Diss. ETH No. 15908

Concepts and System Structures to Support Collaborating Everyday Items

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich (ETH Zurich)

for the Degree of
Doctor of Sciences

presented by
Thomas Marcus Schoch
Diplom-Informatiker, Darmstadt University of Technology
born June 13, 1977
citizen of Germany

Prof. Dr. Friedemann Mattern, examiner
Prof. Dr. Thomas Gross, co-examiner

2005

Acknowledgments

First of all, I would like to thank my adviser Prof. Friedemann Mattern. He always took the time to discuss the different aspects of my work and provided me with constructive feedback when I asked for it. Besides that, he left enough room for me to work on my own ideas.

I am very grateful to my co-advisor Prof. Thomas Gross. Although he had a large workload, he took the time to provide me with valuable feedback in order to improve this dissertation.

I would also like to thank all the students I worked with in order to implement and extend the concepts that are presented in this dissertation. These students are Daniel Schädler, Tobias Schwägli, Reto Vögeli, Andreas Westhoff, Marco Steiner and Thomas Eicher.

Next, I would like to thank all my colleagues at the ETH Zurich and at the University of St. Gallen for the positive and motivating working atmosphere they helped create: Heidi Gülgün, Jürgen Bohn, Oliver Christ, Vlad Coroama, Svetlana Domnitcheva, Christian Frank, Sandra Gross, Marc Langheinrich, Marie-Luise Moschgath, Matthias Ringwald, Michael Rohs, Frank Siegemund, Christian Tellkamp, Frederic Thiesse and Harald Vogt. Special thanks to Kay Römer, Martin Strassner, Christian Flörkemeier, Matthias Lampe and Thomas Dübendorfer, with whom I wrote my publications, and special thanks to Oliver Kasten who supported me with the programming of the BTNodes.

I am also grateful to Prof. Elgar Fleisch, who is the initiator of the M-Lab project, in which I was the first research assistant. Besides its scientific approach, this project taught me a great deal about practical issues.

Last, but not least, I want to thank Nick Bell and Keno Albrecht for their 'read after write verification'.

Contents

Abstract

The goal of this dissertation is to contribute towards the realization of the vision of a world with "smart" everyday items. Smart everyday items differ from regular everyday items insofar as they know their whereabouts, perceive their environment, and are able to communicate with other smart things. For example, a bottle that contains temperature sensitive chemicals can monitor the temperature and if a certain limit is exceeded, the bottle can send an alarm message; and if its own temperature sensor is broken, it simply asks a bottle in its vicinity. Another task that a bottle might take over is less complex and more cost efficient: it signs in and off at the warehouse management systems of supply chain participants, so that they know exactly what inventory is currently stored.

In the introductory *first chapter* of this work, this vision will be extended and it will be shown how it can be embedded in the more general vision of ubiquitous computing, which was first formulated by its pioneer Marc Weiser. Starting from these visions, economically practicable applications will be introduced in the following *second chapter*. One generic application, which will be referenced throughout this work, will be discussed more deeply. It consists of a supply chain application with several participants that exchange products. At first, these economic applications serve, especially the supply chain application, to enumerate the requirements that need to be fulfilled by systems that aim at realizing a world of smart everyday items. By this, we understand a software system that comprises, on the one hand, a software framework that provides application developers with the required abstractions in terms of a generic class library and, on the other hand, middleware services that extract and execute application-independent tasks. It is the assumption of this dissertation that the concepts and systems presented here describe and support a world of collaborating everyday items in a substantially better way than would be possible with current means.

In the *third chapter*, the technologies are presented on which the smart thing systems considered in this work are based. It is about software systems, on the one hand, that support dynamic client/server applications, such as Jini or Web Services, and hardware systems, on the other hand, that serve to identify objects automatically, such as bar code or RFID systems.

After these preparations, the *fourth chapter* introduces a model that describes a world of collaborating everyday items. On the highest level, the model differentiates between four concepts: a *thing*, the *tag* attached to it for identification, a *representation* of that thing in IT systems, and a *service infrastructure* that allows for the coupling of the tag and its representation. These four concepts and their relative dependencies will be expanded and presented. Besides the definition of the vocabulary that is used throughout this work, the model serves to formulate the problems and challenges of an implementation of such systems precisely, so that the model can be used as a template for future implementation. During the work with the model, the following problem areas and questions arose that

are also addressed by the model: "How can we abstract from the implementation details of the underlying identification technology such as bar codes or RFID?", "How must an identifier be structured?", "Which kind of location model is appropriate?", "How does the service infrastructure realize the coupling between a thing and its representation?", "Which other services can be provided by the service infrastructure?", "What is the minimum functionality that has to be provided by a representation?", "What are the possible options for migrating a representation?" and "How is the interaction between applications and representations realized?".

The implementation of the model in three different smart thing systems as well as the corresponding insights, are the topics of the *fifth chapter*. Both topics can be regarded as a solution to the above mentioned problem areas. First, the question of how to structure applications that use one of the three systems will be addressed. A similar issue is how such systems and the applications based on them can communicate with already existing applications. Building on this basis, a description is given of the three systems that cover different aspects of the model, so that a subsequent system extends a previous one, as well as trying out different technologies. Some features that distinguish the systems are the usage of Jini or Web Services as an underlying client/server platform, various location models, and the option of a representation either migrating from one host to another in order to be executed as near as possible to the location of the thing, or of the representation being permanently located at a host where it is reachable from a client application at all times.

The deployment of the smart thing systems, i.e. the execution of the application and the middleware service on computers, as well as the application development, are evaluated in the *sixth chapter*. This evaluation relies, on the one hand, on the supply chain application introduced in the second chapter as a generic example and, on the other hand, on the measurement of the relevant system parameters of the underlying technology. Finally, the results will point out whether the systems actually support application developers and whether they scale well enough to be deployed on a larger scale.

The *seventh chapter* shows to what extent links or overlaps to related work exist, thereby suggesting the broader value of the concepts and implementations presented here. Mainly those projects are presented that also aim to realize a world of smart things, as well as work that is intended for a different goal but has similar subordinate problems to solve.

In the *last eighth chapter*, building on the evaluation of the sixth chapter, the pros and cons of the model and the smart thing systems are discussed, and potential extensions of the model and the systems are sketched. In addition, adjacent research areas and their challenges are addressed.

Zusammenfassung

Die Realisierung der Vision von einer Welt mit "smarten" Alltagsgegenständen ist Gegenstand dieser Dissertation. Smarte Alltagsgegenstände unterscheiden sich insofern von bisherigen Gegenständen, als dass sie ihren Aufenthaltsort kennen, ihre Umwelt wahrnehmen sowie mit anderen smarten Alltagsgegenständen kommunizieren können. So kann beispielsweise eine Flasche, in der sich temperaturempfindliche Chemikalien befinden, die Temperatur überwachen und im Falle des Überschreitens eines Grenzwertes eine Alarmmeldung absetzen; und falls ihr eigener Temperatursensor defekt ist, so fragt sie einfach eine andere Flasche in ihrer näheren Umgebung. Oder sie übernimmt nur eine weniger komplexe Aufgabe, indem sie sich automatisch in die Warenwirtschaftssysteme der Lieferkettenteilnehmer ein- und ausbucht, so dass diese immer genau wissen, was sich tatsächlich in ihrem Lager befindet.

Im einleitenden *ersten Kapitel* dieser Arbeit wird diese Vision ausgebaut und konkretisiert, wie sie sich in die generelle Vision des Ubiquitous Computing, welche erstmalig von ihrem Pionier Marc Weiser formuliert wurde, einbettet. Daneben wird aufgezeigt, worin genau der Beitrag dieser Dissertation besteht und wo sie sich ihre Schranken setzt. Ausgehend davon werden im darauf folgenden *zweiten Kapitel* konkrete Anwendungen mit smarten Alltagsgegenständen vorgestellt, die aus betriebswirtschaftlicher Sicht sinnvoll, d.h. rentabel sind. Detaillierter besprochen wird eine generische Anwendung, die im Verlauf der Arbeit des Öfteren referenziert wird. Es handelt sich dabei um eine Lieferkette mit mehreren Teilnehmern, die smarte Produkte austauschen. Zunächst dienen solche Anwendungen dazu, die Anforderungen aufzustellen, die Systeme erfüllen müssen, welche sich zum Ziel setzen, eine Welt smarter Alltagsgegenstände zu ermöglichen. Unter System verstehen wir ein Software-System, das sich zum einen aus einem Software-Framework, das Entwicklern die nötigen Abstraktionen in Form einer generischen Klassenbibliothek zur Verfügung stellt, und zum anderen aus Middleware-Diensten, die applikationsunabhängige Aufgaben kapseln und ausführen, zusammensetzt. Die Arbeit geht von der These aus, dass die hier vorgestellten Konzepte und Systeme eine Welt kooperierender Alltagsgegenstände wesentlich besser beschreiben und unterstützen als es mit momentan verfügbaren Mitteln möglich ist.

Im *dritten Kapitel* werden die Technologien vorgestellt, auf denen die in dieser Arbeit betrachteten Systeme aufsetzen. Dabei handelt es sich zum einen um bereits existierende Software-Systeme, die dynamische Client-Server-Anwendungen ermöglichen, wie Jini oder Web Services, und zum anderen um Hardware-Systeme, die der automatischen Identifikation von Objekten dienen, wie beispielsweise Barcode- oder Transpondersysteme.

Auf den vorausgegangenen Kapiteln aufbauend, wird im *vierten Kapitel* ein Modell vorgestellt, das eine Welt kooperierender Alltagsgegenstände beschreibt. Auf oberster Ebene unterscheidet das Modell zwischen den vier Konzepten *Gegenstand*, das

an ihm befestigte *Tag* zur Identifikation, eine *Repräsentation* des Gegenstands in IT-Systemen sowie einer *Dienstinfrastruktur*, die die Kopplung zwischen dem Tag und der Repräsentation ermöglicht. Diese vier Konzepte sowie ihre Abhängigkeiten untereinander werden dargestellt und noch feiner unterteilt. Das Modell dient dazu, die Probleme und Herausforderungen, die es bei der Umsetzung von Systemen, die kooperierende Alltagsgegenstände ermöglichen, präzise zu formulieren, so dass das Modell auch als Vorlage der späteren Umsetzung dienen kann. Bei der Ausarbeitung des Modells ergaben sich u.a. die folgenden Problembereiche und Fragestellungen, die in dieser Arbeit ebenfalls betrachtet werden: "Wie kann von der zugrunde liegenden Erkennungstechnologie, wie Barcode oder Transpondersysteme, abstrahiert werden?" "Wie müssen Bezeichner der Gegenstände aufgebaut sein?", "Welche Art von Lokationsmodell ist zweckmässig?", "Wie ermöglicht die Dienstinfrastruktur die Kopplung zwischen Gegenstand und Repräsentation?", "Welche anderen Dienste kann die Dienstinfrastruktur anbieten?", "Was ist die minimale Funktionalität, die eine Repräsentation bieten muss?", "Welche Möglichkeiten bestehen, um eine Repräsentation zu migrieren?", "Wie sieht die Interaktion zwischen Anwendungen und Repräsentationen aus?".

Die Umsetzung des Modells in drei verschiedene Systeme sowie die daraus gewonnen Erkenntnisse sind Gegenstand des *fünften Kapitels*. Beides stellt gleichzeitig den Beitrag der Arbeit zur Lösung der oben aufgezählten Problemebereiche dar. Zunächst wird erläutert, wie überhaupt Applikationen strukturiert werden müssen, die eines der drei Systeme nutzen. Ähnlich gelagert ist die Frage, wie solche Systeme und die darauf entwickelten Applikationen mit bereits existierenden Systemen gekoppelt werden können. Darauf aufbauend werden die drei Systeme, welche verschiedene Bereiche des Modells abdecken und diverse Technologien erproben, beschrieben. Einige Merkmale, in denen sich die Systeme voneinander unterscheiden, bestehen u.a. in der Nutzung von Jini oder Web Services als zugrunde liegende Client-Server-Plattform, unterschiedlichen Lokationsmodellen, und der Möglichkeit von Repräsentationen, von einem Rechner zu einem anderen Rechner migrieren zu können, um möglichst nahe am Aufenthaltsort des Gegenstands ausgeführt zu werden oder sich fix auf einem Rechner zu befinden, wo sie für eine Client-Applikation immer zu erreichen sind.

Sowohl der Einsatz der Systeme, d.h. die Ausführung der Applikationen und der Middleware-Dienste auf Rechnern, als auch die Anwendungsentwicklung, werden im *sechsten Kapitel* bezüglich qualitativen und quantitativen Kriterien evaluiert. Diese Bewertung beruht zum einen auf der Lieferkettenapplikation, die im zweiten Kapitel als generisches Beispiel vorgestellt wurde, und zum anderen in der Messung der relevanten Systemparameter der zugrunde liegenden Technologien. Die Ergebnisse zeigen schliesslich auf, ob die Systeme tatsächlich Anwendungsentwickler unterstützen und ob sie gut genug skalieren, um im grösseren Massstab Anwendung zu finden.

Das vorletzte *siebte Kapitel* zeigt auf, inwieweit sich Anknüpfungspunkte oder Überschneidungen zu verwandten Arbeiten ergeben, bzw. worin relativ dazu der Wert der hier vorgestellten Konzepte und Implementierungen liegt. Es werden hier hauptsächlich die Projekte vorgestellt, die ebenfalls eine Welt smarter Gegenstände ermöglichen wollen, als auch die Arbeiten, die zwar ein anderes Ziel verfolgen, aber ähnliche Teilproblemstellungen zu lösen haben.

Im letzten *achten Kapitel* wird nochmals das Für und Wider des Modells und der Systeme diskutiert. Des Weiteren werden potentielle Erweiterungen des Modells und der Systeme skizziert sowie angrenzende Forschungsgebiete und deren Fragestellungen angesprochen.

Chapter 1

Introduction

The first ideas concerning collaborating everyday items are already some fifteen years old and were mainly triggered by academia [120, 122]. A broader interest from industry first arose within the last few years. One reason for the interest from industry might be that the former Auto-ID Center[1], an organization with over 100 major companies, and its successors, EPCglobal and the Auto-ID labs, are trying to establish industry standards for an "Internet of Things"[2]. Another reason is that the hardware technology necessary to enable collaborating everyday items became cheaper which in turn caused industry to find new applications for collaborating everyday items.

Under the term collaborating everyday items we understand regular everyday items like bottles or umbrellas that possess additional capabilities, actually provided by an electronic background infrastructure that is able to detect the everyday items. Figure 1.1 shows an everyday item that has a connection to a background infrastructure which provides it with additional capabilities. Such augmented everyday items, which we also call *smart things*, have an identity, can be localized, are able to communicate with other everyday items or can sense their environment.

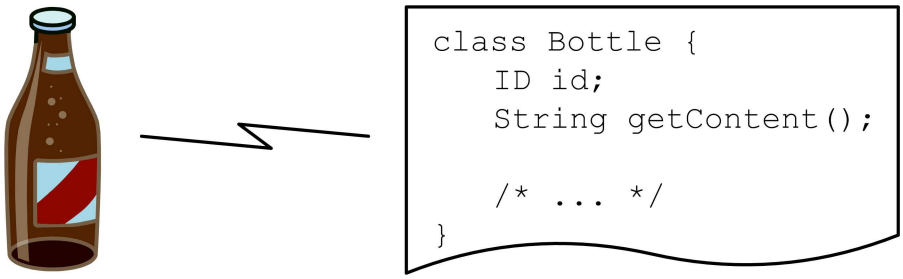

Figure 1.1: An everyday item with additional capabilities

The underlying vision and how it can be derived from the more general ubiquitous computing vision will be explained in Section 1.1. The contribution of this dissertation, which will be described in more detail in Section 1.2, lies, as the title indicates, in providing the necessary concepts and system structures that enable a world of myriads of smart things to be realized. To achieve this goal, the dissertation has to set itself some limitations (see Section 1.3) in order to be able to focus on the core aspects. The outline of the remainder of this dissertation will be presented in Section 1.4.

[1] www.autoidcenter.org
[2] www.autoidlabs.org/aboutthelabs.html

1.1 Vision

> The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. (Mark Weiser, 1991)

This quotation describes what the term and the vision *ubiquitous computing* (ubicomp) means. Mark Weiser, the pioneer who invented this term, already started work on this topic in 1988 at Xerox PARC [120, 122]. From his point of view, the idea of a "personal" computer is misplaced, since human users have to put too much attention on how to handle the personal computer instead of concentrating on their actual task, such as writing a letter. Thus, Weiser proposes designing new computer systems that take into account the natural human environment which allows the computers to vanish into the background. In that sense, ubicomp is the opposite of virtual reality which tries to create or simulate another world than the real world in existing traditional computers. Ubicomp, on the other hand tries to integrate new computers invisibly into the real world with the intention of facilitating our usual tasks.

Weiser started his experiments with computers in three different sizes: the ParcTab is a palm-sized computer, the ParcPad is an electronic notepad and the Liveboard is an electronic blackboard. These were distributed throughout an office environment and used to facilitate the common tasks there. Since these kind of devices are not truly invisible, Weiser called this phase, in which small electronic devices interact with each other and with users, phase I of ubicomp and later phases are yet to be determined. He stated that when almost every object contains a computer, then obtaining information about the real world will be trivial. Questions such as: *"Who made this dress? Are there any more in the store?"* are easy to answer then.

Neil Gershenfeld, a professor at the MIT Media Labs, went a step further. In 1995 he was involved in the foundation of the *Things That Think* (TTT) research consortium, which comprised over forty companies that looked at how the physical world meets the logical world [41]. The pieces these two worlds consist of can be described as bits, people and atoms, i.e. computers, users and everyday items. Among other things, Gershenfeld worked on so-called electronic paper, which is a sheet of paper that can change its content depending on what it should display, or he looked at Personal Area Networks (PAN). The latter could consist of a computer that is integrated into a shoe that powers itself by the energy generated through walking and communicates with other devices like a watch or eyeglasses by using the human body as a conductor for small amounts of current.

Besides these technical aspects, Gershenfeld also considered the basic requirements for things to be able to think by issuing a proclamation of the *Bill of Things' Rights*: things have the right to have an identity, access other objects, and detect the nature of their environment. These basic requirements or rights come close to our understanding of smart things. Gershenfeld sees Radio Frequency Identification (RFID) systems – attaching small chips with antennas to objects which can transmit a unique identifier to a backend system – as one enabling and cheap technology to realize this vision. And when such smart things now start to communicate among themselves, he comes to the following conclusion: *"In retrospect it looks like the rapid growth of the World Wide Web may have been just the trigger charge that is now setting off the real explosion, as things start to use the Net so that people don't need to."*

While Weiser's ubicomp and Gershenfeld's TTT describe the vision in a top-down approach, the term *pervasive computing*, which is frequently used in industry, describes the

vision from the bottom-up, i.e. what can be achieved with the technology that is available right now or in the near future. For example, [16] subsumes issues like batteries, displays, human computer interfaces (HCI), operating systems for small devices, wireless communication protocols, the wireless application protocol (WAP), speech recognition, personal digital assistants (PDAs), web services and more important the integration of all these under the term pervasive computing. Looking from the perspective of industry, IBM's chairman Lou Gerstner described pervasive computing as: *"A billion people interacting with a million e-businesses through a trillion interconnected intelligent devices..."*.

Another explanation of this vision is given by Friedemann Mattern, a professor at ETH Zurich, who argues from a technical point of view that the achievement of this vision is a consequence of five trends in the IT domain [76]. The first trend is given by *Moore's law*, which states that the number of transistors per chip area doubles every eighteen months with the consequence that, on the one hand, chips of the same size become more and more powerful, and on the other hand, if the functionality remains constant, the chips get smaller and smaller, so that they can be integrated into everyday items. The second trend refers to the development of *new materials* like the electronic paper mentioned before that allows natural interaction schemes with the logical world. A third trend can be seen in the progress of *communication technologies*: besides the increase in bandwidth, new wireless near distance technologies like Bluetooth and PANs give rise to new applications. *Wireless sensor networks*, as the fourth trend, provide the means to autonomously monitor the environment for changes, e.g. the temperature in a certain area. The last trend refers to new concepts that *model the infrastructure* for such smart things. This last trend is also dealt with by this dissertation since little research has yet been carried out in this field.

The vision of ubicomp, TTT and pervasive computing as well as its technological drivers build the foundation for our concepts of collaborating everyday items, but one main difference is that in our concepts, we focus on the communication and collaboration of smart things among themselves, and in contrast to the ubicomp vision, we do not pay so much attention to the user. We look at smart things as consisting of the traditional thing and some kind of a *virtual representation* that is responsible for the thing as shown in Figure 1.1. Just as the spirit has control over the body, the virtual representation should have control over the thing. Similar to Gershenfeld's Bill of Things' Rights, we demand that a smart thing has an identity, knows its own location and all the other smart things in its proximity, is able to communicate with other smart things, and is able to monitor its environment and its own state. Although users do not normally pay attention to the additional capabilities of smart things, since they are used in their traditional way, the users benefit indirectly from them, e.g. in a supply chain application where the products know that they have arrived at a distribution center and therefore automatically register themselves at the warehouse management system of this distribution center. If these smart things are perishable goods, they can be equipped with thermometers which allow them to verify whether they are still okay, and if the thermometer of one smart thing breaks, it can ask the other smart things in its proximity for their temperature, which should be similar to its own. This example shows that although these smart things are handled by a human like ordinary things, the collaboration among them provides additional benefits without annoying the human user with any computer interaction.

1.2 Contribution

The overall goal of this dissertation is to present concepts that model a world of collaborating everyday items and a smart thing system that implements this model. Such a system consists of two parts: first, a middleware layer that provides general infrastructural services that can be used by any application, and second, a software framework that facilitates the development of applications. In this dissertation we discuss three systems that were built iteratively. Subsequent systems extend the model or implement other architectural design decisions. The work with the systems and the adaptation of the model revealed the relevant aspects and challenges in this area.

