 2003. Cited in section 4.2.

[LTM$^+$11] F Liu, J Tong, J Mao, R Bohn, John Messina, Lee Badger, and Dawn Leaf. NIST Cloud Computing Reference Architecture. *NIST Special Publication*, 2011. Cited in section 4.3.

[LTW01] J Liu, K C Tsui, and J Wu. Introducing autonomy oriented computation. *Proceedings of First International Workshop on Autonomy Oriented Computation*, 2001. Cited in section 3.7.

[LY02a] Kalle Lyytinen and Youngjin Yoo. Issues and Challenges in Ubiquitous Computing . *Communications of the ACM*, 45(12), December 2002. Cited in sections 2.1.1, 2.2, and 2.3.

[LY02b] Kalle Lyytinen and Youngjin Yoo. Research Commentary: The Next Wave of Nomadic Computing. *Information Systems Research*, 13(4):377–388, December 2002. Cited in sections 2.3 and 2.4.

[LY10] J Laugesen and Yufei Yuan. What Factors Contributed to the Success of Apple's iPhone? In *Mobile Business and 2010 Ninth Global Mobility Roundtable (ICMB-GMR), 2010 Ninth International Conference on*, pages 91–99. McMaster University, DeGroote School of Business Hamilton, Ontario CANADA, 2010. Cited in sections 3.10, 3.10.4, 3.10.5, and 3.10.6.

[MALP12] Zeinab Movahedi, Mouna Ayari, Rami Langar, and Guy Pujolle. A Survey of Autonomic Network Architectures and Evaluation Criteria. *IEEE Communications Surveys & Tutorials*, 14(2):464–490, September 2012. Cited in sections 3.7.2, 3.7.4, and 6.4.10.

[Mar10] P A Martin. Collaborative ontology modelling collaboratively built, evaluated and distributed ontologies. In *Intelligent Computer Communication and Processing (ICCP), 2010 IEEE International Conference on*, pages 59–66, 2010. Cited in section 3.3.4.

[Mat10] Seiichiro Matsumoto. Echonet: A Home Network Standard. *IEEE Pervasive Computing*, 9(3):88–92, September 2010. Cited in section 3.2.2.

[MBdC$^+$06] F Mancinelli, J Boender, R di Cosmo, J Vouillon, B Durak, X Leroy, and R Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *Automated Software*

Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on, pages 199–208, 2006. Cited in sections 4.2 and 7.2.5.

[McC58] J McCarthy. Programs with common sense. *National Physical Laboratory Teddington*, 1958. Cited in section 3.6.1.

[MG10] Victor Matos and Rebecca Grasser. Building applications for the Android OS mobile platform: a primer and course materials. *Journal of Computing Sciences in Colleges*, 26(1):23–29, October 2010. Cited in sections 3.10, 3.10.3, and 3.10.5.

[MH05] M Mernik and J Heering. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 2005. Cited in section 3.3.1.

[MH12] Sarah Mennicken and Elaine M Huang. Hacking the natural habitat: an in-the-wild study of smart homes, their development, and the people who live in them. In *Pervasive'12: Proceedings of the 10th international conference on Pervasive Computing*. Springer-Verlag, June 2012. Cited in sections 2.1, 2.4.4, 3.2, 3.2.2, 3.2.3, 3.2.5, 3.3, 3.3.3, 3.3.4, 4.2, 6.4, 7.2, and 7.2.6.

[MHDM09] D Massaguer, B Hore, M Diallo, and S Mehrotra. Middleware for Pervasive Spaces: Balancing Privacy and Utility . *Middleware 2009*, 2009. Cited in section 3.10.8.

[MHH09] Hermann Merz, Thomas Hansemann, and Christof Hübner. The Basics of Industrial Communication Technology. In *Building Automation*, pages 27–48. Springer Berlin Heidelberg, Berlin, Heidelberg, January 2009. Cited in sections 3.2.1, 3.2.2, and 4.4.2.

[MHP00] Brad Myers, Scott E Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *Transactions on Computer-Human Interaction (TOCHI*, 7(1), March 2000. Cited in section 6.3.3.

[Min74] Marvin Minsky. A Framework for Representing Knowledge. *The Psychology of Computer Vision*, pages 211–277, 1974. Cited in section 3.5.2.

[MN90] Rolf Molich and Jakob Nielsen. Improving a human-computer dialogue. *Communications of the ACM*, 33(3):338–348, March 1990. Cited in section 8.3.5.

[MPR01] A L Murphy, G P Picco, and G C Distributed Computing Systems 2001 21st International Conference on Roman. LIME: a middleware for physical and logical mobility. In *Distributed Computing Systems, 2001. 21st International Conference on*, 2001. Cited in section 3.4.2.

[MRA10] Vittorio Miori, Dario Russo, and Massimo Aliberti. Domotic technologies incompatibility becomes user transparent. *Communications of the ACM*, 53(1):153, January 2010. Cited in sections 3.2.1, 3.2.2, 3.3, 3.3.1, and 3.3.3.

[MS72] A R Meyer and L J Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *SWAT '72: Proceedings of the 13th Annual Symposium on Switching and Automata Theory (swat 1972*, pages 125–129. IEEE Computer Society, October 1972. Cited in section 5.2.14.

[MTC+13] Carolina Milanesi, Lillian Tay, Roberta Cozza, Ranjit Atwal, Tuong Huy Nguyen, Tracy Tsai, Annette Zimmermann, and C K Lu. Forecast: Devices by Operating System and User Type, Worldwide, 2010-2017. Technical report, June 2013. Cited in sections 2.3.2 and 3.10.