Some of the challenges, which can be roughly subdivided into four classes, and which are discussed in this dissertation, are:

- What are the high-level concepts that appropriately model a world of collaborating smart things?
- Technological issues:
  - How can we abstract general requirements from the underlying identification and localization technology?
  - How does the identification take place and how is it related to localization?
  - How are sensors and actuators on a smart thing controlled?
  - What is the appropriate location model for our purposes?
  - How is an identifier structured?
- Infrastructural issues:
  - How is a connection between a smart thing and its virtual representation established?
  - Where and how can a smart thing store and retrieve its data?
  - How is the functionality between the smart thing and the background infrastructure distributed?
  - Where do we put the application logic?
  - What middleware services are mandatory, and which are optional?
- Issues regarding the virtual representation:
  - What is the minimum functionality of a virtual representation?
  - What relations among smart things are useful?
  - How does the communication between two smart things take place?

The results of the discussion of the above questions, on the one hand, should be the basis for further research, and on the other hand, should support developers in designing such systems and applications.

Unless otherwise noted, the personal pronoun "we" refers solely to the author of the dissertation in order to avoid using the personal pronoun "I".

1.3 Limitations

Since the scope of this dissertation covers a broad domain, some limitations have to be made to be able to focus on the core challenges. To avoid confusion about the term *smart*: we neither use this term in the sense of "intelligent" nor do we address problems of artificial intelligence research. Smart in "smart things" only means that such things have additional capabilities that need to be explicitly programmed if the infrastructure does not provide them.

Security and privacy issues play a crucial role in this domain and both issues are subject to current research. Instead of providing a whole security and privacy architecture, we only propose methods of integrating the results of the ongoing research.

The three smart thing systems we developed are only prototypical implementations that enable the verification of the model and the implementation of some example applications. That means they cannot be deployed in real scenarios, but the systems can be easily extended for that purpose. None of the three systems implement the whole model that we propose but parts of it, since we had to minimize the development efforts. Since the three systems are mainly complementary, it would be conceptually easy to build one system based on the others that implements the whole model.

Due to the limited availability of appropriate hardware that provides sensors and actuators, the evaluation of this part of the systems could not be conducted as extensively as the identification and localization part.

1.4 Outline

The outline of the remainder is structured as follows. Building on the vision described in Section 1.1, Chapter 2 is becoming more concrete and presents some applications which make use of smart things and which are profitable in an economical sense. One application, a supply chain, is described in more detail, since it is used as a generic example application throughout this work. Derived from the applications presented, the requirements will also be defined. Chapter 3 describes the technologies, i.e. software systems like Jini or Web Services and hardware systems used to identify objects, like bar codes or RFID, that are used by our smart thing systems. After these preparations, Chapter 4 introduces the model with its high-level concepts, comprising the thing itself, a tag attached to the thing to be able to identify the thing, the virtual representation which provides the additional capabilities and the infrastructure which has to identify the smart thing and contact the virtual representation among other things. Building on these high-level concepts, a solution to the challenges of Section 1.2 will be presented. Chapter 5 presents the architecture of our three systems. A quantitative and qualitative evaluation of the systems is given in Chapter 6, whereby the quantitative part mainly evaluates the underlying software systems that are utilized and the quantitative part compares our system based on the implementation of the supply chain application. Chapter 7 gives an overview of the related work in the field of middleware for ubiquitous computing and smart things, showing the similarities and the value of this work. The last chapter concludes with a resume of the essential contributions of this work, a valuation of the concepts, and discusses the future prospects for this research direction.

Chapter 2

Motivation and Requirements

Keeping the vision from Section 1.1 in mind, this chapter considers a world populated with smart things from a business point of view with concrete deployment scenarios. First, the M-Lab project, where the business and technical aspects of ubiquitous computing solutions are investigated, will be briefly introduced. Then some business applications which have been studied by the M-Lab will be described: these cover a broad range of potential deployment scenarios. One generic example, a supply chain application, will be described in more detail, since the general requirements for our smart thing systems, which are presented next, can be easily derived from it.

2.1 M-Lab

The M-Lab, short hand for "The Mobile and Ubiquitous Computing Lab"[1], is an applied research project, founded in 2001, that – together with several partners from industry – investigates the technical and the business aspects of ubiquitous computing in general with a strong focus on smart things and how they can help companies to make their business processes more efficient, and to provide new value-added services.

Our work at the M-Lab revealed that the vision of smart things is not only a nice idea but can help to solve real business problems, which in turn provides motivation to intensify the research on smart things. One of the M-Lab findings [36] is that the *media break* between the real world and the virtual world is a source of errors and inefficiencies and that the usage of smart things can help to prevent the media break. Figure 2.1 shows how the media break arises: data that describe the real world have to be transmitted into the virtual world. Traditionally, human users are responsible for data entry, e.g. they use a keyboard to tell the warehouse management system that a delivery has arrived. In general, manual labor is error-prone and in comparison with an automated solution less efficient. Using smart things that are able to transmit their data automatically into the virtual world bridges the gap between the real world and virtual world, and provides better performance and less errors. A delivery that is equipped with RFID labels, for example, can automatically register itself at a warehouse management system when it arrives at the warehouse. The more a smart thing can observe of its environment, the smaller is the gap between both worlds, as the figure shows. For a company this means that it can save money on data entry, the mapping is less error-prone, and it is able to process larger quantities of data.

[1] www.m-lab.ch

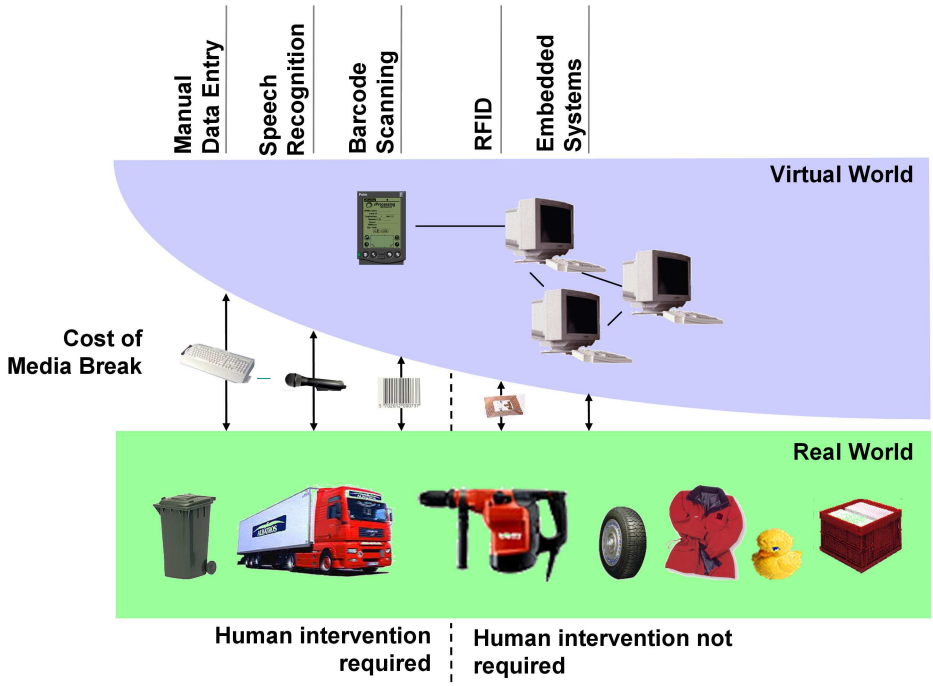

Figure 2.1: Media break (Source: M-Lab, [36])

2.2 Applications

Besides the general consideration of the media break, the M-Lab also investigates in [35] how smart things address internal and external benefit potentials in different business domains and gives an example in each case. A benefit potential is an attractive combination of factors in a company itself, its environment or in the market. Such a benefit potential can be made available by a company for its advantage, as the following examples show. A summary of the examples can also be found in Table 2.1.

Organizational potential This potential refers to the opportunity to increase added value through the reorganization of internal processes and structures. One example of that is a *smart inventory*: Pacific Century Systems, a telecommunication company from Hongkong, equips its assets with electronic tags to be able to locate them [31]. The staff can use their PCs or their mobile phone to get the position of a certain object. Thus, a real-time inventory is possible at any time and additionally, the company can analyze the data in order to monitor and optimize the degree of utilization.

Potential to reduce cost The opportunity to reduce internal costs is realized through technical rationalization. *Smart cross members*[2] at Ford are able to store all data belonging to a certain engine in the engine assembly process. This data is transmitted in real-time to the production planning system, which uses this data to control the main production line where the engines are assembled, in order to prevent bottle necks in the production process. Additionally, Ford is able to record all the production steps for each engine.

[2] A cross member is a frame on which several parts are mounted.

Know-how potential *Smart garbage cans* are an example of the opportunity to utilize internal knowledge. Ten percent of all Swedish garbage cans store a customer number and transmit it wirelessly – together with the actual weight of the garbage can when it is emptied by the garbage truck – to a central computer, which in turn uses this information for weight-based accounting and to optimize the routes of the garbage trucks.

IT potential The IT potential is realized by the added value of the deployment of new technologies and software. One example is *smart groceries*: Campofrio equips its ham with a small chip at the start of production that can record data like weight, temperature or the amount of water and fat in the ham. With this technical support, Campofrio is now able to guarantee the quality of its products.

Financial potential This potential can mainly be utilized by insurance companies. New *insurance models* provide lower insurance contributions if the covered asset can be tracked or monitored, so that insurance companies are able to detect possible risks earlier and can use this knowledge to prevent loss events.

Procurement potential The last potential refers to the opportunity to increase the added value through the adoption of new and innovative procurement concepts and systems. In particular the apparel industry has to struggle with a complex check-in process for incoming apparel: this arrives in containers with bar codes. The process of scanning the bar codes could be replaced by the adoption of new *smart containers* which automatically check themselves in at the warehouse management system.

The six examples show how smart things which have an identity and which can observe themselves and their environment are able to bridge the gap between the real world and the virtual world. Thus, they help companies which use them to reduce costs and to provide new value-added services. Although these six examples only show a small fraction of situations where smart things can be deployed, and their full potential has yet to be tapped, they show that the vision of smart things has concrete applications.

Potential	Example application
Organizational potential	Smart inventory
Potential to reduce cost	Smart cross members
Know-how potential	Smart garbage cans
IT potential	Smart groceries
Financial potential	New insurance models
Procurement potential	Smart containers

Table 2.1: Summary of potentials and example applications

2.3 Smart supply chain application

In the previous section, we presented different classes of applications. Now we shall focus on a concrete application: the smart supply chain application is a generic application

which shows different aspects where the use of smart things can bring benefits. This example application is used to derive the requirements of smart thing systems that support a world of smart things and is used later to evaluate our smart thing systems. These systems are presented in Chapter 5. The concrete benefit of making a supply chain smart has been investigated several times in the literature [5, 62, 113].

2.3.1 Overview

The goal of the following example application is to make a supply chain more efficient and to provide new services by using smart things. In this example, we have two bottlers of mineral water (LidWaters and OpenWaters), one wholesaler for mineral water (DistributeAll), one retail store that sells the mineral water (MigrosCity), and two forwarders (OnTimeDelivery and FastDelivery). All participants and their connections are depicted in Figure 2.2: the names of the six participants are not necessarily real company names but should give a hint regarding the task of each participant, as do the icons.

In comparison to regular supply chains, this one should provide several benefits from the utilization of smart things:

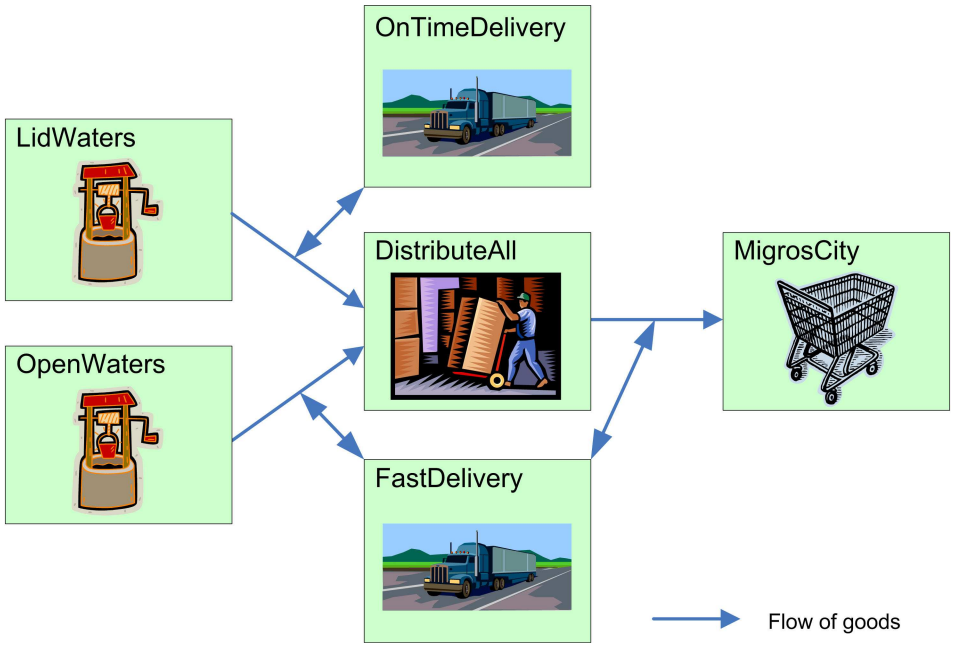

Figure 2.2: A supply chain with six participants

- total stock visibility (since smart things have to check in and check out),
- quality assurance (since smart things record data such as their temperature),
- additional statistical functions (since smart things store their data history),
- process automation (since smart things decide on their actions).

2.3.2 Smart things

First, we have to define which things in the supply chain should become smart. Since we want to cover a broad range of application types in this example, every thing that is relevant in this example supply chain will become a smart thing.

- Every *bottle* of mineral water
- Six bottles are packed into a *box*.
- Two boxes are packed into a *container*.
- A container possesses two *handles*.

These smart things and their relationships, which have been mentioned in the above enumeration, are illustrated in Figure 2.3. This example shows a container with two boxes, twelve bottles, and two handles.

Figure 2.3: A smart container with content

2.3.3 Locations

Second, we have to define the locations which the smart things pass through and where they can interact with the environment (see Figure 2.2).

- *LidWaters* is a bottler with a storage and a check-out area.
- *OpenWaters* is another bottler with a storage and a check-out area.
- *DistributeAll* is a wholesaler with two storage areas, one check-in area and one check-out area.
- *MigrosCity* is a retailer with a storage area and a check-in area.
- *OnTimeDelivery* is a forwarder with a truck.
- *FastDelivery* is another forwarder with a truck.

Since we are only interested in the actual supply chain, we do not have to consider aspects such as how the bottles come into a bottler's storage area and what happens to the bottles after they have been stored in the retailer's storage area.

2.3.4 Procedure

Next, we show how the flow of information and the flow of goods takes place, as well as how the smart things interact to provide benefits. The numbers in parentheses refer to Figure 2.4.

- Every location provides access to a warehouse management system (WMS).
- The retailer can send an order with the number of bottles (OpenWaters or LidWaters) to the wholesaler. (1)
- The wholesaler can send an order with the number of bottles to each bottler.
- The receiver of an order checks the availability of the goods and appoints a forwarder. (4)
- The receiver of an order adds the appropriate packaging (boxes and containers) to the order.
- The receiver of an order sends an advance shipping note (ASN) to the forwarder and the sender of the order. (7, 8)
- If an order arrives, then internal orders are sent to the storage areas, and the ordered goods are moved from the storage areas to the check-out area. (2, 3, 5, 6, 9, 10)
- If a truck arrives at a check-out area, then the goods are moved from the check-out area to the truck. (11, 12)
- If a truck arrives at a check-in area, then the goods are moved from the truck to the check-in area. (13, 14)
- If goods arrive at a check-in area, then internal ASNs are sent to the storage areas and the goods are moved from the check-in area to the storage areas. (15, 16)
- Every location checks, by means of the ASN or the order, whether the right number and the right instances of goods are passing through it.
- Every smart thing registers itself with the WMS when it enters or leaves a location.
- Every bottle should be able to record its temperature, since mineral water should be protected from direct sunlight and should not be frozen.
- If a bottle is no longer able to monitor its temperature, then it contacts nearby bottles.
- If the temperature of a bottle exceeds a defined interval, then an alarm is sent to the WMS.
- The WMS is able to contact every smart thing for its history.
- Every smart thing knows which other smart things it contains.

Figure 2.4 shows all the steps described above when the retailer sends an order to the wholesaler. The orders from the wholesaler to the bottlers are similar to this example.

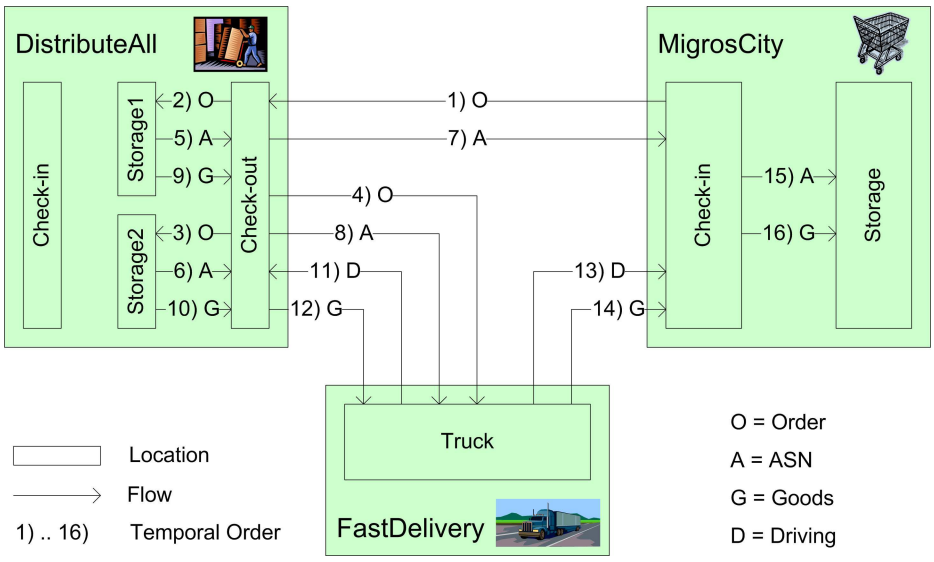

Figure 2.4: Processing of an order

2.4 Requirements

Building on the supply chain application and the other applications from Section 2.2, the requirements of a system that supports smart things can be derived.

Localization of smart things Every smart thing in the example has to be localized, at least when it comes into the range of one of the locations mentioned in Section 2.3.3, in order to be able to register itself with the warehouse management system that is responsible for that location. That means that at least symbolic location information, like "checkin.migroscity", is required, but an exact physical position would be preferable, since this information can be used for more efficient access to smart things. A physical position might be necessary when a smart thing has to be searched for within a symbolic location.

Identification of smart things Normally, the localization of a smart thing only makes sense in combination with its identity, since we need to know which smart thing is at a certain position instead of only knowing that there is an arbitrary smart thing at a certain position. This information is even more important when an instance of a smart thing out of a class of the same smart things has to be identified, e.g. although all bottles in a box are the same, each bottle has to record the temperature individually. On the other hand, sometimes only the class and not the instance information of a smart thing is crucial, e.g. if an order for six bottles of LidWaters comes in, six arbitrary bottles of LidWaters can be taken out of the storage area.

Usage of different identification and localization technologies To be universally applicable, a smart thing system must not rely on a specific identification or localization technology, since different technologies have different advantages and disadvantages, so that it makes sense to use different technologies in different scenarios. The producer of bottles, for example, uses a cheap localization technology which adequately localizes smart things on a symbolic level, while the producer of the containers equips its goods

with a costly localization technology that localizes the containers using physical positions worldwide. In another case, different companies or industries agree on the usage of different technologies, e.g. LidWaters uses another technology than OpenWaters. The use of different technologies can also occur during a transition period when one technology replaces another, e.g. RFID tags replace bar codes.

Programmatic access On the one hand, a virtual representation – the part that encapsulates the additional functionality of a smart thing – needs access to virtual resources like the warehouse management system (WMS), and on the other hand, the WMS needs to contact a virtual representation for its statistics.

Support of sensors and actuators If each bottle must be able to record the temperature for itself then each bottle must be equipped with a sensor. In a future scenario, a product could also be equipped with a small motor, so that the virtual representation could activate it in order to manipulate the product or its position. Although within the next few years sensors and especially actuators will not be as important as the identification and localization of a smart thing, they must also be supported, as this example shows.

Real-time requirements The real-time requirements refer to all processes which allow such a system to react quickly to changes in the environment or to the smart thing, e.g. a virtual representation asks the sensors for the temperature, looks at the temperature history of a product and decides to activate the cooling unit. If the command to activate the cooling unit comes too late, the smart thing could go bad, so that the system has to give some guarantees concerning real-time requirements.

Composition A smart thing can consist of other smart things, e.g. the handles are part of the container. After the production of the container, we are normally not interested in individual handles, but in the container. If we have a whole composition tree of a smart thing, i.e. a smart thing that consists of other smart things which in turn consist of other smart things, then it is possible to inherit much of the information, such as location or sensor information. In this example, only the container, as the root of a composition tree, has to make sure that it gets the location information, so that it can hand down this information to both its handles in order to provide a location history for all parts of a product.

Containedness In a similar way to the composition of a smart thing – what it consists of – a smart thing can also contain other smart things: for example, the box contains six bottles. The difference to composition is that this relation frequently changes: bottles are taken out or put in all the time, while the handles of the container are mounted once during production. Sometimes it is not clear whether a smart thing is contained in another smart thing or in a location, e.g. a bottle is in a truck, whereby the truck can be modeled as a smart thing or as a location. It is up to the model in Chapter 4 to handle such ambiguous aspects consistently.

Structure of location Besides the structure of smart things, which is realized through composition and containedness, the structuring of locations is also useful if aggregated

information is required, e.g. both storage areas of the wholesaler can be combined into one location *AllStorages* which can answer queries concerning both storage areas, such as: *"How many smart items are in all the storage areas?"*.

Neighborhood Neighborhood is an important relation between smart things that make use of a common location. Two smart things that are at the same symbolic location are in a sense neighbors since they are close to each other and share similar environmental influences like the temperature, so that a bottle whose temperature sensor has been broken could ask one of its neighbors for its temperature.

History The history of a smart thing refers to all the data that have changed during its lifetime and that can be useful to decide on actions. In principle, all kinds of data belonging to a smart thing can be part of its history, including location, composition, containedness and neighborhood information as well as sensor values and actuator commands. In the supply chain example, a bottle can use its history information to calculate the average time it stays at a location within the supply chain.