[MTMT06] V Miori, L Tarrini, M Manca, and G Tolomei. An open standard solution for domotic interoperability. *IEEE Transactions on Consumer Electronics*, 52(1):97–103, February 2006. Cited in sections 3.2.1, 3.2.2, and 3.3.3.

[MW12] Mary Meeker and Liang Wu. 2012 Internet Trends. Technical report, May 2012. Cited in section 2.3.2.

[ND00] R J C Nunes and J C M Delgado. Internet Application for Home Automation. In *10th Mediterranean Electrotechnical Conference. Information Technology and Electrotechnology for the Mediterranean Countries. Proceedings. MeleCon 2000*, pages 298–301. IEEE, September 2000. Cited in section 3.2.2.

[Nil90] Nils J Nilsson. Logic and artificial intelligence. *Artificial intelligence*, (47):31–56, July 1990. Cited in sections 3.6.1 and 3.6.

[Nil98] Nils Nilsson. *Artificial Intelligence.* a new synthesis. Morgan Kaufmann Pub, 1998. Cited in sections 3.4.2, 3.6, and 3.6.1.

[NKZ10] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints . In *the 5th ACM Symposium*, page 328, New York, New York, USA, 2010. ACM Press. Cited in section 3.10.8.

[NMMA13] Natalya F Noy, Jonathan Mortensen, Mark A Musen, and Paul R Alexander. Mechanical turk as an ontology engineer?: using microtasks as a component of an ontology-engineering workflow. In *WebSci '13: Proceedings of the 5th Annual ACM Web Science Conference.* ACM Request Permissions, May 2013. Cited in sections 4.2, 7.4, and 7.4.2.

[NPD+07] A.E Nikolaidis, S Papastefanos, G.A Doumenis, G.I Stassinopoulos, and M.P.K Drakos. Local and remote management integration for flexible service provisioning to the home. *Communications Magazine, IEEE*, 45(10):130–138, 2007. Cited in section 3.3.4.

[NSS58] A Newell, J C Shaw, and H A Simon. Elements of a theory of human problem solving. *Psychological review*, 1958. Cited in sections 3.5.4 and 3.6.1.

[NSY08] Chuzo Ninagawa, Tomotaka Sato, and Yahiko. Communication Performance Simulation for Object Access of BACnet Web Service Building Facility Monitoring Systems . In *Factory Automation (ETFA 2008)*, pages 701–704. IEEE, September 2008. Cited in section 3.2.1.

[Nyq24] Harry Nyquist. Certain Factors Affecting Telegraph Speed. *Nidwinter Convention of the A. I. E. E.*, 1924. Cited in sections 3.3.1 and 8.3.2.

[OAS13] OASIS. oBIX Version 1.1, July 2013. Cited in section 3.2.1.

[Ohl12] Stellan Ohlsson. The Problems with Problem Solving: Reflections on the Rise, Current Status, and Possible Future of a Cognitive Research Paradigm. *The Journal of Problem Solving*, 5(1), October 2012. Cited in sections 3.5.4 and 3.5.4.

[Oku86] K Okumura. A formal protocol conversion method. *ACM SIGCOMM Computer Communication Review*, 1986. Cited in section 3.3.4.

[Opp02] Reinhard Oppermann. User-interface Design. In *Handbook on information technologies for education and training*, pages 233–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. Cited in section 8.3.5.

[OPSS94] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus: an architecture for extensible distributed systems. In *ACM SIGOPS Operating Systems Review*, pages 58–68. ACM, 1994. Cited in section 3.4.2.

[oT01] of Standards, National Institute and Technology. Advanced Encryption Standard. *NIST FIPS PUB 197*, 2001. Cited in section 6.5.

[Pah09] Marc-Oliver Pahl. labsystem.sourceforge.net [online]. October 2009. URL: http://labsystem.sourceforge.net/. Cited in sections 9.6.2 and A.3.

[Pah14a] Marc-Oliver Pahl. Smart Space Orchestration Course Material [online]. January 2014. URL: http://pahl.de/download/dissertation/ds2os.lab/. Cited in sections 9.6.1, 9.6.4, 9.6.5, and A.3.

[Pah14b] Marc-Oliver Pahl. Smart Space Orchestration Course Material ePub [online]. January 2014. URL: http://pahl.de/download/dissertation/ds2os.lab/SmartSpaceOrchestration.epub. Cited in sections 9.6.1, 9.6.4, 9.6.5, and A.3.

[Pap05] Seymour Papert. Teaching Children Thinking. *Contemporary Issues in Technology and Teacher Education*, 5(3):353–365, 2005. Cited in sections 3.5.5, 5.6.7, 6.3.3, and 8.3.3.

[Par72] D L Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 1972. Cited in section <R.26>.

[PBSZ13] Martin Peters, Christopher Brink, Sabine Sachweh, and Albert Zündorf. Performance Considerations in Ontology Based Ambient Intelligence Architectures. In *Ambient Intelligence - Software and Applications*, pages 121–128. Springer International Publishing, Heidelberg, 2013. Cited in section 9.3.1.

[PD11] Larry L Peterson and Bruce S Davie. *Computer Networks.* A Systems Approach. Elsevier, March 2011. Cited in sections 3.3.1 and 3.3.4.

[Per82] Alan J Perlis. Special Feature: Epigrams on programming. *SIGPLAN Notices*, 17(9), September 1982. Cited in sections 1, 2, 3, 5, 5.3, 7.4, 8, 8.3.2, 9, 10.1, and 10.3.

[PF04] Siobhan Clarke Patrick Fahy. CASS – a middleware for mobile context-aware applications. *Workshop on Context Awareness*, 2004. Cited in section 4.5.2.