Data storage While the history is in principle independent from the way a smart thing stores its data, a smart thing should have the option to store its data locally at the location where it currently resides and to store it at a central place. The differentiation is important due to efficiency and privacy reasons. On the one hand, local data storage is more quickly accessible and only locally accessible, so that local data storage enables to data to be kept private and efficiently accessible, and on the other hand, central data storage enables the data to be used along the whole supply chain. The wholesaler in the example might not want the bottle to share its temperature data, which it sensed, with other peers in the supply chain.

Other aspects Every smart product possesses a life cycle in the physical world that roughly consists of three phases: production, use and disposal. This must be mapped into the virtual world where the virtual representation also has to go through these phases. One main task of a smart thing system is the management of virtual representations and the handling of the dynamic aspects that mainly come from entering and leaving smart things. Other aspects involve the integration with existing and new applications, i.e. the virtual representation itself has some business logic but also has to collaborate with other applications.

2.5 Summary

Based on the vision of Chapter 1.1, this chapter first introduced the M-Lab project, which enumerates potential applications where smart things can have an impact. One of these, the smart supply chain, was then described, to provide a generic example from which to derive the requirements of systems that support smart things. The requirements are used in later chapters to define a structured model of this domain and to evaluate the smart thing systems that implement this model. The model and the systems have been developed iteratively, so that an earlier system only implements some aspects of the model, and the subsequent systems mainly implement those aspects that are complementary to the previous aspects.

Chapter 3

Technology

The purpose of this chapter is to introduce the technologies, which are currently used or intended to be used with our smart thing systems. As the previous chapters already pointed out, for a thing to become smart, it must be identified, localized and it should have access to sensor data and actuators. For each of these four requirements, there are many solutions available that can contribute to their realization, so that these solutions can be integrated into our systems. There is no special need to develop our own hardware solution, since the existing solutions satisfy our requirements. Also, it is more likely that companies might deploy our systems if they can still use the technologies they have already installed in order to save money and to avoid the complexity of introducing a new system, but we also expect new technologies, or extensions of these with sensors and actuators to emerge. Thus, we will present below technologies that can be used for the identification and localization of smart things, as well as for the retrieval of sensor data and the control of actuators.

Although our focus is on providing a suitable software system, we can make use of already existing software systems. Since our smart thing systems are highly distributed and dynamic, it makes sense to build on platforms that support such scenarios, so that these platforms are also presented in this chapter.

All the systems that are discussed below are already available and because of this, there is plenty of literature available about them, so that we do not need to give a comprehensive introduction. For that purpose we refer readers to the literature references. Instead, we concentrate on those aspects that are relevant to our focus, i.e. how can the technology be integrated with our systems, and in general, where can we use the same or similar concepts of the technology for the design of our systems.

3.1 Middleware platforms

The purpose of a middleware platform is to provide tools and services that facilitate the development and the operation of distributed applications [39, 68, 116]. In Chapter 7, the middleware aspects of other ongoing research projects in the field of ubicomp will be discussed in more detail. *Jini* and *Web Services* are two already existing service discovery platforms that provide the means to describe services, discover them dynamically and invoke them, as well as providing some general ideas on how to cope with the dynamic aspects in distributed systems, which our systems are also confronted with. While Jini, with its focus on new concepts, has attracted more attention in the scientific domain, Web Services with the focus on application, have done the same in the business domain. Jini

and Web Services, which are described next, have both been used as underlying service platforms for our systems. In a nutshell, deploying these systems provides us with the means for service discovery and service invocation as well as with concepts for handling the dynamic aspects of our systems.

3.1.1 Jini

Jini [29], pronounced as in *genie in a bottle* postulates the vision: *"Network anything, anytime, anywhere"*. The current Jini system, which has been provided by Sun Microsystems since January 1999, only consists of a set of Java interfaces that "describe" the vision, but Sun itself also provides a reference implementation of the interfaces. Following the vision, Jini wants to provide the necessary concepts to allow mobile devices to connect to a network which allows the spontaneous interaction of all mobile devices and services that are part of the network. A well-known example of such an interaction is a scenario where a user with a presentation on his or her PDA enters a meeting room and the PDA automatically contacts the projection service of this room to display the presentation.

The Jini model

Jini extends the client/server concept, which is already known from classical distributed systems, with two new entities: a *service proxy*, also referred to as proxy, and a *lookup-service* (LUS). The description of the Jini model is also visualized in Figure 3.1. On the on hand, a service in Jini can be a traditional pure virtual service like a time service, or a device can provide its functionality as a service to the network, as does the projector in the example above. On the other hand, a client can be any program that uses a service, which in turn implies that a service can also act as a client for another service. To handle the dynamic aspect of spontaneously appearing and disappearing services, Jini uses the LUS, where each service has to sign in and to sign off, and if the service is not able to sign off, the LUS does so automatically after a defined interval. This mechanism guarantees that the LUS knows all the services in the subnet where it runs, so that a client can ask the LUS anytime for the services that are currently available in this subnet. To operate independently of the communication protocol between the client and the service, Jini uses the proxy concept. The service API is given by a Java interface that must be implemented by the proxy of that service. A service has to use such a proxy for the registration at the LUS so that a client that asks for that service at the LUS receives the corresponding proxy, which implements the service API. From the perspective of the client, it just locally calls a method on the proxy object that has already been downloaded from the LUS. A proxy has three options for reacting to a method call from the client. First, it can compute the result locally and return it. Second, it contacts the actual service on the Internet with a proprietary communication protocol and returns the result from the service. Third, it contacts a device, if the service represents a device, with a proprietary communication protocol and returns the result from the device.

Key concepts

To support this model and to handle the dynamic aspects in such a network, Jini proposes five key concepts that are new in comparison to traditional rather static distributed systems.

Discovery This issue refers to the bootstrapping process in Jini, since the LUS is a priori neither known by a client nor by a service. To overcome this problem, Jini introduces three discovery protocols that rely on IP multicast or the actual address of the LUS to enable clients and services to find nearby LUSs and vice versa. Closely related to the discovery protocols are the *join protocols* which ensure that a service joins one or more LUSs in a well-defined manner.

Lookup The process of a service lookup is handled by the LUS, which returns a proxy for a service as response to a lookup request from a client. To specify the service for a lookup request, a client has to use at least one of three pieces of information: the unique service identifier, the interfaces the proxy should implement or a set of *Attributes* which can represent any kind of information describing a service. Thus, the lookup process described here covers two essential parts in service discovery platforms: the service description with identifiers, interfaces and attributes and the actual service discovery with the LUS as manager for service discovery.

Leasing Especially in dynamic systems, a situation can occur where an entity does not release resources which are no longer used. In the long run, the system resources can be exhausted due to this fact. Leasing is one option to prevent such situations. Every resource in a Jini community has to be leased, i.e. the resource can only be used until the lease expires, and if the lease has not been renewed, the resource will be released. The LUS, for example, uses this concept to prevent advertising proxies of no longer available services.

Remote Events The proxy concept allows for synchronous communication between the client and the service through local method calls of the client on the proxy, whereas Remote Events allow a service to asynchronously notify a client about changes that the client is interested in.

Transactions For our purposes, this concept is not that important. Jini provides a two-phase commit protocol that allows for distributed transactions.

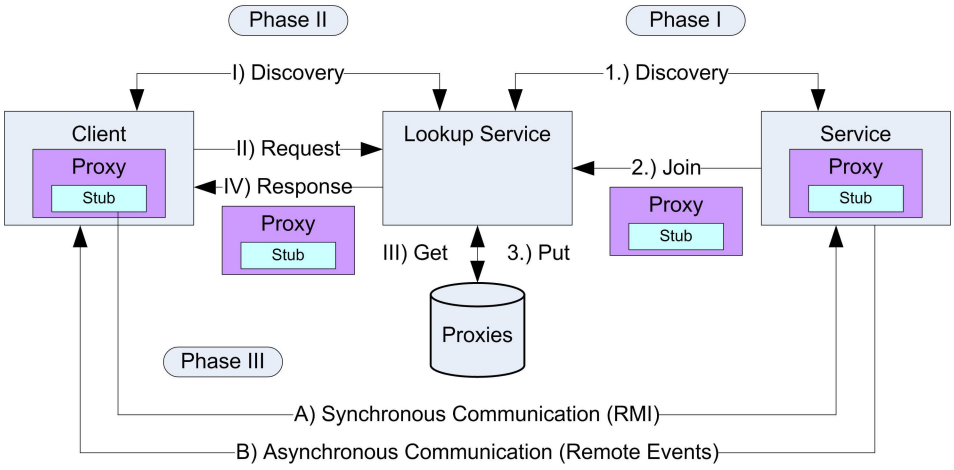

Figure 3.1: The three phases in Jini

Java Remote Method Invocation

As pointed out, Jini builds on Java as programming language, and although Jini operates independently of the underlying communication protocol, the natural choice is to use Remote Method Invocation (RMI) [58], since it is fully integrated into the Java programming language and the Java development tools. As the name indicates, RMI is used to call methods on Java objects over a network, which also means that Java's garbage collection has to be carried out in a distributed way. Thus, RMI provides the means for service invocation that was the third requirement of a service discovery platform, besides the service description and discovery of the current service.

Stub compiler The Java stub compiler tool generates the necessary stubs that are used by Java programs to call the methods of a remote object as if it were a local object since the stubs transparently marshal the parameter, contact the remote host, call the method and unmarshal the result.

RMI registry The RMI registry runs on the host of the remote object that has to register itself at the RMI registry. A Java program that wants to call the remote object first has to contact the RMI registry to retrieve a stub object. Since this functionality is part of the LUS, in a Jini environment, the RMI registry is no longer needed, but the stub classes are still needed if RMI is required to be the communication protocol. In such a case, the proxy uses the stub classes to contact the current service. Thus, the registry and the LUS solve the bootstrapping problem to obtain a first remote reference to an object.

Semantic of parameters RMI supports two semantics: call-by-reference and call-by-value. Both concepts are also depicted in Figures 3.2 and 3.3. If a parameter or a return value of a remote method call itself is a remote object, then RMI uses the call-by-reference semantic. This means that a stub class of that other remote object is taken. If a parameter or a return value of a remote method call is itself a normal object, i.e. not a remote object, then RMI uses the call-by-value semantic. In such a case, the object and all the objects the object itself is transitively referencing have to implement the `Serializable` interface, which indicates that RMI can use the serialization process to write the whole object structure into an object stream. This object stream can be transmitted to the host of the remote object, where the object stream in turn can be deserialized. Thus, the other side uses a clone of the original object.

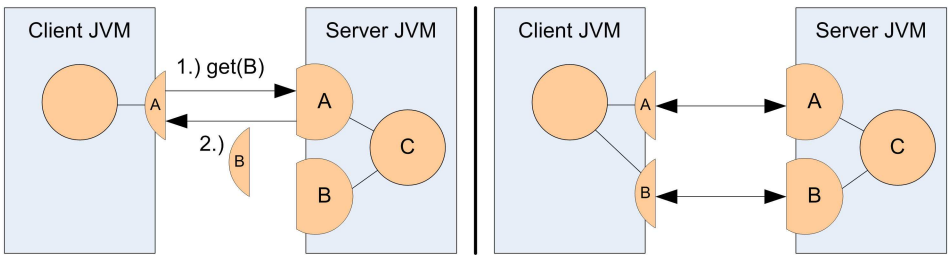

Figure 3.2: RMI: A remote method returns a remote reference

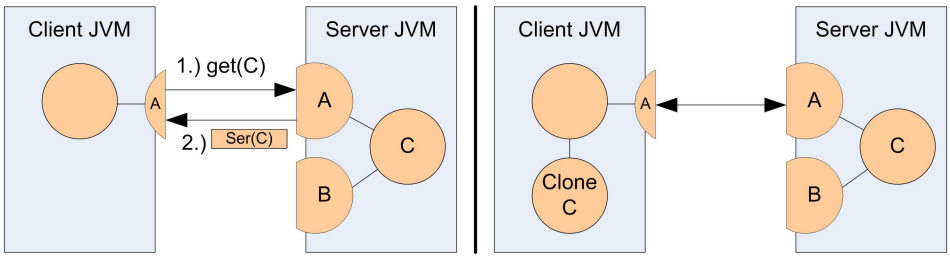

Figure 3.3: RMI: A remote method returns a serialized object

3.1.2 Web Services

The Web Services[1] approach [21] also provides the means for service invocation, service description and service discovery, but the deployment scenarios differ slightly: while Jini is mainly intended to support scenarios where different hardware devices spontaneously appear and disappear, the focus of Web Services is rather on connecting business applications over the Internet, e.g. a supplier can provide a web service, which enables a procurement application of its customer to order the required goods automatically. To fulfil the three tasks, the Web Services framework makes use of different standards: the message invocation uses the *Simple Object Access Protocol* (SOAP), the service description builds on the *Web Service Definition Language* (WSDL) and the service discovery uses the *Universal Description, Discovery, and Integration* (UDDI) standard. In contrast to Jini, the Web Services approach does not provide any additional concepts, but concentrates its efforts on creating general standards for the mentioned tasks to be interoperable, so that Web Services can operate independently of the underlying hardware, operating system and programming language. In contrast, Jini cannot operate independently of its programming language, Java.

The Web Services model As in typical client/server systems, we have a client that remotely calls methods on the service. The whole interaction scheme that is explained next can also be seen in Figure 3.4. For a client to find an appropriate service, it has to contact a UDDI server whose address it has to know in advance. The UDDI servers build a service cloud so that a modification of one of the UDDI servers will be forwarded to all the other UDDI servers that make up the cloud. A UDDI server stores information about a business, the services it provides, and the address of the service. A client can specify its service request with attributes that characterize the service itself or the business; the answer of the UDDI server contains the addresses of matching services. Having the address of a service, the client is able to ask the service for a WSDL document that, on the one hand, describes the method signatures, and on the other hand, defines the protocol binding. Although Web Services regularly use SOAP as a protocol, it is also possible to specify other protocols. SOAP defines how a method invocation can be encoded as an *Extended Markup Language* (XML) document, a model for exchanging SOAP messages, and how a remote procedure call takes place. Since SOAP does not specify a transport protocol, it also needs to be bound to another protocol, normally the *Hypertext Transfer Protocol* (HTTP), which itself relies on TCP/IP. Web Services, in fact, build on well-known web protocols (see Figure 3.5), but such a stack of protocols also means a loss of

[1]The term "Web Services" refers to the Web Services framework and the term "web service" refers to an actual service.

performance.

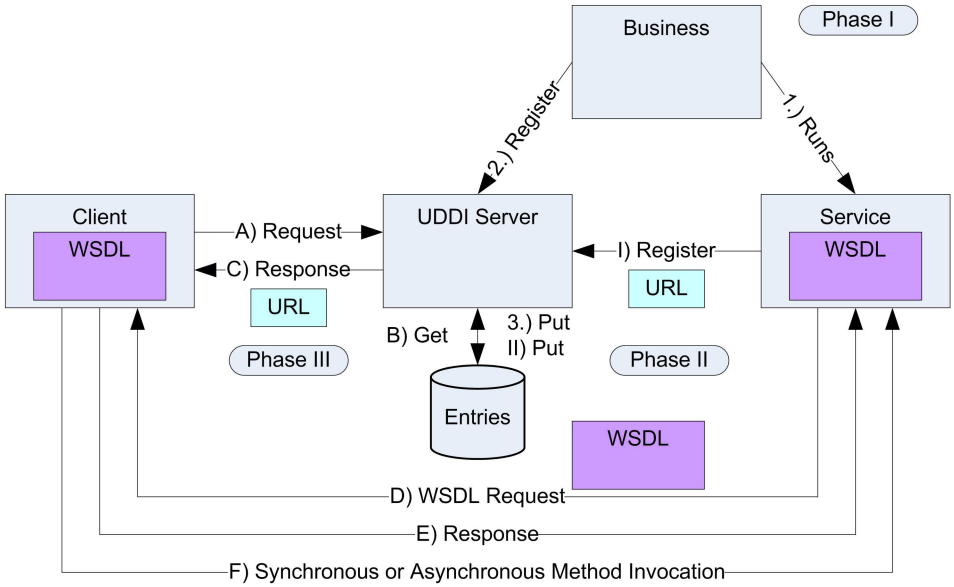

Figure 3.4: Web Services' interaction scheme

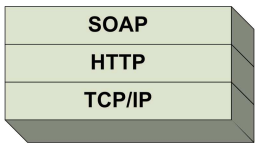

Figure 3.5: Web Services' protocol stack

3.2 Auto-ID systems

The goal of automatic identification (Auto-ID) [3, 56, 92] is to determine the identity of an entity that can either be a person or an object, so that Auto-ID systems are ideal candidates concerning the requirement that a smart thing needs to be identified. To perform the Auto-ID process, two steps have to be considered: capturing an external stimulus or signal and recognizing that signal by a computer analysis.

To classify Auto-ID systems, they can be divided into cooperative and non-cooperative systems. The former systems have additional appliances attached to the entity to facilitate the identification process, whereas non-cooperative systems have to identify the entity without any support from the entity itself. It is important to note that the term cooperative does not refer to the cooperative behavior of a person, which should be identified. Most systems that work reliably are cooperative systems. Although non-cooperative systems are improving, they are still too error-prone in the majority of cases. Besides this classification, another distinction can be made, depending on whether the system recognizes people, objects or both. In our case, only those systems need to be considered that are able to identify objects. All systems rely on the entity having a unique identifier – roughly speaking a name or a number – or unique physical characteristics. The use of a specific system mainly depends on the requirements of the physical environment and on

requirements concerning accuracy and reliability. Since for various reasons a reliability of 100 percent cannot generally be achieved, the system design should make arrangements as to how errors are handled.

3.2.1 Overview

Many different Auto-ID technologies exist today. This overview summarizes the most commonly used ones:

- Bar code
  - 1-dimensional
  - 2-dimensional (see Figure 3.6)
- Radio frequency identification (RFID)
  - Active
  - Passive
- Biometrics
  - Static
    - Fingerprint recognition
    - Retinal scan
    - Face verification
    - Hand geometry recognition
  - Dynamic
    - Speaker recognition
    - Gait recognition
    - Signature recognition
- Cards
  - Magnetic
  - Smart
- Machine vision
- Contact memory
- Infrared beacons
- Data collection systems
  - Optical character recognition (OCR)
  - Optical mark recognition (OMR)
  - Magnetic ink character recognition (MICR)

Since we use the Auto-ID technology to identify smart things, we only consider Auto-ID technologies that fulfil three criteria: they must be able to identify objects, they must be reliable and they must be able to be deployed wirelessly. Only bar code, RFID and infrared beacon systems meet these requirements, which are described next.

3.2.2 Bar code

Bar code systems, as cooperative systems, [8, 106] are used throughout all industries in the world. They identify objects or classes of objects which have unique identifiers. Classical bar codes consist of one row of alternating black and white bars with the same height but with different widths. An identity string is encoded in the spacing between the bars and the actual width of the bars. Different coding standards define the maximum width of a bar and the number of bars that form a character, and they also provide the mapping between the bars and the characters. In terms of the number of characters, some codes are extensible and some are fixed in size. A bar code reader – normally a laser scanner or a CCD camera – performs the mapping into characters which make up the identifier, and provides the data to other systems. Since classical one-dimensional bar codes have a low information density, two-dimensional bar code systems have been proposed to encode additional information besides the identifier. Bar codes of the typical size found on products can store data in the order of magnitude of some 10 bytes, whereas two-dimensional bar codes (see Figure 3.6) can store data in the order of magnitude of some 1000 bytes on the same surface. The reliability of a bar code system mainly depends on the print quality of the bar code and on the bar code being intact. In scenarios in which line of sight is present these systems are a reliable, cheap, and well-known solution for the purpose of automatic identification. There are several bar code standards for many deployment scenarios, e.g. in industry, which facilitate the utilization of these systems.

Figure 3.6: Example of two-dimensional bar code

3.2.3 RFID and infrared beacons

Radio frequency systems for identification [28, 34] are also cooperative systems, which are used to identify objects and people. A complete RFID system can be seen in Figure 3.7. The unique identifiers – which are stored among additional data on a small chip that is attached to an object – are communicated over the radio frequency spectrum. A small antenna needs to be connected to the chip to handle the reception and the transmission of the radio signal. One criterion to distinguish RFID solutions is the power mode: an active tag has its own power supply integrated, which enables it to transmit the data up to 100 meters to a reader, whereas passive tags are powered by an electromagnetic or magnetic field generated by the antenna of a reader that allows them to transmit their data up to 5 meters. The readers are normally connected to a host system, which runs the applications, but networked-enabled readers are emerging. Infrared beacon systems are similar to active RFID solutions, but they differ in the spectrum and the limited radiation angle. In contrast to RFID systems, bar code and infrared systems require line of sight between the reader and the bar code or the beacon. The latter systems are becoming more common and the cost of the tags and readers is decreasing. It is important to note that metal and fluids in the environment or as parts of tagged objects as well as legal restrictions concerning frequency and power can restrict the usage of RFID systems.

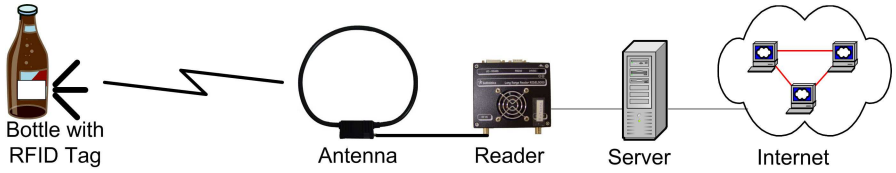

Figure 3.7: RFID system

3.3 Wireless networks

Besides the actual Auto-ID systems, wireless networks [92] can also be deployed as Auto-ID systems since they meet the requirements we used to select the appropriate Auto-ID system for our purposes: they must be able to identify objects, they have to be reliable and they must be able to be deployed wirelessly. The distinction whether a technology is an Auto-ID technology or a wireless network technology is rather vague and mostly determined by the intended purpose. The mobile unit in a wireless network can be used to tag an item and since such a mobile unit has to be identified within the wireless network in order to receive communication streams, this identification process can also be used by our smart thing systems to implicitly identify the tagged item. Depending on their coverage area, wireless networks can be roughly subdivided into *cellular*, which covers whole countries, *wireless local area networks* (WLAN), which cover buildings, and PANs, which cover rooms. Due to their widespread use and the fulfilment of our criteria for identification, it makes sense to utilize such technologies as Auto-ID systems. Since these systems only specify the communication protocol and the communication hardware, they do not necessarily provide the ability to store an identifier. In such a case, the communication module must be integrated into a larger module that participates in the identification process that we call *ID module*. Such an ID module has to be attached to a smart thing, and it must be able to contact a background infrastructure to perform the identification process.

3.3.1 Cellular

Cellular, also referred to as wireless wide area networks, enables people to place phone calls practically everywhere using a small handset. In some countries, e.g. Luxembourg, the number of registered mobile phones is larger than the actual population figure[2], so that this technology really has become ubiquitous. This means for our purposes that the whole background infrastructure is already available, and the technology has proven to be reliable. Independently from our approach, there are already some projects underway that attach communication modules to goods to track them[3].

In Europe, for example, the *Global System for Mobile Communication* standard (GSM) is currently utilized; it should gradually be replaced within the next few years by the *Universal Mobile Telecommunications System* standard (UMTS). In fact, the newer UMTS standard from the year 2000 is more powerful than the first GSM standard from the year 1982, but the differences are not that important for our purposes. More important is the *General Packet Radio System* extension to the GSM standard that allows for package-switched access to the network for data exchange instead of the connection-

[2] www.welt.de/data/2003/10/08/179343.html?s=2
[3] http://www.rfidjournal.com/article/view/100

oriented access for speech services. A connection establishment for every identification process would take too long and would be too costly, so that a package-switched usage of the GPRS standard is preferable.

GSM overview As mentioned above, we only describe those characteristics of the GSM standard that are needed to understand how we intend to integrate the technology with our systems. Figure 3.8 provides an overview of the GSM system. As the name cellular indicates, the GSM system is composed of cells whose dimensions are determined by the range of the antenna in the cell. Such an antenna is operated by exactly one *Base Transceiver Station* (BTS) which has to handle the aspects concerning the data transmission to the *Mobile Station* (MS). One BTS can handle as many MSs as the bandwidth allows. To keep the BTS cheap, most of the work is handled by a *Base Station Controller* (BSC) which can control several BTSs and which is also responsible for the handover procedure, i.e. if a user moves to another cell, the BTS of the other cell has to take over the connection. On the next level, the *Mobile Switching Center* (MSC), which the BSCs are connected to, is responsible for establishing a connection. To establish a connection, the MSC has to contact two databases: the *Home Location Register* (HLR) and the *Visitors Location Register* (VLR). Every telecommunication provider has exactly one HLR that stores data about the customers, including the current cell of the network where the mobile phone has been identified. Every MSC possesses exactly one VLR that stores temporary connection data. There are a few other units in the GSM architecture that enable authentication of users and roaming with other telecommunication providers, among other things which are not described here, since these aspects are not relevant to our systems.

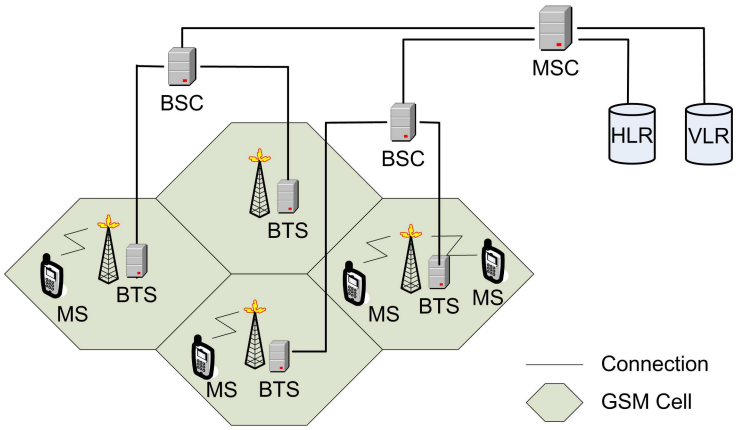

Figure 3.8: GSM overview

3.3.2 Wireless local area networks

The term WLAN is ambiguous since it denominates the whole class of wireless local area network protocols as well as the specific *IEEE 802.11* standard. Since the latter is the most successful standard in this field at the moment, we use the term WLAN below to denote this standard. WLAN as "wireless Ethernet" has become quite successful since it allows a computer to be connected to the local network without wiring. It is available at many public places, so-called hotspots, where people with their laptops can access the

Internet, e.g. at airports or at Starbucks stores[4]. The Texan city Cerritos, with fifteen-thousand inhabitants, recently made a deal with a local Internet provider to equip the whole city with WLAN access points[5]. These examples show, similarly to the GSM case, that WLAN is a wireless network technology that is becoming ubiquitous and that fulfils our requirements, so that it can be used as Auto-ID technology. For this purpose, a smart thing has to be equipped with a WLAN module that is able to transmit its identifier to the network.

Besides WLAN, there are similar technologies such as *HIPERLAN* or *HomeRF* which we do not discuss, since they do not contribute to our systems.

WLAN overview Again, the explanations concerning WLAN only consider those aspects that are relevant to integrating it into our systems. The IEEE 802.11 standard is one of several IEEE 802 standards, and from the OSI reference model perspective, it only defines the lowest layers: the *media access control* (MAC) and the *physical layer*. Since IEEE 802.11 supports two radio and one infrared modes, the physical layer is split into the *physical layer convergence protocol* (PLCP) and the three transmission specific protocols, whereas the infrared mode is used very rarely. There are two other modes in which a WLAN module can interact: the ad-hoc mode allows for spontaneous interaction with other WLAN modules, and the infrastructure mode allows for connection to *access points* (AP) that are connected to the local network. Like a BTS, an AP can handle connections to several WLAN modules. A federation of several APs enables a handover scheme – which is called roaming in this domain for historical reasons – when a user moves from the coverage zone of one AP to the coverage zone of another AP. Figure 3.9 shows a local network with two APs.

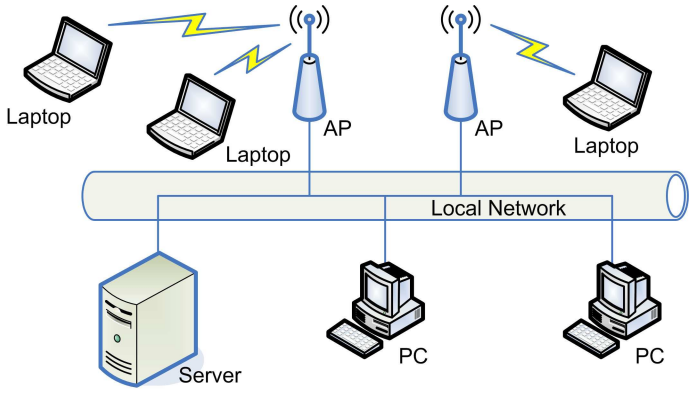

Figure 3.9: WLAN overview

3.3.3 Personal area networks

PANs represent the last category of wireless networks, possessing two important candidates: the *Bluetooth* standard [82] which uses short range radio and the *Infrared Data Association* standard (IrDA) which uses infrared. Although both technologies have slightly different application scenarios, the newer Bluetooth standard from 1999 might replace the older IrDA standard, whose first version is from 1994. Laptops, for example, were

[4] www.starbucks.com/retail/wireless.asp
[5] www.pcwelt.de/news/internet/36016/

equipped with an infrared interface in the past to exchange data with other laptops, while nowadays they are additionally equipped with a Bluetooth module. Besides laptops and handheld computers, Bluetooth modules are integrated into many new mobile phone, which enables the mobile phone to connect to a headset for speech transmission or to a laptop to allow it to connect to the Internet. One property of IrDA systems that can be advantageous, and also disadvantageous, is the limited dispersion of the signal that forms a cone of 1 meter in length with an off-axis angle of 15 degrees. On the one hand, this property limits the deployment since the IrDA modules need to be aligned; on the other hand, this property allows secure data exchange that cannot easily be eavesdropped, but the latter could also be achieved using cryptographic software solutions.

Both systems can be deployed as Auto-ID technology by attaching such a module to a smart thing that transmits its identifier using radio, or infrared to an AP that has a connection to the backend infrastructure. Due to its increasing importance over IrDA, we focus on Bluetooth, describing it in a little bit more detail.

Bluetooth overview The motivation behind Bluetooth is to replace cumbersome wiring with a PAN standard. The ability of a Bluetooth device to connect to another Bluetooth device is limited by the master/slave concept: a Bluetooth device that successfully has initiated a connection with another Bluetooth device becomes the master of the connection and correspondingly, the other becomes the slave. Within a so-called piconet, a Bluetooth device must either be a slave or a master, and there is exactly one master per piconet, which can connect to at most seven slaves. A scatternet is an extension where a node is part of two piconets, with the restriction that the node can be a master in at most one piconet. Figure 3.10b shows a scatternet consisting of two piconets in which one node is the master within one piconet and a slave within the other piconet.

Bluetooth builds on a protocol stack (see Figure 3.10a) where the master/slave concept is also realized. We only describe those layers that are needed for our purposes. On the lowest level is the Bluetooth radio layer that uses *Frequency Hopping Spread Spectrum* (FHSS) and which is responsible to synchronize different Bluetooth devices. The *Baseband*, one level above, provides two physical connections: an asynchronous one for data traffic and a synchronous one for audio only or audio and video in combination. The *Link Manager Protocol* (LMP) on the next level is responsible for establishing the connection and some further management tasks concerning the connection. These three protocols are part of the Bluetooth hardware unit and they can be accessed through the *Host Controller Interface* (HCI). Thus, the following layers must be implemented by an additional hardware unit, i.e. in our case by an ID module. The next protocol, the *Logical Link Control and Adaption Protocol* (L2CAP), supports multiplexing as well as segmentation and assembly of network packets. The last layer we mention here is *RFCOMM* that emulates a serial RS232 interface.

As mentioned above, a Bluetooth module that initiates a connection becomes the master. This initialization process consists of several states and transitions (see Figure 3.11). After a Bluetooth module has been turned on, it is in the *Standby* mode. The next state is the *Inquiry* state that enables the Bluetooth module to listen for other device addresses. The actual establishing of a connection takes place in the *Page* and the *Page Scan* modes: while one Bluetooth device is in the page mode and sends out messages with the address of the device it wants to connect to, the other device must be in page scan mode to receive such messages and if it receives its own address then a connection will be established with the Bluetooth device as master in the page mode. Devices must

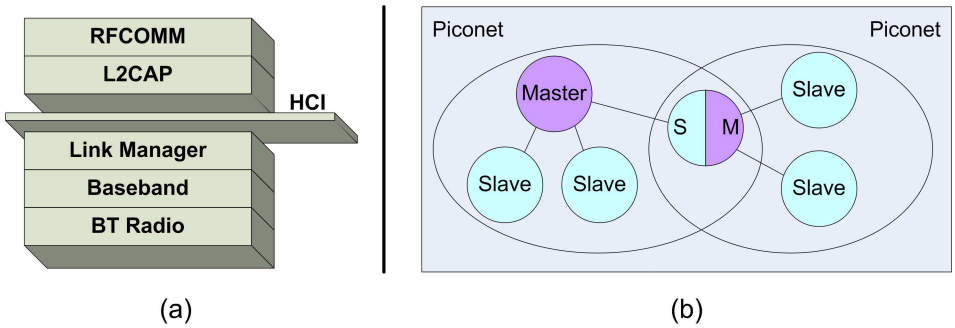

Figure 3.10: Bluetooth overview

periodically change between the two modes to enable a rendez-vous. After the connection establishment, both devices become *Active*. To save power they can change into three other modes, or if the connection should be closed, they go back to the standby mode.

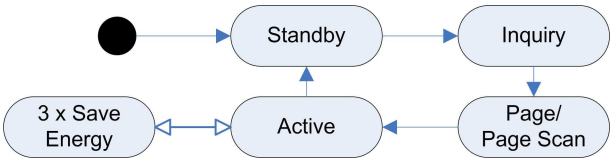

Figure 3.11: Bluetooth connection states

It is important to mention that Bluetooth provides a name request function in the LMP layer: this returns a user definable name that can be used as identifier in our systems. For such an identifier request, a connection does not need to be set up, so that the whole procedure is highly efficient.

3.4 Sensors and actuators

Since we can regard sensors and actuators as black boxes, the following description can be kept short. While sensors are used to perceive the environment, actuators are designed to manipulate the environment. In mechanical engineering, the combination of both is known as a control loop (see Figure 3.12): a system uses the sensor values to control the actuators, which in turn affects the sensors, since it changes the world the sensors perceive, e.g. a thermostat on a heater has a temperature sensor, and depending on its sensor values the thermostat decides whether to turn the heater on or off, which in fact changes the temperature in the room which the sensors perceive.

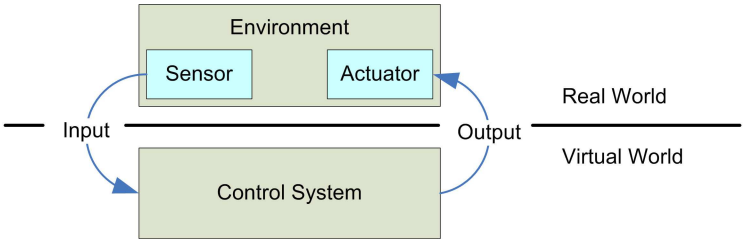

Figure 3.12: Control loop

In our systems, we do not interact directly with sensors or actuators, but indirectly, since they have to be connected to an ID module. There are already RFID modules that possess an integrated temperature sensor, but this represents the minority of cases, since most RFID tags provide only an identifier and some additional amount of writable memory. Actuators mounted on Auto-ID modules are not known. Due to their cost and the effort to save energy, it is not likely that they will be integrated into such modules in huge amounts in the near future. In general, actuators can be used to adapt the environment as a smart thing requires it, e.g. a smart thing might turn on a lamp when the environment is too dark to enable its small built-in camera to take a photo. LEDs and loudspeakers (beepers) can also be regarded as actuators although they are mainly used as a means of HCI. In a nutshell, smart things use sensors to perceive their environment, and they may use actuators to affect their environment. From the point of view of a smart thing, the actual functioning of sensors and actuators can be disregarded, since they are used as black boxes.

Sensors From an abstract point of view, a sensor transforms a non-electrical value into an electrical value. This means that a sensor relies on physical effects that change or generate a current or a voltage that can be measured, for example. In the following, we only give a brief overview about the most important sensor types and briefly sketch one of several active principles[6], since for our purposes, we can ignore the details of their actual mode of operation[7]:

Path and angle: the physical movement controls a voltage divider.

Humidity and electrical conductivity: liquids that contain ions can be regarded as resistance.

Light: light-dependent resistance (photo resistance).

Temperature: temperature-dependent resistance.

Pressure and force: foils with several layers of semiconductors with a changing transition resistance.

Sound: electrical microphones work capacitively, using a charged membrane.

Streaming: relies on the same principle as blower turbines.

Infrared: uses foils that rely on the piezoelectric effect in the infrared spectrum.

Actuators An actuator can be regarded as the opposite of a sensor: it transforms an electrical value into in a non-electrical value. An electric motor is a familiar example of an actuator: it transforms current into movement. In our case, we could imagine the following actuators being integrated into smart things:

- motors,
- LED,

[6] Active principle denotes the physical effect a sensor relies on.
[7] http://www.as-workshop.de/grundlag/snstypen.htm

- loudspeakers,
- cooling units,
- heating units or
- lamps.

The list of sensors as well as the list of actuators is not complete and only serves to give a first impression of what kind of sensors and actuators can principally be integrated into and controlled by smart things. We explicitly do not mention displays as simple actuators, but as a complex means of HCI that has to be modelled separately.

3.5 Localization

In contrast to sensors and actuators, we cannot easily ignore the details of the localization function. Localization contributes to one of the core abilities of a smart thing: that it should know where it is currently residing, and since there are no complete localization solutions available that can be integrated into our systems, we have to consider all aspects belonging to localization. These are described below.

Localization refers to the process of identifying the location where an entity resides. This information is needed by applications that provide location-based services, i.e. in our case this information is needed by the smart things themselves and the location-dependent applications. As mentioned earlier, this issue is closely related to automatic identification, since in the majority of cases, the location information is only requested in conjunction with the identity of the entity that should be localized.

3.5.1 Location Models

Location models describe the representation of location and the locatable entities [26]. Such a formal description is necessary in order for a computer to process location information. In general, the models can roughly be split into two groups, which are described next.

Symbolic location One advantage of symbolic location models is that they are easy to understand for humans. *Geographic models* that belong to this group rely on hierarchical geographic structures like postal addresses, e.g. *Haldeneggsteig 4, 8006 Zurich, Switzerland* denotes our building in Zurich. Generalizations are *symbolic models* that refer to locations by means of abstract symbols, such as *warehouse.floorA.caseI.bottle1*, which denotes the location of a bottle within a warehouse. As these examples show, locations are modeled as sets, and the members of these sets are the locatable entities. One disadvantage of these models is the manual construction and management of hierarchical symbols.

Physical positions Unlike symbolic models, these models can be efficiently processed by computers. *Physical location models*, which belong to this group, have a global and unique coordinate system. The position of an object is given as a tuple of latitude, longitude and if necessary altitude, e.g. *47.5° North, 8.5° East* refers to Zurich. *Geometric*

models, as an extension of physical location models, are also models of the second group: they represent the location and the locatable entities as a set of coordinate tuples. If they possess a single reference coordinate system, then they are called simple, whereas unified geometric models have several reference coordinate systems. The raw location information can be enriched using a mapping function that assigns additional information to a location.

Semi-symbolic models Models that use coordinates and symbolic names are called semi-symbolic. A function can map a physical position to a symbolic location and vice versa: a symbolic location can be mapped to a set of physical positions, e.g. a postal address can be mapped to set of coordinates. Most of the applications need semi-symbolic models: for efficient calculations, they use geometric models, whereas they use symbolic models for HCI purposes. Research [75, 59] currently focuses on this topic, since these kinds of models are needed most, and they are also part of our solution for smart things.

Further aspects The association of a locatable entity with a location can be realized in the following two ways. *Containment* is one possibility, i.e. to identify the region which contains the entity. The size of the container determines the resolution. Another possibility is *positioning* that relies on reporting the coordinates of an underlying coordinate system. In this case, the resolution is given by the granularity of the reference coordinate system. A relationship between the locations of two entities can be established by calculating the distance between two entities. Physical models normally use the Euclidean distance, which can be totally ordered, whereas in symbolic models objects are only partially ordered. A question concerning the distance of two objects can be answered in a physical model, for example, with "10m", in a symbolic model, however, the answer could be: "2 floors and three rooms". *Relative* versus *absolute localization* refers to the reference point: absolute localization uses a shared reference point for all locatable entities, whereas relative localization permits every entity to have its own reference point.

3.5.2 Localization Methods

After an appropriate location model has been chosen, it still needs to be fed with actual location data, i.e. some data describing the location has to be retrieved in order to determine the location within a location model. This task builds on methods that roughly can be classified into three groups [50].

Lateration and angulation Lateration computes the position of an entity by using several distance measurements from known points, i.e. for an n-dimensional determination of the position of an entity, $n + 1$ points are necessary, whereby restraints of the environment can reduce the necessary amount of known points, e.g. the knowledge that an object to be localized is beneath a satellite allows the opposite possibility to be excluded and therefore the amount of points to be reduced by one. Three methods for distance measurements can be identified. The first and conceptually simplest possibility is direct *measurement by physical actions or movements*, which is in general hard to perform automatically, for example, a motor controls a measuring stick that has to touch the object that should be localized. *Time-of-flight* information is another solution that measures the time a signal travels at a constant speed from a known point to the locatable entity, vice versa or in both directions. Ultrasonic and radio signals are commonly

used for this purpose, but filtering out the reflections of the signals is a challenge. Another challenge is the synchronization of clocks if more than one clock is involved in the measurement. The third solution for distance measurement is *attenuation*, which utilizes the decrease of signal strength depending on distance.

Angulation is similar to lateration except for the fact that angles instead of distances are measured. An n-dimensional determination requires n angles and one distance to be measured. Both procedures are depicted in Figure 3.13.

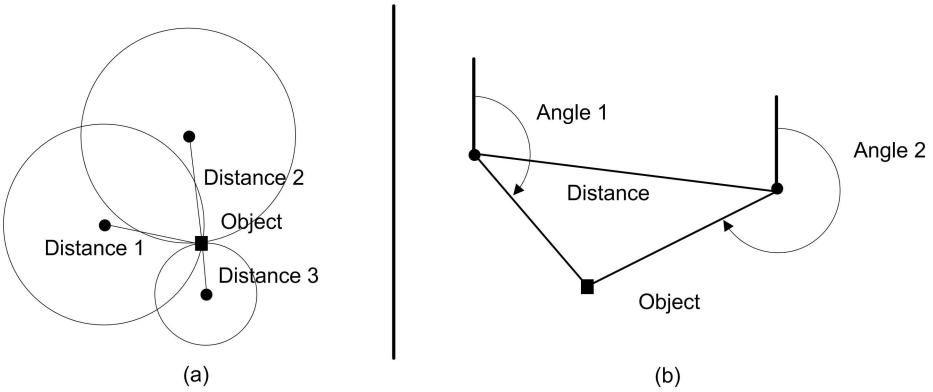

Figure 3.13: Lateration and angulation

Scene analysis This is the second general localization method. Features of a scene – not necessarily images – from a particular vantage point are extracted to determine the location of the observer, or the entity. A *static analysis* looks up the features in a database to determine the location, whereas a *differential analysis* tracks differences which correspond to movements of the observer. If features are recognized at known points, the location can be determined. One advantage of this solution is passive observation, whereas one disadvantage is the management of frequently changing environments. Image data and signal strengths are commonly used for scene analysis, but experience shows that systems that make use of this method are not very reliable, since the impact of changes in the environment is too big to be handled by these systems.

Proximity Proximity is the third general localization method. It determines whether an entity is near a known reference point by using physical phenomena that are range-limited (see Figure 3.14). One solution, i.e. *physical contact*, uses detectors or sensors such as pressure or touch sensors to determine the location of an entity. These solutions require that the entity is in contact with the system. The second group monitors *wireless cellular access points* to detect entities within the cells and therefore to determine their approximate location. This group of solutions can be used in conjunction with the wireless networks from Section 3.3. *Automatic identification* systems can also be used to determine the location of an entity if the position of the automatic identification system is known and if the identified entity is in the proximity of the system. Auto-ID systems have also been presented in Section 3.2. Lateration with time-of-flight information, as well as proximity with monitoring of wireless cellular access points, are the most promising methods to determine the position of entities. Thus, we can use proximity as a default localization method, since we either use an Auto-ID technology or a wireless network technology for identification.