[PK10] Kolin Paul and Tapas Kumar Kundu. Android on Mobile Devices: An Energy Perspective. In *CIT '10: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology.* IEEE Computer Society, June 2010. Cited in section 3.10.1.

[PMC$^+$12] Andreas P, Matos, Christina, Andreas C, Vanessa, Michael, and Stijn. Developer Economics 2012. Technical report, London, June 2012. Cited in sections 2.3.3, 3.10.3, 3.10.6, 3.10.7, and 4.2.

[PNKC13] Marc-Oliver Pahl, Heiko Niedermayer, Holger Kinkelin, and Georg Carle. Enabling Sustainable Smart Neighborhoods. In *3rd IFIP Conference on Sustainable Internet and ICT for Sustainability 2013 (SustainIT 2013)*, Palermo, Italy, 2013. Cited in sections 2.4.4, 3.2.1, 8.3.4, and 10.2.

[Pos11] Stefan Poslad. *Ubiquitous Computing.* Smart Devices, Environments and Interactions. Wiley, August 2011. Cited in sections 3, 3.3.1, 3.3.4, 3.9.1, 4, 4.5.1, and 6.4.

[PPB$^+$04] R W Picard, S Papert, W Bender, B Blumberg, C Breazeal, D Cavallo, T Machover, M Resnick, D Roy, and C Strohecker. Affective Learning — A Manifesto. *BT Technology Journal*, 22(4), October 2004. Cited in sections 5.6.7 and 8.3.3.

[Pri97] Wolfgang Prinz. Perception and Action Planning. *European Journal of Cognitive Psychology*, 9(2):129–154, June 1997. Cited in section 3.5.4.

[PRL10] T Perumal, AR Ramli, and CY Leong. Middleware for heterogeneous subsystems interoperability in intelligent buildings. *Automation in Construction*, 2010. Cited in section 3.2.1.

[PS03] A Pras and J Schoenwaelder. On the Difference between Information Models and Data Models. Technical report, Internet Engineering Task Force, 2003. Cited in section 4.4.1.

[PTDL07] Michael P Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007. Cited in section 3.8.1.

[RC96] Gary S Robinson and Carl Cargill. History and Impact of Computer Standards. *Computer*, 29(10):79–85, October 1996. Cited in section 3.3.2.

[RC00] Manuel Román and Roy H Campbell. Gaia. In *the 9th workshop*, page 229, New York, New York, USA, 2000. ACM Press. Cited in section 4.5.3.

[RCKZ12] Vaskar Raychoudhury, Jiannong Cao, Mohan Kumar, and Daqiang Zhang. Middleware for pervasive computing: A survey. *Pervasive and Mobile Computing*, September 2012. Cited in sections 2.4.1, 3, 4, 4.5.1, 4.5.2, 4.5.16, 4.3, and 8.3.6.

[RCL09] B P Rimal, Eunmi Choi, and I Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51, 2009. Cited in section 4.3.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM Request Permissions, August 2001. Cited in section 6.4.6.

[RHC$^+$02] M Roman, C Hess, R Cerqueira, A Ranganathan, R H Campbell, and K Nahrstedt. A middleware infrastructure for active spaces. *Pervasive Computing, IEEE*, 1(4):74–83, 2002. Cited in section 4.5.3.

[RI10] R Rajkumar and Insup. Cyber-physical systems: The next computing revolution. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 731–736, 2010. Cited in sections 2.6.2 and 3.7.4.

[RK90] Stanley J Rosenschein and Leslie P Kaelbling. The Synthesis of Intelligent Real-Time Systems. November 1990. Cited in section 3.6.1.

[RKC01] Manuel Román, Fabio Kon, and Roy H Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001. Cited in section 4.5.3.

[RLT11] P E Rovsing, P G Larsen, and T S Toftegaard. A reality check on home automation technologies. *Journal of Green Engineering*, 2011. Cited in sections 3.2.1 and 3.2.2.

[RM90] M T Rose and K McCloghrie. Structure and identification of management information for TCP/IP-based internets. Technical report, Internet Engineering Task Force, 1990. Cited in section 4.4.1.

[RN95] Stuart J Russell and Peter Norvig. A modern approach. *Prentice Hall, New Jersey*, 1995. Cited in sections 3.3.4, 3.5.1, 3.5.3, 3.6, and 3.9.1.

[Rom02] Gaia: A Middleware Infrastructure to Enable Active Spaces. 2002. Cited in section 4.5.3.

[RR81] Richard F Rashid and George G Robertson. Accent: A communication oriented network operating system kernel. *SOSP*, 1981. Cited in sections 3.4.4, 6.2, and 7.2.4.

[RS00] L Rosenthal and V Stanford. NIST Smart Space: pervasive computing initiative. *Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2000. Cited in sections 3.3.2 and 3.3.3.

[RSA78] R L Rivest, A Shamir, and L Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), February 1978. Cited in section 6.5.

[RvdM13] Janessa Rivera and Rob van der Meulen. Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013 [online]. April 2013. URL: http://www.gartner.com/newsroom/id/2408515. Cited in section 2.3.2.

[SA97] B Segall and D Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG97*, 1997. Cited in section 4.5.11.

[SA11] P Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. Technical report, Internet Engineering Task Force, 2011. Cited in section 6.3.3.

[Sat01] M Satyanarayanan. Pervasive computing: Vision and challenges. *Personal Communications*, 2001. Cited in sections 2.1.1, 2.2, 2.4, and 3.3.3.

[SAW94] B Schilit, N Adams, and R Want. Context-Aware Computing Applications. *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 85–90, 1994. Cited in section 2.6.4.