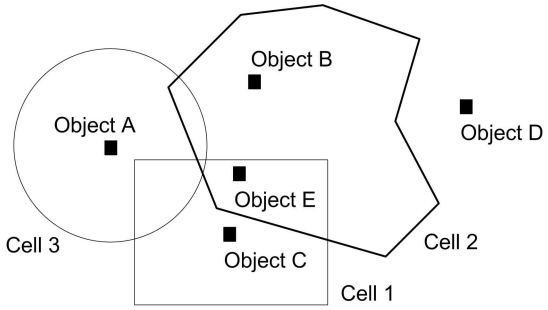

Figure 3.14: Proximity

3.5.3 Localization Systems

Localization systems [51] are systems that allow the actual localization of objects and people. They differ from the underlying phenomena used to localize the entities, the form factor of the mobile units that should be localized, which support the localization process, the energy consumption of the mobile units, whether the computations are performed by the infrastructure (remote vs. self positioning) and the resolution in time and space. The computations to localize an entity can be performed in the infrastructure, which in general is not limited by the power consumption but has the disadvantage of not guaranteeing privacy with respect to the location of an entity at a given time. A policy can specify how the infrastructure should handle privacy aspects. In such a case, the system with privacy policies needs to be trusted to actually provide the privacy that is specified in the privacy policy.

Quality To compare the quality of different localization systems, some characteristics of the systems have to be taken into consideration. *Accuracy* specifies the granularity of the system's resolution, whereas *reliability* refers to how often this resolution is achieved. Another characteristic is the statement of an *error distribution* that describes the connection between accuracy and precision. Precision refers to the difference between the real and the measured position, so that high precision represents a small difference. Finally, the *infrastructure density* has to be considered in order to compare two systems. Infrastructure density means the number of infrastructural entities, such as antennae per area which are required to achieve the specified quality.

Sensor fusion This aspect refers to the aggregation of sensor data that can improve accuracy and reliability through hierarchies and overlapping domains. Wireless sensor networks try to dynamically adapt accuracy and reliability in the effort to save power. Most applications do not need accuracy to the nearest centimeter, as often a rough determination of location is sufficient, e.g. in some cases the information that a smart thing is in a certain warehouse can be enough.

Scalability One aspect of this issue is the dimension of localization that ranges from worldwide localization of entities down to localization within single rooms. Another aspect refers to the number of objects that can be localized at any one time and/or per unit of the infrastructure, since scaling up systems is often possible by extending the infrastructure. Obstacles for scaling up are infrastructure costs, middleware complexity that might increase, and physical limitations. Cost can refer to time cost for installation,

space cost for the infrastructure and the form factor, and also capital cost for maintenance and equipment.

Outdoor systems Positioning systems may be limited to outdoor or indoor usage. The *Global Positioning System* (GPS) [7], for example, utilizes satellites for three-dimensional localization, whose signals cannot be received within buildings. A possible solution could be to install repeaters in the buildings. The system consists of 27 synchronized satellites. Four satellites and therefore four distances at a time and one GPS receiver are needed to localize the receiver's position via lateration. Since entities have to be beneath the satellites, three distances should be sufficient for lateration, but a fourth variable has to be introduced to express the time difference between the receiver and the synchronized satellites.

Indoor systems The *Active Badge* system, which was developed at the Olivetti Research Laboratory, uses infrared senders built into a badge that sends out a signal with an identifier every ten seconds, which is received by the indoor infrastructure to determine the location via proximity. *Active Bat*, which was developed at AT&T Cambridge, uses ultrasonic flight-of-time and radio signals to determine the location via lateration. The radio signal is used to synchronize the infrastructure and to recognize reflected ultrasonic signals. The reverse is done in the *Cricket* system, which complements the Active Bat system. The infrastructure sends out ultrasonic signals that are received by the entities that compute the location locally, so that privacy can be achieved. The *RADAR* system from Microsoft Research uses the existing wireless LAN technology, i.e. base stations and mobile units to determine the mobile unit's two-dimensional position by scene analysis of the signal strength or by lateration. Very accurate and precise measurements can be performed by utilizing *axial DC magnetic field pulses* from devices in the infrastructure that are received by mobile units equipped with three orthogonal antennae under the constant influence of the magnetic field of the earth.

Commercial systems Currently, GPS is most frequently used to perform localization in commercial systems. Localization with mobile phones exists in emerging applications, e.g. worried parents can obtain the location of their child's mobile phone [57], or rescue organizations can obtain the location of people that need their help [97]. All the other systems we have described here are mainly research projects or their practical usage is limited.

3.6 Summary

The aim of this chapter was to present the technologies that can be integrated into our smart thing systems which, on the one hand provide the means to manage the dynamic aspects of these systems, and on the other hand provide the necessary hardware means for a smart thing to be able to be identified and localized, to retrieve sensor values, and to control actuators.

First, Jini and Web Services were introduced: these provide the means for service description, service discovery and service invocation. Jini, additionally, provides five key concepts to handle the dynamic aspects in distributed systems: discovery, lookup, leasing, remote events and transactions. Second, Auto-ID systems were introduced: these provide

the means to identify a smart thing, in particular bar code, RFID and infrared beacons are appropriate systems. Third, wireless networks were introduced as another means for Auto-ID, since these systems also fulfil the requirements for a smart thing to be identified. Wireless networks were classified into three groups and one system for each group was proposed to be used as a means of identification: GSM modules in the cellular domain, WLAN modules in the wireless local area networks domain and Bluetooth modules in the PAN domain. Fourth, sensors and actuators were introduced as a black box that enables smart things to perceive and to affect their environment. Finally, localization was introduced to show how a smart thing can be localized. Location models, location methods and location systems have been identified as the building blocks for a localization solution. Proximity can be used as a default localization method, since it is supported by the identification technologies we have presented previously.

Chapter 4

Modeling of Collaborating Everyday Items

The aim of this chapter is to introduce a model of collaborating everyday items that serves two purposes: it is used as a template for implementation, and more generally, it defines the relevant aspects in this domain and their interdependencies. Another interpretation of the model is that it breaks down the high-level vision from Chapter 1 into concrete concepts, and it enables future research to focus on concrete domains that are denoted in this model.

We use a top-down approach to develop the model: we start with the basic high level concepts that contain the definition for a smart thing, followed by the concepts that model the basic abilities, including a location model. Next, we refine the concepts of the smart thing entities and introduce the concepts of the infrastructure in more detail. Building on these concepts, we describe the procedure by which these parts of our model have to collaborate. At the end of this chapter, we describe how the application logic can be distributed, describe some extensions of the model, then we explain how the lifecycle of a smart thing is modeled and conclude with a summary.

Besides the actual definition of the concepts, the decision process for a definition and the other options for our definition are also discussed. Although we do not discuss the requirements of Chapter 2 in the same order, they will nevertheless all be addressed. Our model, which implements these requirements, is introduced with four means: first, we define or describe each concept in a compact and numbered paragraph. If possible we try to use a rather formal definition of a concept, i.e. the preconditions also need to be well defined. In some circumstances, such a formal approach would be too complex, so that we use a rather informal description for better understanding. Second, we describe these compact paragraphs in the continuous text in more detail, third, we give examples for the concepts and fourth, we summarize the most important aspects of a concept in a figure.

4.1 High level concepts

The high level concepts presented in this section roughly illustrate how we model our domain in order to build the foundation for the rest of our model, i.e. subsequent definitions of the concepts directly make use of the following concepts.

4.1.1 Smart thing

Already in Chapter 1, we introduced a *smart thing* as an everyday item that possesses additional functions that are provided by a virtual representation in the infrastructure. The exact definition for a smart thing, which is illustrated in Figure 4.1, is as follows:

Concept 1 (Thing and representation) *A thing denotes an arbitrary object of the real world and a representation represents the additional functionality of a smart thing.*

Concept 2 (Smart thing) *A smart thing consists of a* thing *and a* representation. *A thing possesses exactly one representation and a representation possesses exactly one thing.*

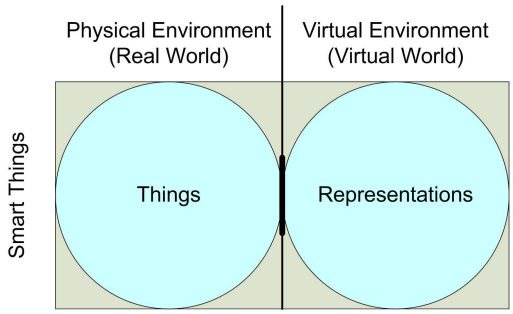

Figure 4.1: Composition of smart things

While the first half of the definition does not provide any additional information in comparison to the first introduction of the smart thing concept in Chapter 1, the second half of the definition states the unique association of a thing and its representation, which both make up a smart thing. We use the term 'thing' synonymously for an everyday item and the term 'representation' synonymously for a virtual representation. Neither can a representation provide the additional functionality for two things, nor can a thing be represented by two different representations, so that a thing and a representation build a unit that we call smart thing. The name smart thing has been chosen since a regular and "dumb" thing becomes smarter through the functions provided by the representation. As stated previously, with the term smart we neither mean intelligent, nor do we address issues of artificial intelligence (AI) research. In contrast to AI research, the additional functionality of a smart thing needs to be explicitly programmed or is part of the infrastructure, as will be shown in the following.

Things and their representations do not exist uncoupled in time and space but are rather entities of an environment: things are part of a *physical environment* that we also call the *real world*, and representations are part of a *virtual environment* that we also call the *virtual world*. As Figure 4.1 shows, both worlds divide a smart thing into its two parts.

Concept 3 (Environments I) *Every thing is an entity of the physical environment. Every representation is an entity of the virtual environment. Both environments are complementary concerning these entities.*

Later on, both environments will be enriched with additional entities, so that the purpose of the concept will become clear. Until now, we only said that a smart thing is

the combination of a thing and a representation but did not mention how they are actually connected to each other. Therefore, we introduce the concept of a *coupling*: coupling means that the thing needs to be connected with its representation, which is indicated by the bar in Figure 4.1. This general concept can be split into two more specific ones: an implicit coupling refers to the situation where the representation is directly located on the thing itself, whereas an explicit coupling demands that the representation is somewhere in the infrastructure.

Concept 4 (Coupling) *The concept coupling is equal to the concept smart thing. A coupling can either be an implicit coupling or it can be an explicit coupling.*

The equality in the first half of the definition only says that both concepts can be used synonymously. We use both synonymous terms to look at the same concept from two different views: smart thing describes the concept coming from the vision and coupling describes the concept from a technical point of view. The implicit coupling is more complex than an explicit coupling, since every smart thing must provide the same functionality of the infrastructure directly on the thing. For this purpose, a more powerful hardware solution is required on the thing, which makes the implicit coupling more expensive, so that we see the explicit coupling at first, followed by the implicit coupling shortly afterwards.

4.1.2 Tag detection system

Conceptually, a tag represents the additional functionality of a smart thing that is attached to a thing and thus, directly located on the thing itself. In that sense, one component of an explicit coupling is the *tag detection hardware* which consists of the *tag* that is *attached* to the thing and a *tag reader* that is installed somewhere in the environment that is able to *wirelessly detect* the tags and therewith the thing. Tag detection hardware is hardware-specific, which means that bar code or RFID detection hardware consists of bar codes or RFID tags attached to the things and a bar code or RFID reader that wirelessly detects these bar codes or RFID tags. The main purpose of the tag detection hardware is the identification of a thing that is indirectly identified by the tag that is attached to the thing.

Concept 5 (Tag) *A tag is a means to mark a thing. Tags are partitioned into hardware-specific classes. Every tag is attached to a thing and every thing possesses at least one tag but not more than one tag from any hardware-specific class.*

Concept 6 (Tag reader) *Tag readers are partitioned into hardware-specific classes analogously to the tags. A hardware-specific class of tag readers can wirelessly detect the tags of the corresponding hardware-specific class of tags. For every class of a hardware-specific tag reader there is at least one hardware-specific tag and vice versa.*

Concept 7 (Tag detection hardware) *The tag detection hardware consists of tags and tag readers. Tag detection hardware is hardware-specific. Every class of hardware-specific tag detection hardware covers the corresponding classes of tags and tag readers. The tag detection hardware is part of the* infrastructure.

One of the restrictions of the definitions allows that one thing can have several tags from different technologies, e.g. a thing can possess a bar code tag and an RFID tag at the same time. Although the model, and therefore the implementing smart thing systems, would be much simpler if we restrict a thing to possessing exactly one tag, we do not want to restrict the model since it is possible that a thing can have, for example, a bar code and an RFID tag during a transition period. In contrast to this example, we recommend using only one technology in order to allow for a simpler development of applications, since the existence of several tags with different capabilities results in separate handling of these capabilities, which in turn means a greater development effort. At the moment, there are several applications where an object is tagged with different bar codes, but in most cases, the information in the additional bar codes is not intended to identify the object, but to describe it. In our model, this information is encapsulated by the representation and the bar code is only used to identify the object, so that one bar code is sufficient. We define the tag detection hardware as part of an infrastructure that makes smart things possible. With the introduction of the tag detection hardware, we can complete the definition of the physical environment:

Concept 8 (Environments II) *The tag detection hardware that comprises tags and tag readers is part of the physical environment. An entity of the physical environment is either part of the tag detection hardware or is a thing.*

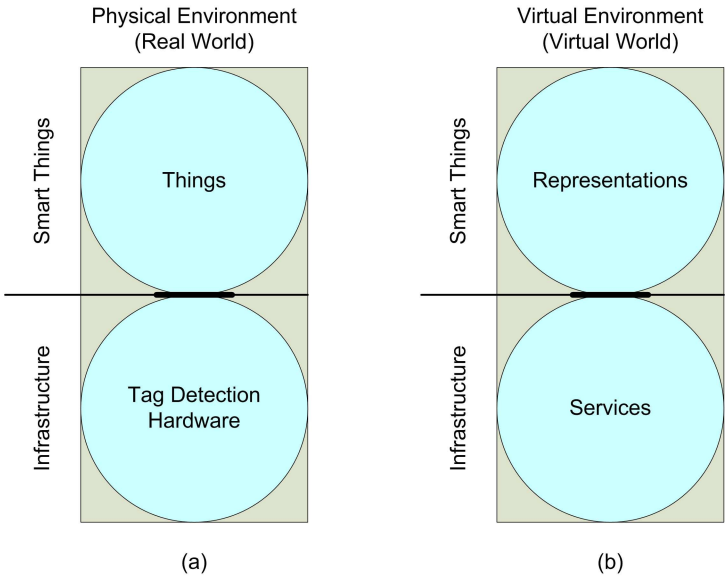

Figure 4.2: Composition of the physical and virtual environment

As Figure 4.2a shows, only things, tags and tag readers can be part of the real world. We do not need to model any other entity of the real world for our model of collaborating everyday items. Independently of whether a tag is a pure hardware solution, e.g. a bar code or a hardware/software solution, e.g. a Bluetooth module, the whole tag including the software is part of the physical environment, since we do not include the implementation of the tag. The implementation issues are transparently covered by the tag reader, which wirelessly detects the tag using a proprietary communication protocol.

Within the coupling mechanism, on the one hand, the tag builds the interface to the thing it is attached to, which is indicated by the bar in Figure 4.2a. On the other hand,

the tag reader as part of the hardware infrastructure builds the interface to the software infrastructure, which is indicated by the bar in Figure 4.3. This software infrastructure consists of several *services* which are used to manage smart things. One of these services is the *tag detection service* which is responsible for controlling a tag detection reader. Together with the tag reader, they build a bridge between the real and the virtual world, as Figure 4.3 shows. The combination of the tag detection hardware and the tag detection service is called *tag detection system.*

Concept 9 (Tag detection service and tag detection system) *A service refers to a software service in the IT domain. Every tag detection service is service- and hardware-specific. Every tag detection service controls exactly one tag reader and every tag detection reader is controlled by exactly one tag detection service. The combination of tag detection hardware and tag detection services for specific hardware is called a tag detection system.*

An RFID tag detection system, for example, comprises all RFID tags and all RFID readers with their corresponding RFID detection services. Using the above introduced concepts, we are now able to complete the definition of the virtual environment, which is shown in Figure 4.2b:

Concept 10 (Environnments III) *An entity of the virtual environment is either a representation or a service. A tag detection service is a service among other services. Every service is part of the infrastructure that enables smart things.*

The definitions for a smart thing and the environments allow us to reason about the infrastructure (see Figure 4.3):

Corollary 1 (Infrastructure) *Either a service or a tag detection hardware entity is an entity of the infrastructure. An arbitrary entity of the model is either part of a smart thing or of the infrastructure.*

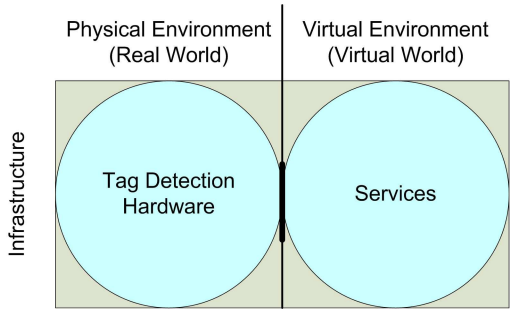

Figure 4.3: Composition of the infrastructure

In a nutshell, on the highest level in the model, we have two orthogonal concepts: an entity of the model either must be part of a smart thing or the infrastructure and an entity either must be part of the physical environment or the virtual environment. Thus, four combinations are possible:

- an entity that is part of a smart thing and of the physical environment is called thing,

- an entity that is part of a smart thing and of the virtual environment is called representation,
- an entity that is part of the infrastructure and of the physical environment is called tag detection hardware entity, or
- an entity that is part of the infrastructure and of the virtual environment is called service.

4.1.3 Managing services

Figure 4.4 shows that the four concepts in the big circles have four interfaces indicated by the four bars. We already mentioned three of them: the interface between the thing and the representation, between the thing and the tag detection hardware and between the tag detection hardware and the services. The first interface between the representation and the thing, which we also call coupling, can be regarded as a logical interface that has to be realized through the other three interfaces, from which only the one between the representation and the services is missing. Until now, we have a tag detection service in the virtual environment that finds out about what tags are detected by its tag detection reader in the physical environment and we have the representations of the tagged things in the virtual environment, also. To bridge this last gap within the virtual environment between the tag detection service and the representation, we introduce *managing services* that, on the one hand, have to use the tag detection systems to find out about things in the real world and, on the other hand, have to provide the necessary means for a representation to be able to provide its functionality.

Concept 11 (Managing services) *Managing services bridge the gap between the tag detection services and the representations.*

The actual managing services are described in Section 4.4. However, without the actual introduction of the managing services and from a high level view, we now can explain how the coupling between a thing and its representation takes place:

Corollary 2 (Coupling) *A coupling between a thing and its representation is given by the following combination of the previous concepts. Every thing has at least one tag. Every tag can be at least wirelessly detected by one tag detection reader. Every tag detection reader is controlled by exactly one tag detection service. Every tag detection service is connected to every representation by the managing services.*

The idea behind this corollary is to show that the connection between a thing and its representation can be realized through the use of the infrastructure. The above concepts and their combination make-up the high-level concepts in our model. Although these concepts intuitively describe our world of smart things, this model is still far away being easy to implement. However, the following sections will make use of these concepts to extend and to refine the model, so that an implementation of the model will become straightforward.

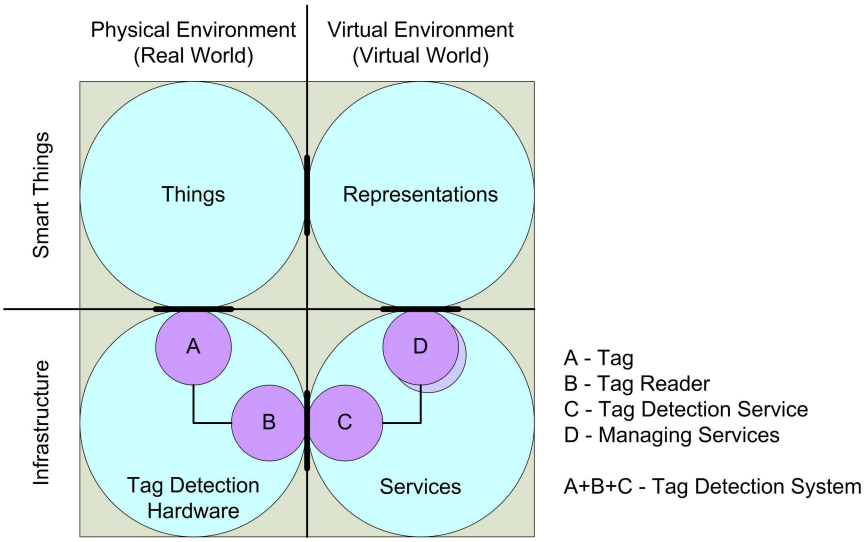

Figure 4.4: High level concepts of the model

4.2 Concepts for basic abilities

Using the high-level concepts that roughly describe our world of smart things, we are now able to introduce the concepts that model the basic abilities of a smart thing, comprising its identification and its localization as well as the retrieval of sensor values and the control of actuators.

4.2.1 Identifier

The overall goal of the utilization of an *identifier* is to uniquely identify a smart thing, which comprises a tagged thing and its representation, so that the identifier refers to three entities: the thing, its tag and its representation. Such a unique identifier is needed for two purposes: first, it can be used by the managing services as a key to store and to retrieve data for a certain smart thing, and second, it can be used by applications to identify individual things, since applications normally depend on the identities of smart things. In a nutshell, an identifier is the answer to the question: *"What is the identity of a certain smart thing?"*.

The identifier is partitioned into a *name* and *home address* analog to a *Universal Resource Identifier* (URI) [12] that consists of a *Universal Resource Name* (URN) [81] and a *Universal Resource Locator* (URL) [13]. The name is used to denote a smart thing. It need only be unique within its namespace, which is the home address. Such a home address is used to denote the name of the location in the virtual world where a representation can be *accessed*. In our case, the whole identifier is unique in all contexts and serves to distinguish two smart things. The structure of an identifier is shown in Figure 4.5.