[SBK$^+$11] James Scott, A J Bernheim Brush, John Krumm, Brian Meyers, Michael Hazas, Stephen Hodges, and Nicolas Villar. PreHeat: controlling home heating using occupancy prediction. In *UbiComp '11: Proceedings of the 13th international conference on Ubiquitous computing*. ACM Request Permissions, September 2011. Cited in section 3.2.3.

[Sch08] J Schonwalder. Protocol-Independent Data Modeling: Lessons Learned from the SMIng Project. *IEEE Communications Magazine*, 46(5):148–153, September 2008. Cited in section 4.4.1.

[SFK$^+$10] A Shabtai, Y Fledel, U Kanonov, Y Elovici, S Dolev, and C Glezer. Google Android: A Comprehensive Security Assessment. *IEEE Security & Privacy Magazine*, 8(2):35–44, 2010. Cited in sections 3.10.7 and 3.10.8.

[SG02] J P Sousa and David Garlan. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. 2002. Cited in section 4.5.4.

[SG03] Joao P Sousa and David Garlan. The Aura Software Architecture: an Infrastructure for Ubiquitous Computing. August 2003. Cited in section 4.5.4.

[Sha48] Claude Elwood Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27, July 1948. Cited in section 8.3.2.

[Sha01] Claude Elwood Shannon. A mathematical theory of communication. *SIGMOBILE Mobile Computing and Communications Review*, 2001. Cited in section 3.3.1.

[SHB10] Gregor Schiele, Marcus Handte, and Christian Becker. Pervasive Computing Middleware. In *Handbook of Ambient Intelligence and Smart Environments*, pages 201–227. Springer US, Boston, MA, 2010. Cited in sections 2.4.1, 3, 3.3, 3.3.1, 4, and 4.5.1.

[Sho78] John F Shoch. Inter-Network Naming, Addressing, and Routing. In *Seventeenth IEEE Conference on Computer Communication Networks*, pages 72–79, Washington, D.C, 1978. Cited in section 5.2.13.

[SIE13] SIEMENS AG. Communication in building automation. Siemens Switzerland Ltd Infrastructure & Cities Sector Building Technologies Division International Headquarters Gubelstrasse 22 6301 Zug Switzerland, 2013. Cited in section 3.2.1.

[SJR11] P Sánchez, M Jiménez, and F Rosique. A framework for developing home automation systems: From requirements to code . *The Journal of Systems and Software*, 2011. Cited in section 3.2.1.

[SL89] J C Shu and M T Liu. A Synchronization Model for Protocol Conversion. In *IEEE INFOCOM '89, Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 276–284 vol.1. IEEE, September 1989. Cited in section 3.3.4.

[SLP04] T Strang and C Linnhoff-Popien. A Context Modeling Survey. *Workshop Proceedings*, 2004. Cited in sections 3.3.4, 3.9, 3.9.2, 3.3, 3.9.3, 3.9.5, and 3.9.6.

[SM03] D Saha and A Mukherjee. Pervasive computing: a paradigm for the 21st century. *Computer*, 36(3):25–31, March 2003. Cited in sections 2.1.1, 2.2, 2.4, 2.4, 2.4.1, and 2.4.4.

[SM10] Jose Simoes and Thomas Magedanz. Smart advertising in the home of the future. *International Journal of Computer Aided Engineering and Technology*, 2(2/3):164, 2010. Cited in section 2.4.4.

[SM13] C Stach and B Mitschang. Privacy Management for Mobile Platforms — A Review of Concepts and Approaches. *Mobile Data Management (MDM)*, 2013. Cited in section 3.10.8.

[Sno03] D Snoonian. Smart buildings. *IEEE Spectrum*, 40(8):18–23, August 2003. Cited in section 4.4.2.

[SP05] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface.* Pearson Education, Inc, 4 edition, 2005. Cited in section 9.3.1.

[Spi03] Diomidis D Spinellis. The information furnace: consolidated home control. *Personal and ubiquitous computing*, 7(1), May 2003. Cited in section 3.2.2.

[SRN$^+$08] F Sandu, M Romanca, A Nedelcu, P Borza, and R Dimova. Remote and Mobile Control in Domotics . In *2008 11th International Conference on Optimization of Electrical and Electronic Equipment (OPTIM)*, pages 225–228. IEEE, September 2008. Cited in section 3.2.3.

[SRT00] S Soucek, G Russ, and C Tamarit. The smart kitchen project–an application of fieldbus technology to domotics. In *Proceedings of the 2nd International Workshop on Networked Appliances*, 2000. Cited in section 3.2.2.

[SRV08] R Sanchez, L Raptis, and K Vaxevanakis. Ethernet as a carrier grade technology: developments and innovations. *IEEE Commun Magazine*, 46(9):88–94, September 2008. Cited in section 2.2.1.

[ST94] B N Schilit and M M Theimer. Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, 1994. Cited in section 2.4.1.

[Str04] John Strassner. *Policy-Based Network Management.* Solutions for the Next Generation. Morgan Kaufmann, 2004. Cited in section 3.7.5.

[SWG13] M Shiraz, M Whaiduzzaman, and A Gani. A Study on Anatomy of Smartphone. *Computer Communication & Collaboration*, 2013. Cited in section 3.10.1.

[SWS$^+$04] Carl-Fredrik Sørensen, Maomao Wu, Thirunavukkarasu Sivaharan, Gordon S Blair, Paul Okanda, Adrian Friday, and Hector Duran-Limon. A context-aware middleware for applications in mobile Ad Hoc environments. In *MPAC '04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing.* ACM, October 2004. Cited in section 4.5.6.