Concept 12 (Identifier) *An identifier consists of two parts. One part, the home address, denotes the location of a representation's access point in the virtual world. The name of every virtual location of a representation's access point is unique within the virtual world. It also represents the namespace for the other part of an identifier: the name. A name only needs to be unique within its namespace.*

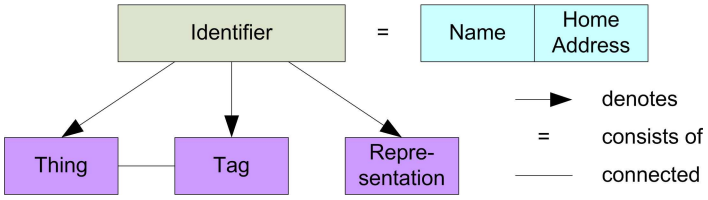

Figure 4.5: Structure of an identifier

Although the description does not include the structure of a name, there is no requirement to encode information into the name, except for at most a *serial number* and *type* information. The structure of the home address is given by the underlying naming scheme of the network technology deployed. The home address might indirectly also denote the producer of a thing, if the producer provides the access point for a representation. One example that builds on Web Services and that refers to a bottle with a serial number could be: *"bottle123@bottlemaker.com/home.asmx"*. In this example, *"bottle"* refers to the type, *"123"* is the serial number, *"@"* is used as delimiter and *"bottlemaker.com/home.asmx"* is the URL of the virtual location of the representation's access point and indirectly also denotes the producer of the bottles: "Bottlemaker". Although we referred to a type classification in the previous example, we explicitly do not demand a certain type system nor do we need one. It is up to the participants of a supply chain to agree on identifiers that contain type information. This also means that these participants have to choose a suitable type system for their purposes. The following corollary shows that with the previous definition, an identifier is unique in all contexts:

Corollary 3 (Uniqueness) *An identifier is unique in all contexts, since the home address as a namespace is unique for every virtual location of a representation's access point and the name is unique within such a namespace.*

4.2.2 Locations

Although an identifier denotes both parts of a smart thing: the tagged thing in the real world as well as the representation in the virtual world, the location information is different for each. In our model, every entity of the real world and every entity of the virtual world has an *address* in the real, or the virtual world (see Figure 4.6). In this case, an address is the name of the *location* of the entity, so that the address is the answer to the question: *"Where is a certain entity?"*. Since we want to state the location of a smart thing in the world or within another smart thing, a *physical location* can refer to the real world or to a smart thing, e.g. a smart bottle can be in the smart fridge that in turn is in the kitchen – the actual location model is defined in the next section. On the other hand, we also have to state the location of a representation in the virtual world, i.e. its *virtual location*. The home address is the name of the service, which knows the *current address* of a representation. The current address itself is the name for another service that is currently hosting a representation. A service name in turn might also contain an address of the computer where the service is hosted. The general definition for a location is as follows:

Concept 13 (Location) *A location can either be a physical location or a virtual location. Every entity of the real world has a physical address. Every entity of the virtual*

world has a virtual address. An address of an entity is the name of the location where the entity resides.

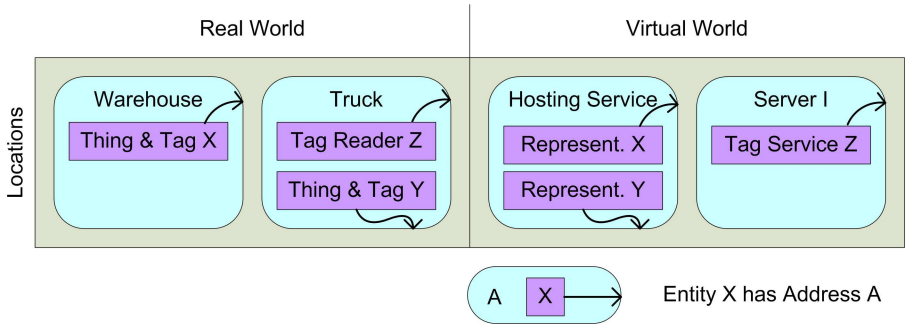

Figure 4.6: Locations in the real and the virtual world

While a smart thing has exactly one identifier for its thing and its representations, a thing and a representation have different addresses, since they are located in different worlds: the real world and the virtual world. Building on the definition of a location in general, we can also describe a virtual location.

Concept 14 (Virtual location) *Every representation has exactly one home address that refers to a "home service". Every representation has exactly one current address that refers to a "hosting service". Every service has an address that refers to the computer where the service is hosted. The addressing scheme depends on the network technology deployed.*

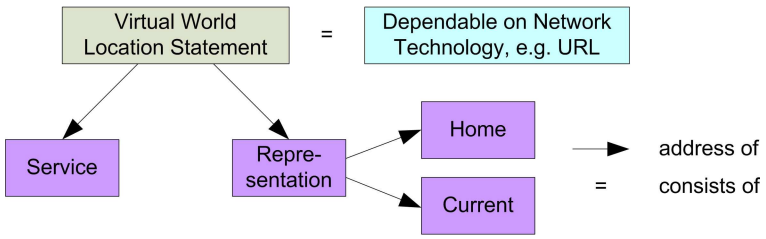

Figure 4.7: Structure of a virtual location statement

In the previous example: *"bottlemaker.com/home.asmx"* is the name for the service that knows the current address of a representation. This service is hosted on a computer with the DNS name *"bottlemaker.com"*. The structure of a virtual location statement can also be seen in Figure 4.7.

Concept 15 (Physical location) *A thing and its tags have the same address. The address can refer to the location within the real world or to the location within another smart thing. The address relies on a location model.*

The description also states that a thing and a tag always have the same address, i.e. that actually only the address of a tag can be determined and this information is then associated with its thing. Figure 4.8 shows the structure of a real world location statement that is used to describe the location of a tagged thing.

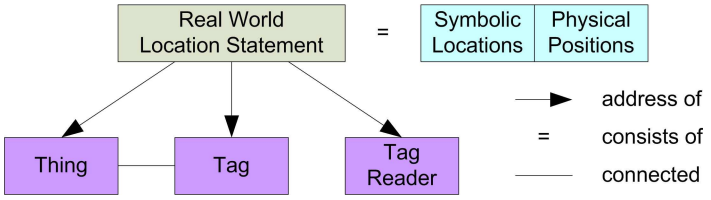

Figure 4.8: Structure of a real world statement

4.2.3 Location model

In comparison with the virtual world, where the "location model" is given by the underlying network technology, the location model for the real world needs to be defined. We need such a location model to state the address of a tagged thing in the real world, since most applications are location-dependent. An address refers to the name of the location where the tagged thing or any other entity of the real world currently resides. The address of a tag reader, which is an entity of the real world, can be sufficient if proximity is used, which in general, has been introduced in Chapter 3. In such a case, all tags in the coverage space receive the tag reader address as their address. For our purposes, we develop a hybrid model, which has also been introduced in Chapter 3, for two reasons: the first reason is that we want to support different localization technologies that are designed to work either with *physical positions* or with *symbolic locations* and the second reason is that applications might depend on physical positions, symbolic locations or both.

Generally, we distinguish two high-level concepts: static locations, (LOC), which represent spaces in the real world, and mobile smart things (ST). By LOC we denote the set of all modelled locations in the real world and by ST we denote the set of all modelled smart things, as can be seen in Figure 4.9. A location can contain other locations and smart things. A smart thing can contain other smart things.

We can describe a location in two ways. On the one hand, we can use symbolic names to denote a location. These location names are elements of the world symbolic location model (WSYM). On the other hand, we can describe a location by its physical positions. These coordinates are part of the world physical position model (WPP).

Both models also exist for smart things, since smart things can contain other smart things. In such a case, it is possible to state the location of a contained smart thing in its containing smart thing. On the one hand, symbolic location names of smart things are part of the smart thing symbolic location model (STSYM). On the other hand, physical positions within a smart thing are part of the smart thing physical position model (STPP).

A namespace of a location or smart thing is a subset of WSYM or STSYM that contains all the symbolic location names that refer to that location or smart thing. A coordinate system of a location or smart thing is a subset of WPP or STPP that contains all the positions that refer to that location or smart thing.

which actual location models are used, depends on the particular implementation, with only one restriction – all four models must be connected to each other, so that the following conditions hold:

- The symbolic location name of a smart thing within another smart thing must be appendable to the symbolic location name of the containing smart thing.

- Every symbolic location name can be mapped to a set of physical positions and every physical position can be mapped to a set of all containing symbolic location names.
- A physical position within a smart thing can be mapped to a physical position in the world physical position model,
- and vice versa: a physical position in the world physical position model can be mapped to a physical position in the smart thing physical position model.

To prevent confusion about all the models, we call the meta location model that consists of the four other location models the *Smart Things Infrastructure Location Model* (*STILM*), which is shown in Figure 4.9. Later in this section, we will also give an example showing the four location models of STILM, so that their meaning and their interdependencies will become clear.

Concept 16 (STILM overview) *STILM consists of four location models. One location model (WSYM) describes the symbolic location of a physical entity within the real world. Another symbolic location model describes the symbolic location of a physical entity within a smart thing (STSYM). One physical position model (WPP) describes the physical position of a physical entity within the real world. And finally, another physical position model (STPP) describes the physical position of a physical entity within a smart thing. A namespace of a location or a smart thing is a subset of WSYM or STSYM that refers to the symbolic location name of that location or smart thing. A coordinate system of a location or a smart thing is a subset of WPP or STPP that refers to the physical positions of that location or smart thing.*

Now, we know that a location has symbolic location names in WSYM and physical positions in WPP. Analogously, a smart thing has symbolic location names in STYSM and physical positions in STPP. In the following description, four functions describe the dependencies of locations, smart things and the four models.

Concept 17 (Basic functions) *We require four functions that associate symbolic location names as well as physical positions with every smart thing and every location. For every smart thing a there is a function p_a that returns a set B of all symbolic location names that represent the smart thing: $B = p_a$ with $B \subseteq STSYM$. Another function q_a returns a set B of all physical positions the smart thing a possesses: $B = q_a$ with $B \subseteq STPP$. There are also two corresponding functions for every location. The first function r_a returns a set B of all symbolic location names that represent the location a: $B = r_a$ with $B \subseteq WSYM$. The second function s_a returns a set B of all physical positions the location a possesses: $B = s_a$ with $B \subseteq WPP$.*

A location or a smart thing can contain smart things. Since every smart thing has its own namespace, we have to describe how symbolic location names can be transformed from one namespace to another smart thing' namespace or to a namespace of a location and vice versa.

Concept 18 (Symbolic transformation functions) *Second, we require three functions with their reverse functions that describe how symbolic location names can be appended or removed from other symbolic location names. For every smart thing a that*

contains another smart thing b there is a function f_a that maps a name c of the contained smart thing's namespace into a name d of the containing smart thing's namespace: $d = f_a(b, c)$. Thus, the reverse function f_a^{-1} maps a name of the containing smart thing's namespace into a symbolic location name of the contained smart thing's namespace: $c = f_a^{-1}(b, d)$. The function g_a with its reverse function g_a^{-1} does the same for a location a and a contained smart thing b: g_a maps a name c of the contained smart thing's namespace into a name d of the containing location's namespace: $d = g_a(b, c)$. Thus, the reverse function g_a^{-1} maps a name of the containing location's namespace into a symbolic location name of the contained smart thing's namespace: $c = g_a^{-1}(b, d)$.

Since a location can contain other locations and every location has its own namespace, we also have to describe the transformation functions that transform the symbolic location name of a location into the symbolic location name of another location.

Concept 19 (Symbolic location transformation functions) *The function h_a with its reverse function h_a^{-1} also does the same for a location a and a contained location b: h_a maps a name c of the contained location's namespace into a name d of the containing location's namespace: $d = h_a(b, c)$. Thus, the reverse function h_a^{-1} maps a name of the containing location's namespace into a symbolic location name of the contained location's namespace: $c = h_a^{-1}(b, d)$.*

In the end, we also have to describe the transformations of physical positions between the coordinate system of different smart things and the global coordinate system of the world physical position model.

Concept 20 (Position transformation functions) *Finally, we require two functions with their reverse functions that describe how physical positions of a certain physical position model can be transformed into physical positions of another physical position model. For every smart thing a that contains another smart thing b there is a function m_a that transforms a physical position c of the contained smart thing's coordinate system into a physical position d of the containing smart thing's coordinate system: $d = m_a(b, c)$. Thus, the reverse function m_a^{-1} transforms a physical position of the containing smart thing's coordinate system into a physical position of the contained smart thing's coordinate system: $c = m_a^{-1}(b, d)$. The function n_a with its reverse function n_a^{-1} does the same for a location a with the physical position d of its coordinate system, a contained smart thing b and the physical position c of its coordinate system: $d = n_a(b, c)$ and $c = n_a^{-1}(b, d)$. A function that transforms physical positions within the world physical position model is not necessary, since all locations share the same coordinate system.*

The signatures of the functions are as follows:

- $f_{ST} : \ ST \times STSYM \ \rightarrow STSYM$
- $f_{ST}^{-1} : \ ST \times STSYM \ \rightarrow STSYM$
- $g_{LOC} : \ ST \times STSYM \ \rightarrow WSYM$
- $g_{LOC}^{-1} : \ ST \times WSYM \ \rightarrow STSYM$
- $h_{LOC} : \ ST \times WSYM \ \rightarrow WSYM$

- $h_{LOC}^{-1}: \ ST \times WSYM \ \rightarrow WSYM$
- $m_{ST}: \ ST \times STPP \ \rightarrow STPP$
- $m_{ST}^{-1}: \ ST \times STPP \ \rightarrow STPP$
- $n_{LOC}: \ ST \times STPP \ \rightarrow WPP$
- $n_{LOC}^{-1}: \ ST \times WPP \ \rightarrow STPP$
- $p_{ST}: \rightarrow STSYM$
- $q_{ST}: \rightarrow STPP$
- $r_{LOC}: \rightarrow WSYM$
- $s_{LOC}: \rightarrow WPP$

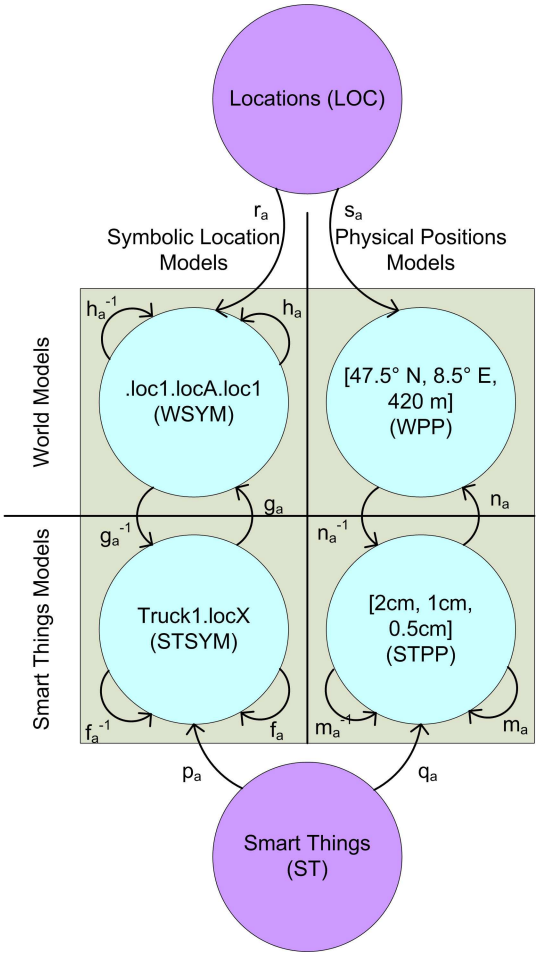

Figure 4.9: Composition of STILM

With the support of the nine functions and their five reverse functions, which are also shown in Figure 4.9, we can determine the following corollary:

Corollary 4 (Transfomations) *Every physical position in every coordinate system can be* transformed *into every symbolic location in every namespace and vice versa. The application of the functions f_a, g_a, h_a and their reverse functions allow us to express any symbolic location name of a certain namespace in any other namespace. Analogously, the application of the functions m_a, n_a and their reverse functions allow us to express any physical position of a certain coordinate system in any other coordinate system. Finally, the four functions p, q, r and s allow us to switch between the physical positions and the symbolic location names.*

In a nutshell, a location always can be expressed as a combination of location names and the containing physical positions. Independently of which location model is used, any transformation between the four location models is possible. Coming from the definition of STILM, which consists of four location models, we can now define the more detailed properties of STILM:

Concept 21 (Further STILM properties) *Every namespace has a* reference point. *A location statement contains symbolic location names and the containing physical positions. An* absolute location statement *refers to the fact that the symbolic location names and the physical positions are given relative to the absolute reference points. A* relative location statement *refers to the fact that the absolute reference points are not used. A location can contain other locations. A location can contain physical entities.*

Although we do not denote the four actual location models for STILM, we propose four models that seem to be appropriate to support most applications. First, we take a look at the *world location model*. Since we have to model every point within, on and above the earth, we use the *WGS 84* standard that is commonly used, also by the widespread GPS system, to describe the physical position of an entity on earth. In fact, we do not support other physical position models, but they are normally easily convertible into the WGS 84 standard. In the WGS 84 standard, positions are specified by latitude, longitude and altitude, so that every point on, in and above the earth can be expressed.

The symbolic model consists of a graph, where every node of the graph represents an arbitrary three-dimensional space and a descendant node is a true subset, concerning the three-dimensional space, of its parent nodes. It is not necessary that all descendant nodes of a node make-up a partition of the node. Every node possesses arbitrary abstract names that are unique within its namespace, which consists of all the names of the parent's descendant nodes, except the root node that is referred to as ".". An absolute statement of a node's symbolic location is given by the path from the root to the node by appending the node's names separated by ".", except for the root node that is appended without a separator, since its name equals the separator – examples would be: ".europe.switzerland.zuerich.zuerich.kreis6.haldeneggsteig4" or ".eth.zuerich.ifw". Although all locations start with a separator, it is not redundant, since it is used to distinguish absolute and relative locations – an example for a relative location is "kreis6.haldeneggsteig4" that does not start with a separator and that could theoretically also be the part of the following absolute location: ".europe.switzerland.zuerich.winterthur.kreis6.haldeneggsteig-4".

Concept 22 (World location model) *The hybrid location model consists of the WGS 84 standard and a symbolic model. The symbolic model has one root node. Every node has at least one name and covers a three-dimensional space. The root name is ".", and*

the other names are unique within the descendant's names of the parent node and they do not contain the root name. An edge states that the child's coverage space is a real subset of the parent's coverage space. Location statements are created by appending subsequent nodes with "." as a separator, except the root node that is appended without a separator. A location that starts with the root name is called absolute; all other locations are called relative.

Besides the world location model, we also propose a *smart thing location model*. The usage of the WGS 84 standard is not appropriate for a smart thing itself, since it is not a sphere. Here we use a *standard Cartesian coordinate model* with three standard axes (x, y and z). The alignment of the Cartesian coordinate system is given by the standard vector of the x axis at the reference point of the smart thing. The construction of the symbolic location names is the same as with the symbolic world location model except the fact that the root name is a colon followed by the identifier of the smart thing and that after the root name a separator has to be used, since the root name is not the separator.

Concept 23 (Smart thing location model) *The hybrid location model consists of a standard Cartesian coordinate system and an adapted symbolic location model from the last definition. The Cartesian coordinate consists of three axes: x, y and z. The x-axis is given by the standard vector that starts at the reference point of a smart thing. The symbolic location model is the same as above except the fact that the root name is the ":" sign followed by the identifier of the smart thing and that a separator is used after the root name.*

The nine functions and five reverse functions demanded for STILM are given by the world location model and the smart thing location model. Two symbolic location names can easily be appended with a separator, and the transformation between two physical location coordinate systems, i.e. between two Cartesian coordinate systems or a Cartesian coordinate system and the WGS 84 model, can also be easily made. Since a location name always refers to a set of containing physical positions, the transformation between the two world location models is also given. A complete location graph that contains our building, office and fridge as well as two addressing schemes of our building can be seen in Figure 4.10.

Besides the actual location model, we also need to define the format for measurement of location data and its transmission between the entities in our model. The following definition follows a pragmatic approach:

Concept 24 (Localization datum) *Every localization module has a name and logs values with a timestamp. A* localization datum *consists of three fields: the name of the* localization module*, the* localization value *and a* timestamp *when it was recorded.*

Since we want to support different localization technologies, we first have to mention which localization module was used, followed by the actual localization value and the time this measurement was recorded, which is shown in Figure 4.11. (GPS; (47.5,8.5,420), 2003-Dec-25-18:45:00) refers to a GPS measurement that was made at Christmas in Zurich and that localizes an object exactly at 47.5° North, 8.5° East and 420m height.

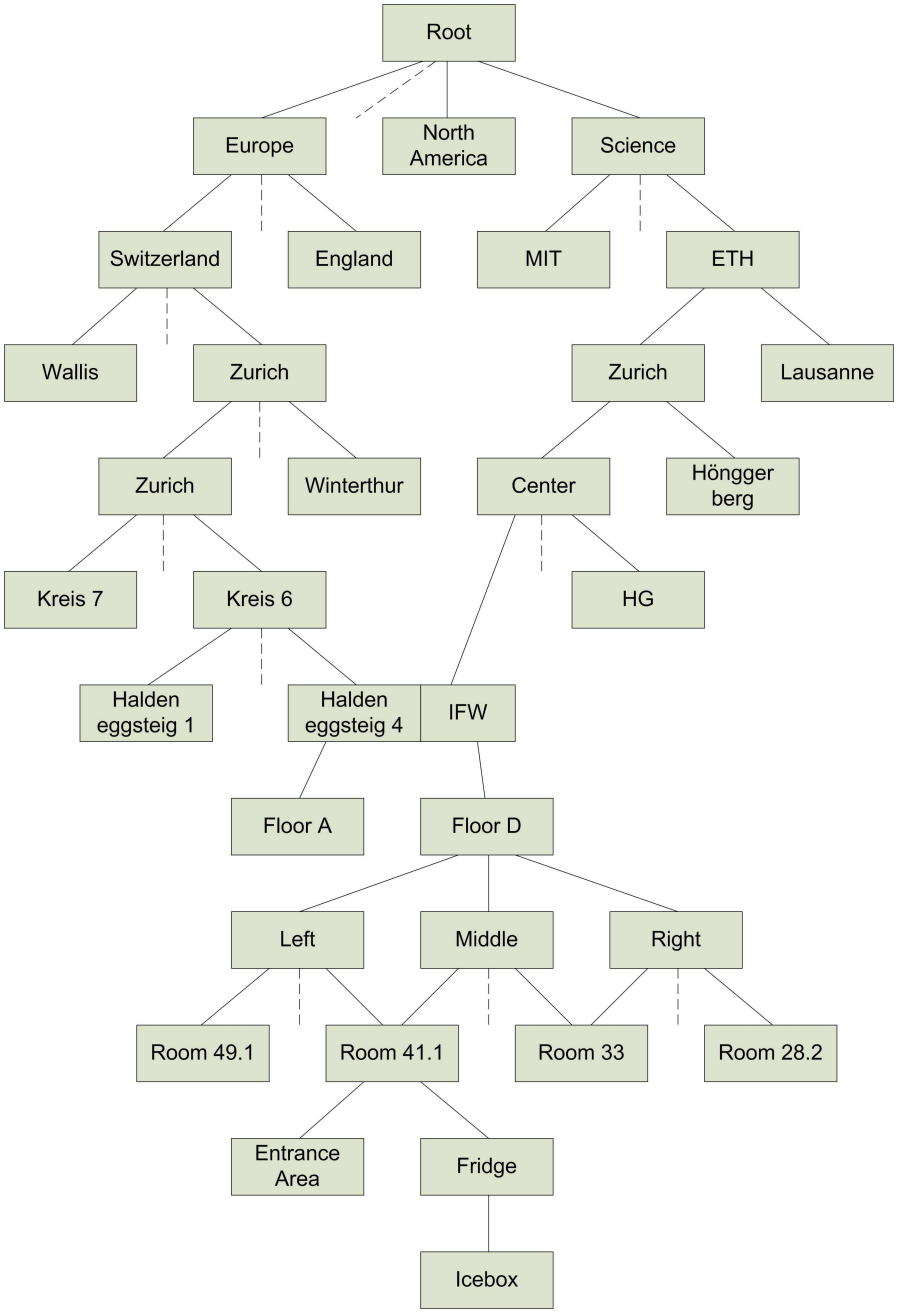

Figure 4.10: Example location graph

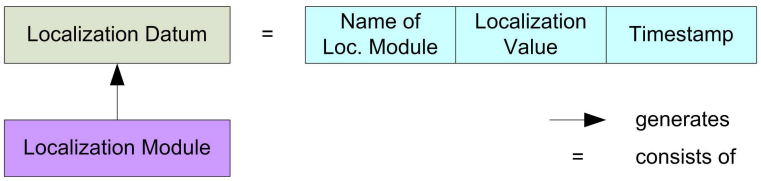

Figure 4.11: Structure of a localization datum

Summary We differentiate between static locations in the real world that can contain other static locations and mobile smart things. Mobile smart things in turn can also contain other mobile smart things. In order to describe the location of a smart thing, we have to consider two cases:

- a mobile smart thing is within a static location, e.g. a smart fridge is located in the kitchen
- a mobile smart thing is within another mobile smart thing, e.g. a smart milk bottle is located in the smart fridge

Besides the distinction between mobile smart things and static locations, we also differentiate between:

- symbolic location names, such as eth.zurich.ifw.d41 and
- physical positions, such as 47.5° North, 8.5° East and 420m height.

This differentiation is driven by two reasons: first, localization technologies return either symbolic location names or physical positions, and second, applications depend on either symbolic location names, physical positions or both.

Since both differentiations are orthogonal, we can describe the location or position of a smart thing in four ways:

- a mobile smart thing is within a static location that is described by a symbolic location name, e.g. the mobile smart thing "fridge1@eth" is located in the static location "eth.zurich.ifw.d41".
- a mobile smart thing is within a static location that is described by a physical position, e.g. the mobile smart thing "fridge1@eth.ch" is located at the physical position "47.5°,8.5°,420m".
- a mobile smart thing is within another mobile smart thing that is described by a symbolic location name, e.g. the mobile smart thing "milkbottle1@eth.ch" is located with the symbolic location name ": 'fridge1@eth.ch'.door" of the mobile smart thing "fridge1@eth.ch".
- a mobile smart thing is within another mobile smart thing that is described by a physical position, e.g. the mobile smart thing "milkbottle1@eth.ch" is located at the physical position "(2cm,4cm,6cm)" of the mobile smart thing "fridge1@eth.ch".

In a nutshell, STILM has the following requirements:

- location models for locations and mobile smart things
- location models that use symbolic location names and physical positions
- functions that can transform location statements and position statements between the four location models

4.2.4 Sensor and actuator data

Besides the identification and localization of a smart thing, it also has to control sensors and actuators. The data format for both can be kept very simple: we consider only simple sensors and actuators that return, or accept only simple values, i.e. it is not intended to encode complex information structures in a value. A *sensor value*, for example, could be the temperature in Kelvin. A collection of the temperature values of the last hour with a recording frequency of one minute that are all encoded into one sensor value instead is not within the scope of this model. The same holds for an actuator. An *actuator value*, for example, could be the brightness of a lamp in lux that an actuator has to adjust. A plan when to turn the lights on and off within the next hour encoded into one actuator value instead is also not within the scope of the model. The basic idea is to focus the control logic in the representation to allow for simple sensor, actuator and tag modules.

Concept 25 (Sensor datum) *Every sensor has a name and logs values with a timestamp. A* sensor datum *consists of three fields: the name of the sensor, the actual sensor value and a timestamp when it was recorded.*

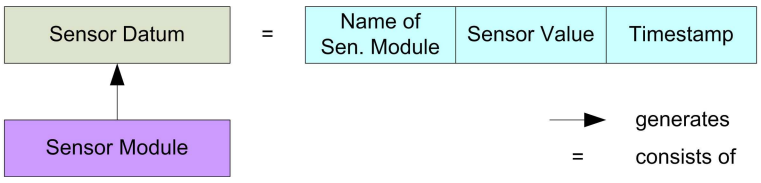

Figure 4.12: Structure of a sensor datum

Concept 26 (Actuator datum) *Every actuator has a name and accepts actuator values. A* actuator datum *consists of two fields: the name of the actuator module and the current actuator value.*

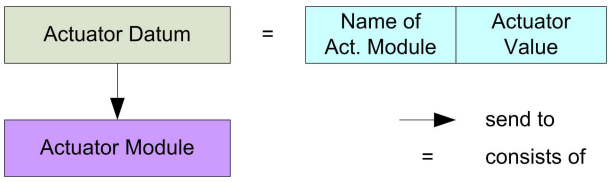

Figure 4.13: Structure of an actuator value

Both definitions are illustrated in Figure 4.12 and 4.13. Since an actuator value has to be executed instantly, a timestamp for the execution is not necessary. We neither define the naming scheme, nor do we define how the values are interpreted. With both these definitions, we only want to provide the necessary infrastructural means to control sensors and actuators, so we do not define a terminal list of names and how the actual values have to interpreted here, since this task is better performed by a group of sensor and actuator experts – the same approach has been chosen in Jini with the attribute concept that describe a Jini service: Jini provides only the basic mechanisms to support attributes, instead of defining the actual categories for describing a Jini service.

4.3 Concepts for smart thing entities

A smart thing consists of a thing and a representation. Since we do not manipulate the actual thing, apart from attaching a tag to it, the concept "thing" needs no further consideration, so that we only need to refine the descriptions of the representation and the smart thing in general in the following.

4.3.1 Representation

From the above definitions, we already know that a representation possesses an identifier, which is the same for the tagged thing, and that a representation has two addresses. One address, the home address, refers to the name of the service that knows the current address of the representation. The current address in turn refers to the name of a service that actually hosts the representation. The task of a representation is to encapsulate the additional functionality of a smart thing within the virtual world, so that a representation must be realized as some piece of software. We do not demand the actual implementation of a representation, e.g. as a web service or as a CORBA object, but we do demand the differentiation between *static code* and *dynamic state data* to efficiently support *migration* of representations within the virtual world. The efficiency gain results from the fact that the same type of smart things typically but not necessarily possess the same static code, so that the code only needs to be migrated once, when many representations of the same type have to be migrated. However, the dynamic state data needs to be migrated for each representation separately. Since we envision myriads of smart things with their representations, the actual implementation has to be realized highly efficiently concerning the usage of computer memory and the required time to instantiate and to execute a representation. At the same time, the implementation should allow a range of representations to be realized, varying from a very passive behavior to a very active one. On the one hand, a very simple representation, for example, may have no real dynamic state and may only answer requests with static and pre-programmed responses. On the other hand, a powerful representation may actively control its sensors and actuators, store its whole history and cooperate with other smart things and applications. The actual description of a representation can be kept short, since we have already given many implementation hints above that are not part of the description. Figure 4.14 shows the structure of a representation.

Concept 27 (Representation) *A representation is a piece of software that encapsulates the additional functionality of a smart thing. It consists of static code and dynamic state data. The migration of a representation is achieved by migrating the code and the state data separately. Representations of the same type of smart things may share the same static code. A representation has two addresses: the home address refers to the "home service" that returns the current address, where the representation is hosted by a "hosting service". A representation possesses two "communication channels" that allow for communication with the sensor and actuator modules on the tag. A representation is able to communicate with other representations or with "applications". A representation has the same identifier as its associated thing.*

The home service, the hosting service, the communication channels and the applications will be introduced later. The representation itself provides a lean generic communication protocol, since its functionality is highly application-dependent, so that only those

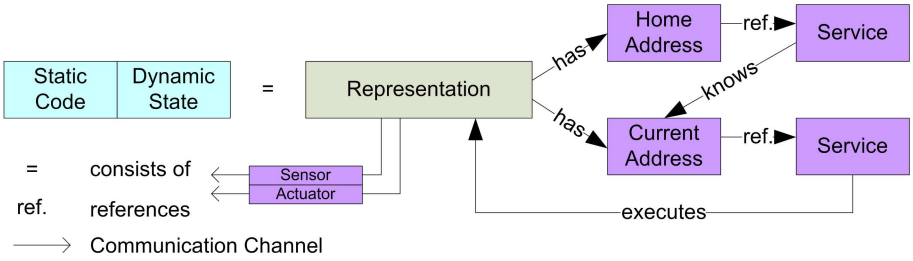

Figure 4.14: Structure of a representation

functions are defined that are absolutely needed by the infrastructure. This approach is comparable to the object concept in object oriented programming languages: since the behavior of an object is highly application-specific, the object class itself provides only methods of managing the object, e.g. providing a hash code for data storage, means for synchronization, a human readable representation, information about the type and so on. Here, a representation provides access to its identifier, its current symbolic locations and its physical positions, among other data, and it provides information about which other protocols it supports. With the last function, every smart thing or application is able to check whether a smart thing provides a certain function by asking the smart thing for its communication protocol.

We already mentioned the *history* concept, which is a rather general concept. Every past state data can be part of a smart thing's, i.e. of a representation's history. That means that the current state and the static code of a representation cannot be part of its history.

Concept 28 (History) *Only past state data can be part of a representation's history. The history of a smart thing refers to the history of its representation.*

We explicitly do not demand that a smart thing must implement the history concept to allow for simple representations.

4.3.2 Smart thing

Some concepts rather refer to the smart thing as whole instead of to the representation only, although it is implemented by the representation. Three such concepts are binary relations of smart things that are needed so frequently that they should be supported by the infrastructure:

- *Containedness*
- *Composition*
- *Neighborhood*

The simplest concept is containedness, which refers to the fact that one smart thing, i.e. a tagged thing, is physically fully contained by another smart thing, e.g. a supply part is fully contained by a truck. One reason for this relation is that properties of the containing smart thing can be handed down to the contained smart thing, such as the location or sensor values – the temperature, for example. Another reason for

it is that some applications might depend on such structured information, e.g. that a smart transport container contains certain smart supply parts. A last reason is that containedness is the prerequisite for a location model for smart things, as required by the general STILM.

Concept 29 (Containedness) *Containedness is a binary relation between two smart things. It is given when the dimensions of one smart thing are fully within the dimensions of another smart thing. If one thing is only partly within another smart thing then a third party, e.g. the producer of a smart thing, has to provide a logic that decides on that relation. The relation is asymmetric and transitive and not reflexive.*

Another relation that is similar to containedness is composition. If a smart thing is composed of other smart things then the first also contains the others. The difference is a logical: with composition we express that one thing structurally consists of other smart things. If such a smart thing that is part of another smart thing is missing, then the other smart thing changes its character. One example shows the difference between composition and simple containedness: a cupboard contains cups and consists of a door. If we take out some cups, the cupboard still remains a cupboard, but if we demount the door, the cupboard becomes a shelf. Besides the pure logical differentiation between the two concepts, it also has some practical aspects. While the containedness relation refers mainly to the usage of a product and is highly dynamic, the composition relation, in contrast, is more static and refers to the production, disposal, and maintenance process that occur rather rarely during a product's lifecycle, e.g. it might be beneficial to check after a truck has arrived at its destination whether smart things that should be contained by the truck are still in the truck, but it is unlikely that one of its tires is missing. Another aspect refers to the passive and active behavior of a smart thing: when an active smart thing becomes part of another smart thing it makes sense that this smart thing becomes passive to avoid too much communication between all the parts. A car, for example, consists of several thousand parts and if all of these parts become smart and every part wants to communicate with every other part at the same time, this might exhaust the resources, such as communication bandwidth. Structures like containedness and especially composition provide the means that can also be used for the communication between smart things, i.e. the root of the containedness tree may coordinate the communication within its tree.

Concept 30 (Composition) *Composition is a specialization of containedness. Two smart things or a third party have to decide whether two smart things are part of each other.*

Again, whether a smart thing is part of another smart thing or is contained by a another smart thing is not obvious, but rather application-specific. Either an application determines whether a smart thing is contained by or is part of another smart thing at all, or the location determines it, if the location of one smart thing is within the dimensions of another smart thing. Within an assembly process, for example, an assembly application might decide on which smart thing is part of another smart thing and configures the smart things accordingly.

Besides the two aspects that two smart things are in a relation if one contains another, another relation makes use of the *collocation* of two smart things that we call the neighborhood concept: such a relation postulates that two smart things are neighbors,

which means from a symbolic location's point of view that they are close to each other. It is more likely for two smart things to collaborate when they are collocated: if the temperature sensor of a smart thing breaks, it can ask one of its neighbors for the current temperature, since the temperature for two collocated smart things should be approximately the same. The neighborhood relation can be absolute or relative: an absolute neighborhood refers to all the smart things that are in the smallest location that contains both smart things. A relative neighborhood means two smart things are neighbors relative to a certain location. Figure 4.15 shows two sets of nodes that are neighbors. An example for a neighborhood relative to the location kitchen is the following: a mug is a neighbor of the chair, but the mug is not a neighbor of the cupboard, since it is contained by the cupboard.

Concept 31 (Neighborhood) *Neighborhood is a binary relation between two smart things. It is symmetric and transitive but not reflexive. Neighborhood means that two smart things share the same symbolic address. Absolute neighborhood means that both smart things are located at the smallest symbolic location, whereas relative neighborhood refers to a certain symbolic location where both smart things are located. Two smart things of the composition or the containedness relation cannot be in the neighborhood relation.*

While the containedness and neighborhood relation can be automatically determined by the infrastructure, the composition relation needs to be explicitly stated by the programmer, since we do not rely on a global all-embracing ontology. However, with the three definitions given above, we can draw some conclusions about each relation and the connection of these relations.

Corollary 5 (Smart thing relations) *Smart things that are in the composition relation are also in the containedness relation. Smart things that are in the neighborhood relation cannot be in the containedness relation and vice versa. The containedness relation, and therefor the composition relation, make-up a rooted tree.*

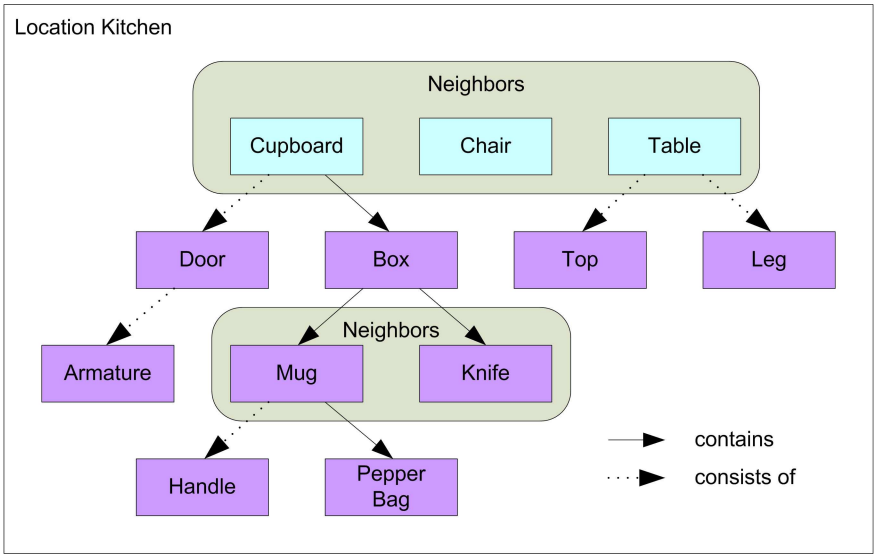

Figure 4.15: Example of smart things' relations

At this point, we can use the above definitions to define more precisely what a smart thing is – besides the previous definition that a smart thing consists of a thing and a representation.

Concept 32 (Smart thing) *A smart thing is mobile, it can contain other smart things, it can be composed of other smart things, and it has its own location model.*

It is important to note that a smart thing is mobile and possesses its own location model, so that a truck also has to be modeled as a smart thing in our model, in contrast to a static room in a building. Since the association of the room with an identifier and the physical positions is static, it brings no benefit to dynamically identify and localize the room since the static result is already known. Unlike a truck, a room would be modeled as a location in the world location model. Thus, static objects such as buildings or streets can only be described by a location from the world location model. Mobile smart things are modeled as smart things that additionally have a smart things location model. While a truck as a smart thing, which contains other smart things, can communicate with them, a room modeled as a location cannot communicate with its containing smart things.

A question that might arise is, how are sensors and actuators modeled that are installed in a room? In such a case, each sensor and each actuator must be modeled as a smart thing, since they are in principle mobile, i.e. they can be demounted and mounted again in another room, which would not be possible with a "logical" room. A temperature sensor, for example, must be modeled as a smart thermometer. It is not necessarily a disadvantage that a room cannot communicate with its smart things, since we intend to put the corresponding application logic for a static region into an application that is responsible for that static region. Using this modeling scheme, we expect to cover almost all reasonable real world scenarios. One might find a pathological real world scenario that cannot be naturally modeled with our model, but the complexity of the real world is too high to consider the pathological scenarios as well.

4.4 Concepts for infrastructure entities

As a complement to the previous section, we now introduce the concepts of the infrastructure, which consists of the tag detection system and the managing services.

4.4.1 Tag

The tag as part of the physical world builds the connection with a thing by attaching the tag to the thing. To support the four basic abilities, i.e. identification, localization, sensors and actuators, a tag defines four corresponding groups of modules that cover the necessary communication for it. At minimum, a tag needs to define an *identification module* that covers two tasks: first it has to store an identifier and second, the module must be able to transmit the identifier to the service infrastructure. The other three module groups are optional. One of them, the second module group, contains *localization modules*, e.g. a GPS module that measures its own location through the support of at least one localization technology, and that transmits such localization data to the representation. The next module group, the *sensor modules*, e.g. light sensors, are similar to the location modules: they must measure sensor values and transmit sensor data to the representation. The last module group, the *actuator modules*, e.g. LEDs, must be able to receive actuator data coming from the representation and to execute these "commands". A direct connection between the tag and the representation is not necessary, so that there might be other entities that intercept the communication between the tag and the representation. A tag with its modules can be seen in Figure 4.16

Concept 33 (Tag II) *A tag is a software/hardware unit that is attached to a thing and consists of four modules. The software part is optional. The identification module is obligatory and is able to transmit its stored identifier. The localization module is optional and is able to transmit measured localization values as localization data. The sensor module is optional and is able to transmit measured sensor values as sensor data. The actuator module is optional and is able to execute received actuator commands. A tag belongs to a hardware-specific class of tags.*

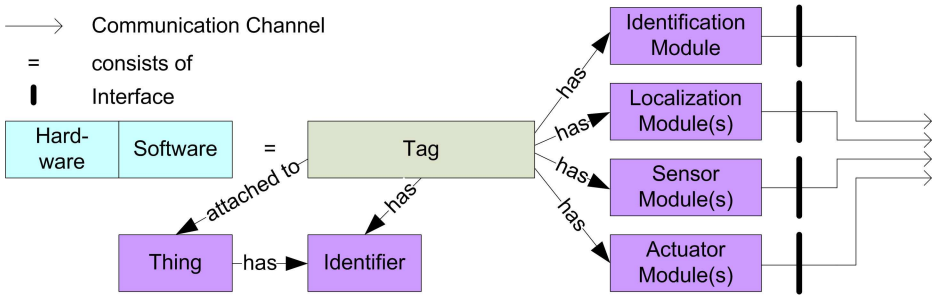

Figure 4.16: Structure of a tag

The definition states that the software part is optional, since bar codes, for example, are only available as visual signs that are unable to process software instructions.

4.4.2 Tag reader

The tag reader as part of the physical world and the infrastructure is also the entity in the physical world that builds the connection to the virtual world. Its main task is to detect the hardware-specific tags within its coverage space. It is recommended that the coverage space is also represented as a location in the location model. The tag reader optionally provides a localization module, e.g. one that uses lateration together with some other nearby tag readers, that determines the physical position of the tags within its coverage space and transmits a localization datum to the representation. As part of the path between the tag and the representation, the tag reader has to forward the identification, localization, sensor and actuator data. The structure of a tag reader is shown in Figure 4.17. A tag reader can be installed in the static world or in a dynamic smart thing. The latter can be classified in real dynamic means of transport, such as trucks, freighters, or airplanes, and more static containers, such as fridges or boxes.