[SWS07] Anoop Singhal, Theodore Winograd, and Karen A Scarfone. Guide to Secure Web Services. *Guide to Secure Web Services*, August 2007. Cited in section 3.8.

[SWW$^+$02] W Stark, Hua Wang, A Worthen, S Lafortune, and D Teneketzis. Low-energy wireless communication network design. *IEEE Wireless Communications*, 9(4):60–72, August 2002. Cited in sections 3.3.1 and 3.3.4.

[SZWN11] N Schultz, R Zarnekow, J Wulf, and Quoc-Tuan Nguyen. The new role of developers in the mobile ecosystem: An Apple and Google case study. In *Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference on*, pages 103–108, 2011. Cited in sections 3.10.4 and 3.10.7.

[Tan01] Andrew S Tanenbaum. Modern Operating Systems, 2nd edition. *Modern Operating Systems, 2nd edition*, February 2001. Cited in sections 3.4.4, 3.10.3, 5.6.2, and 6.3.1.

[Tan04] A S Tanenbaum. Distributed Systems, 2004. Cited in sections 2.2, 2.4, 3.4.1, 3.2, and 3.4.2.

[TBEJ08] Ling Tai, Ron Baker, Elizabeth Edmiston, and Ben Jeffcoat. IBM Tivoli Common Data Model: Guide to Best Practices. *IBM Redbook*, 2008. Cited in section 4.4.1.

[TH10] Chia-Chi Teng and R Helps. Mobile Application Development: Essential New Directions for IT. In *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*, pages 471–475, 2010. Cited in sections 3.10 and 3.10.5.

[The12] The OSGI Alliance. *OSGi Service Platform Core Specification.* Release 4, Version 4.3. The OSGi Alliance, April 2012. Cited in sections 7.2.1 and 7.2.2.

[TK10] Friedbert Tiersch and Christian Kuhles. BACnet and BACnet, IP. Desotron Verlag Erfurt, 2010. Cited in sections 3.2.1 and 4.4.2.

[TKDHC13] J Taneja, A Krioukov, S Dawson-Haggerty, and D E Culler. Enabling Advanced Environmental Conditioning with a Building Application Stack. *eecs.berkeley.edu*, 2013. Cited in sections 3.2.1, 3.2.2, and 4.5.7.

[TMK+03] E Topalis, L Mandalos, S Koubias, G Papadopoulos, and I Nikiforakis. A novel architecture for remote home automation e-services on an OSGi platform via high-speed internet connection ensuring QoS support by using RSVP technology. *IEEE Transactions on Consumer Electronics*, 48(4):825–833, November 2003. Cited in section 3.2.2.

[TMKP02] Ulrich Topp, Peter Müller, Jens Konnertz, and Andreas Pick. Web Based Service for Embedded Devices. *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, October 2002. Cited in section 3.2.1.

[TPR+12] Leila Takayama, Caroline Pantofaru, David Robson, Bianca Soto, and Michael Barry. Making technology homey: finding sources of satisfaction and meaning in home automation. In *UbiComp '12: Proceedings of the 2012 ACM Conference on Ubiquitous Computing*. ACM Request Permissions, September 2012. Cited in sections 2.4.4, 3.2.3, 3.3.3, 5.5, 5.5.6, 6.4, 7.2, and 7.2.6.

[TTP11] V K Tuunainen, T Tuunanen, and J Piispanen. Mobile Service Platforms: Comparing Nokia OVI and Apple App Store with the IISIn Model. In *Mobile Business (ICMB), 2011 Tenth International Conference on*, pages 74–83, 2011. Cited in sections 3.10, 3.10, 3.10.4, and 3.10.5.

[TVS02] Andrew S Tanenbaum and Maarten Van Steen. *Distributed Systems Principles and Paradigms*. Prentice Hall, July 2002. Cited in sections 3.4, 3.4.2, 3.4.3, 3.4.4, 3.7.4, and 7.2.4.

[TVS06] Andrew S Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2 edition, October 2006. Cited in sections 3.4.2, 4.3, 4.4.1, 4.4.2, 5.3.2, and 5.6.3.

[TWDT13] Joshua G Tanenbaum, Amanda M Williams, Audrey Desjardins, and Karen Tanenbaum. Democratizing technology: pleasure, utility and expressiveness in DIY and maker practice. In *CHI '13: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Request Permissions, April 2013. Cited in sections 3.2.3 and 4.2.

[VD10] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP*. The Next Internet. Morgan Kaufmann, July 2010. Cited in sections 2.1, 2.4, 3.2, 3.2.1, and 3.2.2.

[vdMR13] Rob van der Meulen and Janessa Rivera. Gartner Says Smartphone Sales Grew 46.5 Percent in Second Quarter of 2013 and Exceeded Feature Phone Sales for First Time [online]. August 2013. Cited in section 2.3.2.

[vE13] Birgit van Eimeren. âAlways on" – Smartphone, Tablet & Co. als neue Taktgeber im Netz . *Media Perspektiven 7-8/2013*, pages 386–390, August 2013. Cited in sections 2.3.1, 2.3.2, 2.5.1, 3.10, 7.6.4, and 7.8.

[vEF12] Birgit van Eimeren and Beate Fress. 76 Prozent der Deutschen online – neue Nutzungssituationen durch mobile Endgeräte. *Media Perspektiven*, 7-8(ohne Seitennummern), 2012. Cited in sections 2.3.1, 2.3.2, and 3.10.4.

[VSCK11] Michael Vakulenko, Stijn Schuermans, Andreas Constantinou, and Matos Kapetanakis. Mobile Platforms: The Clash of Ecosystems. Technical report, November 2011. Cited in sections 2.3.2, 2.3.3, 3.10, 3.10.1, 3.10.3, 3.10.3, 3.10.3, 3.10.4, 3.10.5, and 7.2.1.

[VVE09] Toby Velte, Anthony Velte, and Robert Elsenpeter. *Cloud Computing, A Practical Approach.* McGraw Hill Professional, September 2009. Cited in sections 3.8, 3.8.2, and 4.3.

[Wan09] R Want. When Cell Phones Become Computers. *Pervasive Computing, IEEE*, 8(2):2–5, 2009. Cited in sections 2.3.2, 3.2.4, and 3.10.1.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, September 1991. Cited in sections 1, 2.1, 2.1, 2.1, 2.3.4, 2.4.3, 2.4.4, 2.4.5, and 2.5.3.

[Wei93] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7), July 1993. Cited in sections 2.1 and 2.1.

[WHI97] K N WHITLEY. Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages & Computing*, 8(1):109–142, February 1997. Cited in sections 3.5.5 and 6.3.3.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4), April 1971. Cited in section 3.1.1.

[WK12] Thomas Williams and Colin Kellay. *gnuplot 4.7*, 2012. Cited in section 9.3.3.

[WL05] JKW Wong and H Li. Intelligent building research: a review. *Automation in Construction*, 2005. Cited in sections 2.4.4, 3.2.1, and 3.2.4.

[WM10] Joel West and Michael Mace. Browsing as the killer app: Explaining the rapid success of Apple's iPhone. *Telecommunications Policy*, 34(5-6):270–286, June 2010. Cited in sections 2.3.1, 2.3.2, 2.3.3, 3.10, and 3.10.2.

[WN07] G Wood and M Newborough. Energy-use information transfer for intelligent homes: Enabling energy conservation with central and local displays. *Energy and Buildings*, 39(4):495–503, April 2007. Cited in section 2.4.4.

[Woo09] M J Wooldridge. An Introduction to MultiAgent Systems, 2009. Cited in sections 2.4, 3.5.3, 3.5.4, 3.6.3, 3.6.4, 3.6.5, 5.5.6, and 5.5.6.

[WOW08] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe: the least-authority filesystem. In *StorageSS '08: Proceedings of the 4th ACM international workshop on Storage security and survivability.* ACM Request Permissions, October 2008. Cited in section 6.5.5.

[WS97] A C W Wong and A T P So. Building Automation In The 2lst Century . In *APSCOM-97. International Conference on Advances in Power System Control, Operation and Management*, pages 819–824. IEE, November 1997. Cited in sections 3.2.1 and 3.3.1.

[YLY05] Y Yoo, K Lyytinen, and H Yang. The role of standards in innovation and diffusion of broadband mobile services: The case of South Korea. *The Journal of Strategic Information Systems*, 2005. Cited in section 3.3.2.

[YYC10] Young Seog Yoon, Jaeheung Yoo, and Munkee Choi. Revenue sharing is the optimal contractual form for emerging app economy? In *Information and Communication Technology Convergence (ICTC), 2010 International Conference on*, pages 329–334, 2010. Cited in section 3.10.6.

[YZYN08] L Yan, Y Zhang, L T Yang, and H Ning. The Internet of Things. 2008. Cited in sections 2.1, 2.4.1, 2.6.1, and 2.6.4.

[Zim80] H Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *Communications, IEEE Transactions on*, 28(4):425–432, 1980. Cited in sections 3.3.4, 3.3, and 3.3.4.

[ZM03] Zuo Zhihong and Zhou Mingtian. Web Ontology Language OWL and its description logic foundation. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 157–160, 2003. Cited in section 3.9.6.

[ZPOW11] J Zhu, H K Pung, M Oliya, and Wai Choong Wong. A context realization framework for ubiquitous applications with runtime support. *Communications Magazine, IEEE*, 49(9):132–141, 2011. Cited in section 4.5.14.

Index

Acronyms

AAL Ambient Assisted Living.

ACID Atomic Consistent Isolated Durable.

AES Advanced Encryption Standard.

AI Artificial Intelligence.

AmI Ambient Intelligence.

ANSI American National Standards Institute.

API Application Programming Interface.

AR Augmented Reality.

ARS Advanced Reasoning Service.

ASHRAE American Society of Heating, Refrigerating and Air Conditioning Engineers.

ASN.1 Abstract Syntax Notation One.

BACnet Building Automation Control network.

BAS Building Automation System.

BDI Believe-Desire-Intention.

BGP Border Gateway Protocol.

BMS Building Management System.

BPEL Business Process Execution Language.

CA Certification Authority.

CAS Complex Adaptive System.

CCMS Central Control and Monitoring Systems.

CIM Common Information Model.

CISC Complex Instruction Set Computer.

CMR Context Model Repository.

CORBA Common Object Request Broker.

CPS Cyber-Physical System.

CPU Central Processing Unit.

CSCW Computer Supported Cooperative Work.

DALI Digital Addressable Lighting Interface.

DCOM Distributed Component Object Model.

DDC Direct Digital Control.

DFA Deterministic Finite Automaton.

DHT Distributed Hash Table.

DIY Do-It-Yourself.

DMI Desktop Management Interface.

DMTF Distributed Management Task Force.

DS2OS Distributed Smart Space Orchestration System.

DSL Domain-Specific Language.

ECA Event-Condition-Action.

EMS Energy Management System.

FMS Facility Management System.

FSM Finite State Machine.

FTP File Transfer Protocol.

GPS Global Positioning System.

GUI Graphical User Interface.

GUID Globally Unique IDentifier.

HDD Hard Disk Drive.

HTML Hyper Text Markup Language.

HTTP Hyper Text Transfer Protocol.

HTTPS Secure Hyper Text Transfer Protocol.

HVAC Heating, Ventilation, Air-Conditioning.

IBS Intelligent Building System.

ID Identifier.

IDE Integrated Development Environment.

IDL Interface Definition Language.

IEC International Electrotechnical Commission.

IEEE Institute of Electrical and Electronics Engineers.

IETF Internet Engineering Task Force.

IoT Internet of Things.

IP Internet Protocol.

IPC Inter Process Communication.

ISO International Organization for Standardization.

JADE Java Agent Development Environment.

JAR Java ARchive.

JDBC Java Database Connectivity.

JINI Jini Is Not Initials.

JSON JavaScript Object Notation.

JVM Java Virtual Machine.

KA Knowledge Agent.

KOR Knowledge Object Repository.

KVM Keyboard-Video-Monitor.

LAN Local Area Network.

LIME Linda In Mobile Environments.

LOC Lines Of Code.

LRU Least Recently Used.

LTE Long-Term Evolution.

M2M Machine-to-Machine.

MAN Metropolitan Area Network.

MAPE Monitor-Analyze-Plan-Execute.

MAPE-K Monitor-Analyze-Plan-Execute-Knowledge.

MIB Managed Information Base.

MIF Management Information Format.

MIT Massachusetts Institute of Technology.

MO Managed Object.

MOF Managed Object Format.

MOM Message Oriented Middleware.

NLSM	Node Local Service Manager.
NMS	Network Management System.
NTP	Network Time Protocol.
OASIS	Organization for the Advancement of Structured Information Standards.
oBIX	Open Building Information Exchange.
OOP	Object Oriented Programming.
OS	Operating System.
OSGI	Open System gateway Initiative.
OSI	Open Systems Interconnection.
OSS	Open Source Software.
OWL	Web Ontology Language.
P2P	Peer-to-Peer.
PAN	Personal Area Network.
PARC	Palo Alto Research Center.
PC	Personal Computer.
PLC	Programmable Logic Controller.
PLC	Power Line Communication.
POJO	Plain Old Java Object.
PUP	PARC Universal Protocol.
QoS	Quality of Service.
RAM	Random Access Memory.
RAS	Remote Access Service.
RDF	Resource Description Framework.
REST	Representational State Transfer.
RFID	Radio Frequency IDentification.
RISC	Reduced Instruction Set Computer.
RMI	Remote Method Invocation.
RPC	Remote Procedure Call.
RTE	Run Time Environment.
S2S	Smart Space Service management.

S2Store	Smart Space Store.
SCADA	Supervisory Control And Data Acquisition.
SDK	Software Development Kit.
SGML	Standard Generic Markup Language.
SHE	Service Hosting Environment.
SLCA	Site Local Certificate Authority.
SLSM	Site Local Service Manager.
SMTP	Simple Mail Transfer Protocol.
SNMP	Simple Network Management Protocol.
SOA	Service Oriented Architecture.
SOAP	Simple Object Access Protocol.
SOAP	Service Oriented Architecture Protocol.
SSH	Secure SHell.
TCP	Transmission Control Protocol.
TUM	Technische Universität München.
UDDI	Universal Description Discovery and Integration.
UDP	User Datagram Protocol.
UI	User Interface.
UML	Unified Modeling Language.
UMTS	Universal Mobile Telecommunications System.
UPNP	Universal Plug aNd Play.
URL	Unified Resource Locator.
VM	Virtual Machine.
VPN	Virtual Private Network.
VR	Virtual Reality.
VSL	Virtual State Layer.
W3C	World Wide Web Consortium.
WAN	Wide Area Network.
WAP	Wireless Application Protocol.
WBEM	Web-Based Enterprise Management.
WS	Web Service.

WSDL Web Service Description Language.

WWW World Wide Web.

XML Extensible Markup Language.

Glossary

μ-middleware A μ-middleware is a middleware that provides only basic, non domain-specific functionality, and that supports transparent extension with functionality via regular services at run time. μ-middleware enables overcoming middleware silos (see Sec. 4.5, Sec. 6.2).

adaptation service A services that provide software interfaces to Smart Devices are called *adaptation services* in this thesis. (see Sec. 8.3.1).

App The term App is the short form of application. It is typically used in the context of smartphones and the App economy. Apps for Smart Spaces are services that provide functionality for Smart Spaces. A Smart Space App provides functionality for Smart Space Orchestration (see Sec. 3.10).

App economy The term App economy describes the ecosystem that emerged around App stores for smartphones. The App economy comprises the App store, the users, and the developers (see Sec. 3.10).

App store The App store is the repository for smartphone applications that was introduced by Apple Inc. in 2008 (see Sec. 3.10). The equivalent of an App store for Smart Spaces is called Smart Space Store (S2Store) in this thesis (see Sec. 7.1.1).

context Context is information that is relevant for an entity to reach its goal. In this document, the term *context* is used to describe the virtual representation of information that is relevant for a service to implement Smart Space Orchestration (*virtual object* in Fig. 3.12). Such a *context* representation is typically structured by so-called context models (see Sec. 3.9, Sec. 2.4.1).

Context Model Repository The Context Model Repository (CMR) contains the ontology of all Virtual State Layer (VSL) Smart Spaces. An ontology is typically defined as triple ⟨subject, predicate, object⟩. Subject and object are context models identifiers in the Context Model Repository (CMR). The predicates that it supports are "derives" and "composes". The CMR contains the context models of all Distributed Smart Space Orchestration System (DS2OS) services (see Sec. 5.2.8).

crowdsourced software development The term *crowdsourced (software) development* is used to describe distributed user-based software development. In contrast to user-based development, crowdsourced development typically comprises competitive elements (see Sec. 3.10, Sec. 7.4).

Distributed Smart Space Orchestration System The DS2OS is introduced in this thesis. It provides the technology required for establishing an App economy for Smart Spaces. The core of DS2OS is the VSL (see Sec. 7).

knowledge The term *knowledge* is used to describe instances of context nodes in this work (see Sec. 5).

Knowledge Agent The VSL is implemented as a system of distributed peers. The peers are called Knowledge Agents (KAs). All KAs together span the VSL context overlay. Each DS2OS service connects to exactly one KA that gives access to the distributed knowledge that is stored in the context nodes of the overlay (see Sec. 6.3).

knowledge base The term knowledge base describes the conjunction of all knowledge trees of a VSL instance (see Sec. 5).

Knowledge Object Repository The VSL context repositories that contain the instances of VSL context nodes are also-called *Knowledge Object Repository (KOR)* or shorter *knowledge repository* in this thesis. Each KA has one Knowledge Object Repository (KOR) (see Sec. 6.3).

knowledge repository The KA context repository that contains the instances of context models from the CMR is called knowledge repository in this document (see Sec. 5).

knowledge tree All instances of context models that are stored in a VSL context repository are called *knowledge tree* in this work.

Node Local Service Manager The Node Local Service Manager (NLSM) is part of the Smart Space Service management (S2S) service management of DS2OS. The NLSM manages the DS2OS services on a computing node. To do so, the NLSM monitors and controls the status of each service using the Service Hosting Environment (SHE) interface (see Sec. 7.2.3).

ontology The term *ontology* originates in philosophy. It typically describes an explicit specification of a conceptualization [GN87, Gru93, AZW06]. In this work, the CMR contains an ontology for Smart Spaces (see Sec. 3.9.1, Sec. 5.2.8).

orchestration In this thesis, the term orchestration describes the execution of pervasive computing workflows to reach a certain goal. It includes the orchestration of services such as adaptation services. See Smart Space Orchestration.

orchestration service A service that implements the logic of apervasive computing scenarios is called *orchestration service* in this thesis (see Sec. 8.3).

regular node The term regular node is used for all context nodes that are stored in, and served by the context repositories of the KAs that span the VSL (see Sec. 5.3.2).

Run Time Environment The Run Time Environment (RTE) is part of the S2S service management of DS2OS. The RTE runs the Java services in DS2OS. In the prototype it uses . The RTE is interfaced by the SHE (see Sec. 7.2.1).

Service Hosting Environment The SHE is part of the S2S service management of DS2OS. The SHE provides the management interfaces of the RTE. The SHE is interfaced by the NLSM (see Sec. 7.2.2).

Site Local Certificate Authority The Site Local Certificate Authority (SLCA) is part of the S2S service management of DS2OS. The SLCA manages the certificates for services within a DS2OS site. Services certificates are the central element of the DS2OS security-by-design architecture. It issues and renews service certificates automatically. When a new service is installed in a site, it creates a new locally-signed certificate for the service using the certificate issued by the S2Store and the rights the user set over the Site Local Service Manager (SLSM) interface. When the certificate of a service is about to expire, the managing NLSM contacts the SLCA asking for a certificate renewal (see Sec. 7.3).

Site Local Service Manager The SLSM is part of the S2S service management of DS2OS. The SLSM manages all services within a DS2OS site. It offers the interfaces for users to install, start, and stop services. The SLSM delegates the task to run a service to the NLSMs that run on the distributed computing nodes (see Sec. 7.2.4).

Smart Device A Smart Device is an embedded system with pervasiveness support via sensors or actuators, and with networking support that enables remote control (see Sec. 2.4.1).

Smart Space Spaces that contain Smart Devices and support Smart Space Orchestration are called Smart Spaces (see Sec. 2.4.1).

Smart Space Back-End The term *Smart Space Back-End* is used to describe the distributed networked hosts that run KAs and span the VSL in a Smart Space.

Smart Space Orchestration The term Smart Space Orchestration describes the management of Smart Devices in a Smart Space via software. Smart Space Orchestration typically has a goal, such as saving energy in a space, that it reaches by orchestrating multiple Smart Devices (see Sec. 2.4.1).

Smart Space Service management The S2S is the service management of DS2OS. It deploys services in to Smart Spaces, and it manages services within a Smart Space. The S2S consists of the RTE, the SHE, the NLSM, the SLSM, end the S2Store (see Sec. 7.2).

Smart Space Software Orchestration See Smart Space Orchestration..

Smart Space Store The S2Store is the App store for Smart Spaces. It is introduced as part of DS2OS in this thesis (see Sec. 7.2.6).

Software Orchestration See Smart Space Orchestration.

Virtual Context Virtual Context is context that is not served by the VSL context repository of a KA but instead delivered via a callback into a service that registered a Virtual Node. Virtual Context enables a direct coupling of services by using a descriptive data structure. It extends the blackboard communication pattern (see Sec. 5.3.4).

Virtual Node The term Virtual Node is used for VSL nodes that provide context via a function callback into a service (see Sec. 5.3.4).

Virtual State Layer The VSL is the μ-middleware that is developed in this thesis. It is implemented as Peer-to-Peer (P2P) system of distributed KAs. The VSL provides a context overlay that enables transparent exchange of context between services (see Sec. 5).

ds2os.org/