Concept 34 (Tag reader II) *A tag reader is a hardware/software unit that is installed in the world or in a smart thing and that detects hardware-specific tags within its coverage space. The coverage space is also represented as location in the location model. It optionally possesses localization modules that measure the physical positions of the tags and is able to transmit the corresponding localization data in the direction of the representation. It forwards the identification, localization, sensor, and actuator data between the tag and the representation.*

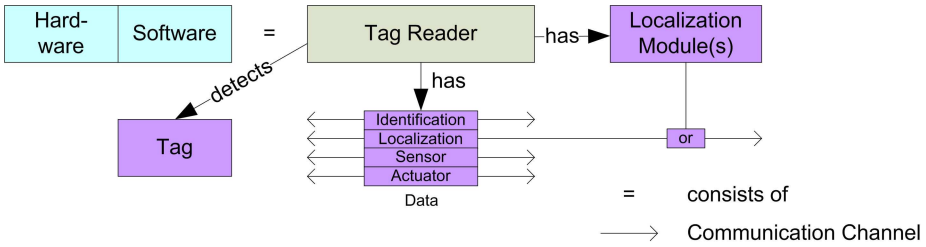

Figure 4.17: Structure of a tag reader

4.4.3 Tag detection service

The tag detection service is the last part of a hardware-specific tag detection system that also consists of the tags and the tag readers. It is part of the virtual world and the infrastructure. Together with the tag reader, it builds the bridge between the real world and the virtual world. Its main task is to control a tag reader. Besides the localization modules of the tag and the tag reader, the tag detection service might also provide a localization module that can be regarded as a default localization module if neither the tag nor the tag reader provide a localization module. This localization module just provides the symbolic name of the covered space of the tag reader. As part of the path between the tag and the representation, the tag detection service has to forward the identification, localization, sensor and actuator data. Figure 4.18 shows the tag detection service and the other entities related to it. The default location manager will be described next.

Concept 35 (Tag detection service II) *A tag detection service is a service that controls a tag reader. It optionally possesses a localization module that returns the symbolic location of the coverage space of the tag reader as a localization datum to the representation. It forwards the identification, localization, sensor, and actuator data between the tag and the representation. The "default location manager" is the next entity to the representation.*

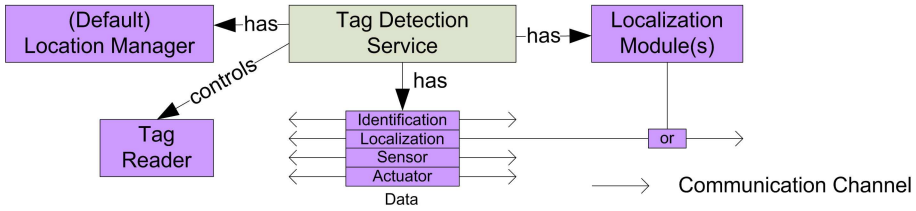

Figure 4.18: Structure of a tag detection service

In a nutshell, depending on where the localization takes place and whether physical positions or symbolic locations are used, four combinations are possible:

1. a tag detection service determines the symbolic location: an RFID reader with its coverage space defines a symbolic location,

2. a tag reader determines the physical position: a GPRS reader with other GPRS readers determines the physical position by lateration,

3. a tag determines the symbolic location: an infrared beacon receiver mounted on the tag receives the symbolic location beacon,

4. a tag determines the physical position: a GPS module mounted on the tag computes its physical position.

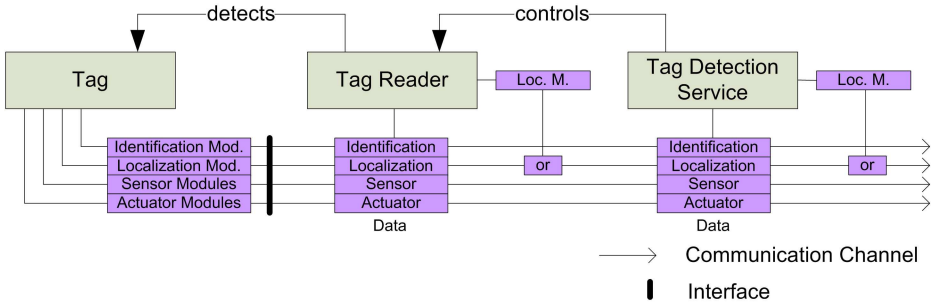

Figure 4.19: Structure of a tag detection system

Figure 4.19 summarizes the three entities of a tag detection system and shows their dependencies as well as how they are connected through communication channels.

4.4.4 Home service

The *home service* can be regarded as an access point for a representation in the virtual world, since it is referenced by the identifier of a smart thing. It is one of the managing services that uses the name part of the identifier to look up the corresponding data of the representation. Its virtual location is used for the home address part of the identifier, i.e. the home address of the identifier references the home service where the necessary data concerning the representation is stored. As already mentioned, a representation consists of static code and dynamic state data. A home service uses a *code storage service* and a *data storage service* to store both parts of a representation. The name part of an identifier of a smart thing is used to look up the code or the state data in each case. Additionally, a home service also stores the current address of the representation, i.e. where the representation currently resides in the virtual world. The structure of a home service is shown in Figure 4.20.

Concept 36 (Home service) *A home service is a managing service. It uses a data storage service and a code storage service to store the static code and the dynamic state data of a representation. It uses the name part of an identifier to look up the current address of a representation. The home address of an identifier references a home service.*

Concept 37 (Data storage service) *A data storage service is a service which uses an identifier or its name part to look up and to store the corresponding dynamic state data.*

Concept 38 (Code storage service) *A code storage service is a service which uses an identifier or its name part to look up and to store the corresponding static code.*

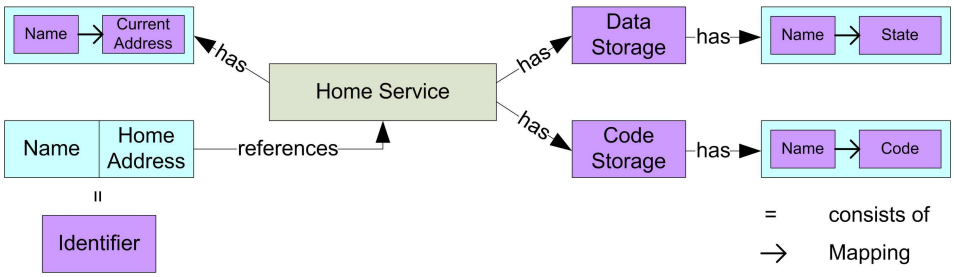

Figure 4.20: Structure of a home service

4.4.5 Hosting service

A *hosting service* is another managing service that provides the actual execution environment for the representations. If migration is supported, the hosting service that itself is hosted on a computer should be physically close to the tagged thing to enable efficient communication between the representation and the tag. Migration can be interpreted as meaning that the representation tries to follow its thing through the world and tries to be as close as possible to it. Since the home service indirectly knows the current hosting service, it is able to return the current address of a representation that indirectly references the hosting service. A representation can be executed by every hosting service so that a representation, i.e. its code and its state, can be migrated from one hosting service to another. A hosting service might use a code service to cache the static code for future use again. A data storage instead is obligatory: a representation can use this data storage to keep its data private at this hosting service. If a representation is asked to shut down at a certain hosting service, the representation can write its state data to the local data storage to keep it private or it can use the central data storage at the home service where it can be accessed from every hosting service. Besides the data and the code storage service, a hosting service also makes use of an *event bus service*. Such a service provides a local event bus for the asynchronous communication with the representations. At minimum, a representation will be indirectly notified by the hosting service after the representation has been *instantiated* with a *start event* and when a representation needs to *be shut* down with a *stop event*. Before a representation is finally shut down, the representation receives a *final grace* period that is used for two purposes: first, a representation can use the period to correctly shut down and second, during this grace period it can be migrated to another hosting service. The event bus service can also be used by the representations or other services for asynchronous communication. As part of the path between the tag and the representation, the hosting service has to forward sensor and actuator data. All entities are shown in Figure 4.21. The mentioned events and states are shown in Figure 4.22

Concept 39 (Hosting service) *A hosting service is a managing service. It is the execution environment for representations. It provides a data storage service for representations. It also provides an event bus service. At minimum, this service is used to notify the representation to start and to stop. It forwards the sensor and actuator data between the tag and the representation. It has a reference to a default hub location manager to determine the managing hub. The current address of a representation indirectly references the hosting service where a representation is executed. The default hub location manager is needed to update the home service.*

Concept 40 (Representation states) *A hosting service provides a state machine for every representation: initially, a representation is in the* not executing state. *A* manage event *triggers the transition to the* instantiation state. *After the instantiation has been completed, the representation is in the* executing state. *The hosting service triggers the representation with a start event. Triggered by an* un-manage event, *the representation gets to the grace state. The hosting service triggers the representation with a stop event. After a predefined grace period time, the representation is in the* clean-up state *or if a* re-manage event *occurs the smart thing is again in the executing state and the representation is again notified by a start event. After the clean-up by the hosting service, the representation is again in the not executing state.*

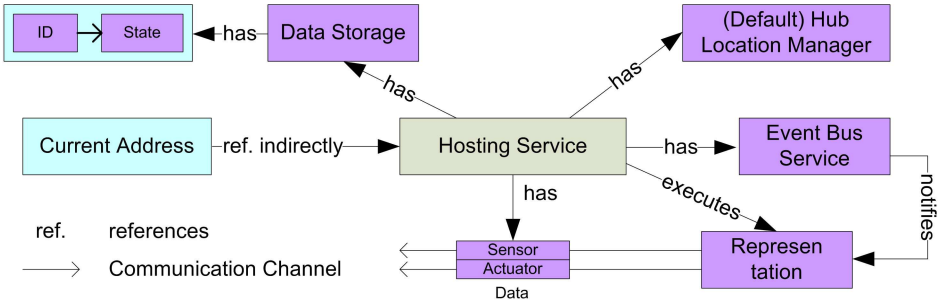

Figure 4.21: Structure of a hosting service

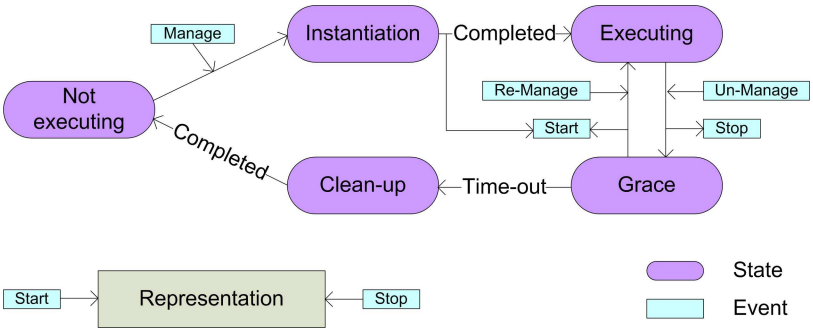

Figure 4.22: State machine of a hosting service for a smart thing

The above definition showed the concrete state transition and the event communication between the hosting service and the representation (see Figure 4.22). For a hosting service to be able to migrate a representation, a representation needs to provide the state data that can be migrated after it has received the stop event.

Concept 41 (Event bus service) *An event bus service is a service which provides the means for asynchronous communication between services and representations by providing the publish/subscribe pattern. Representations are addressed with their identifier and services are addressed by their virtual address.*

4.4.6 Location manager services

The location manager services, which are part of the managing services, are the last piece that brings all other parts together. The location manager services consist of

several location services that we also call location managers. A location manager is responsible for a certain location, i.e. for a set of physical positions with symbolic names, and has to provide location-dependent services, e.g. determining the smart things in the neighborhood of a certain smart thing. We differentiate between three kinds of location services that are mainly driven by performance considerations: a base location service, a hub location service, and a super location service. A base location service, which is on the lowest level, is responsible for managing a smart thing. It knows whether itself or one of its children manage a certain smart thing. A hub location service, one level above, is a special base location service that is referenced by the current address of a representation. A hub and its bases beneath build a *location domain*, which is illustrated in Figure 4.24. Thus, the current address of a representation only references the location domain in which the representation is managed. A home service only knows the hub service, which represents all bases beneath it, where the smart thing is handled, so that the current address does not need to be updated every time a smart thing moves to another base of that hub. Location services above hub location services are called super location services; these are not able to manage smart things and do not even know whether a smart thing is managed by one of its descendants, so that their only task is to provide anonymous, but location-dependent services. In Figure 4.24, for example, the ETH node can be asked for all the inventory of the ETH, which is a very complex query that should not be executed too often. Since we allow different symbolic location trees that might overlap, the corresponding location managers also overlap. The only restriction is that a hub cannot be a descendant of another hub within one location model, and that a hub cannot overlap with another hub. Figure 4.24 shows three hubs with their location domain and two other important facts: first, one location manager, i.e. the *Haldeneggsteig 4 / IFW* hub is referenced by two different super location managers and second, the *fridge* hub represents the location domain of a smart thing and it is referenced by a base of the world location model. In general, we can distinguish three cases of how two location services can be in relation with each other: first, a location service could be fully part of the other, second, a location service could partly overlap with another location service, and third, two location managers are disjunctive. This distinction is necessary to determine the responsible base location service for a smart thing. In the first case, the base location service is responsible for where the smart things are contained, in the third case, the smallest base that contains the smart thing is responsible. Every base references a hosting service where the representation is actually executed. If it is not possible to find an unambiguous smallest base, all smallest bases that contain the smart thing have to reference the same hosting service. Due to the overlapping character of the location manager, more than one hub could be an ancestor of a managing base. In such a case, when a smart thing moves from one base to another base, the managing hub only changes if the old base is not one of the new base's ancestors. If a smart thing was *offline*, i.e. it was not detected by any tag detection reader, and a new hub needs to be selected, every hosting service has a default hub that will be selected. Since applications for smart things are mostly location-dependent, e.g. a warehouse management system is interested in the check-in and check-out area as well as the actual storages, the application can register with location managers, which provide an event bus service to notify the applications about the entry and the exit at the monitored location. Applications can ask the location manager for the hosting service of a smart thing to contact the smart thing.

Concept 42 (Location managers) *Location managers are managing services. There*

is a one-to-one mapping between locations and location managers. A location manager can have child location managers. A location manager can ask its child nodes for the smart things in its location. A base location manager knows whether itself or one of its children manages a smart thing. It has a state machine for every smart thing: initially, the state for a representation is unknown*. When one of its children manages a smart thing then the state is* known*. The state is* managed *when a location manager manages a smart thing. In such a case, a reference to the corresponding managing hosting service is stored. Every location manager has an event bus service. Applications can register at a location manager. A location manager notifies applications about the entry and exit of smart things.*

Besides the functional aspects of location managers, they also build a hierarchy, which is described in the following.

Concept 43 (Location manager hierarchy) *A base location manager can only have base location managers as children. The location manager graph selects the smallest location managers as managing location managers. All base location managers up to the next hub will be set to known. Every base location manager has a default hosting service that is contacted if a base has to manage a new representation. Overlapping smallest bases have the same default location manager. Every base location manager supports the neighborhood concept. A hub location manager is a base location manager and can be referenced by a current address. A super location manager can have other super location managers or hubs as its children. A base location manager has to contact the home service in case a representation is not already locally managed in order to obtain the previous hub and possibly the code and the state data if no hub was managing the representation.*

A location manager with its entities is illustrated in Figure 4.23, and the mentioned states can be seen in Figure 4.25. The above definition states that all bases on the path between the hub and the managing bases are set to known and the managing bases are set to manage, and the hub is registered at the home service, so that an entity that is interested in the current hosting service first asks the home service for the managing hub. The managing hub knows all paths to the managing bases, which in turn know the hosting service of a representation. One could ask why the path from the hub to the managing base does not directly store a reference to the hosting service. The answer is that this scheme allows for local migration of the representation without notifying the above nodes.

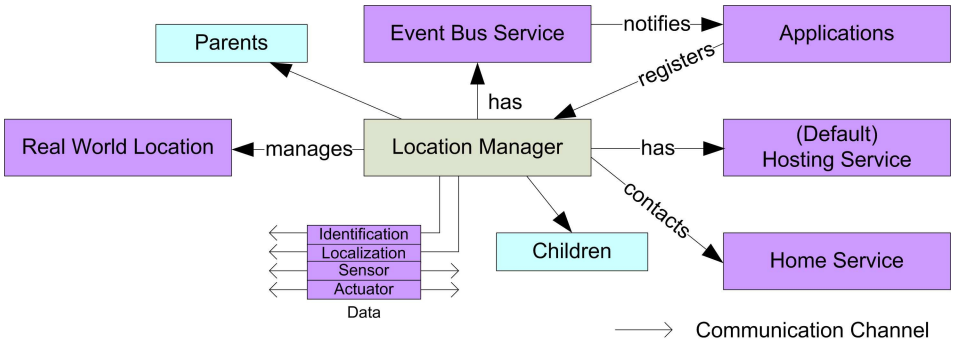

Figure 4.23: Structure of a location manager

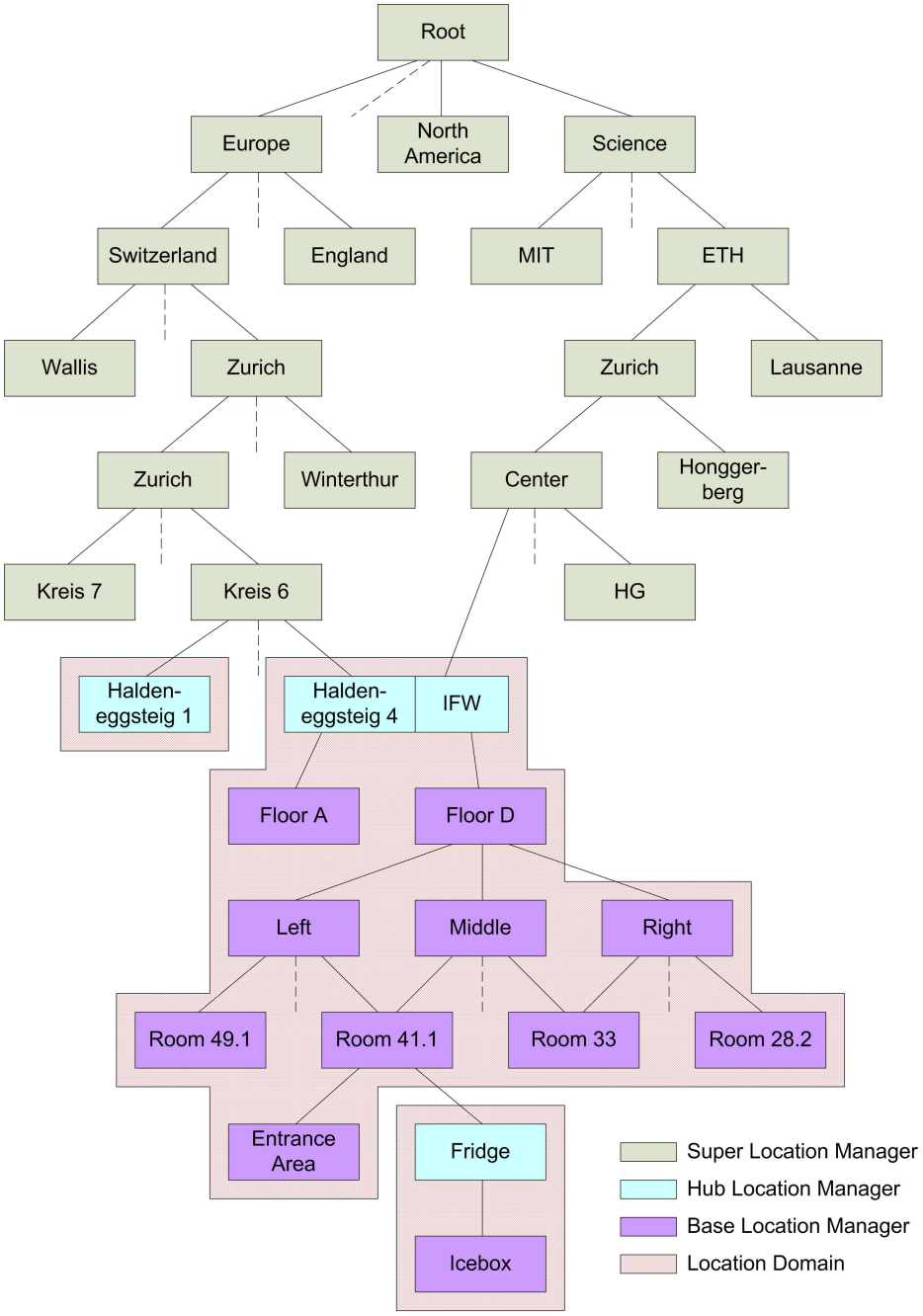

Figure 4.24: Example location manager graph

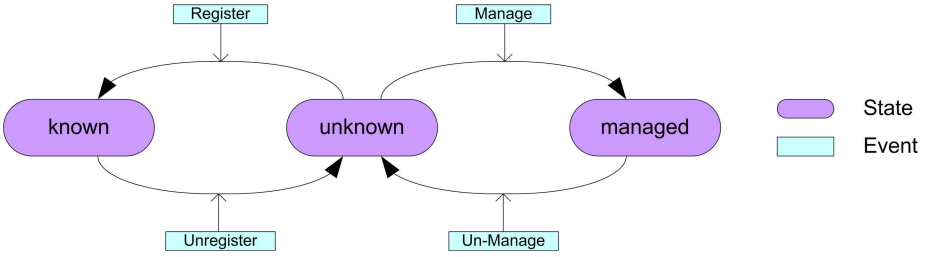

Figure 4.25: State machine of a base location manager for a smart thing

In a nutshell, location managers manage the locations of smart things in the real world and they also reference the actual executing hosting service of that smart thing. They support the neighborhood concept.

4.5 Procedure of registering a smart thing

After the presentation of each of the infrastructural entities, i.e. the tag, the tag reader, the tag detection service, the location manager services, the home service and the hosting service, that are needed to couple a smart thing, i.e. a thing with its representation, we now can describe how these entities cooperate in detail.

4.5.1 Communication channels

We already mentioned that the identification, localization, sensor, and actuator data need to be communicated coming from the tag in the direction of the representation. For this purpose, we now introduce communication channels. We have two communication channels between the tag and the location manager graph, which are used to transmit the identifier and the localization data between the tag and the location manager graph. We have another two communication channels between the tag and the representation to transmit the sensor and actuator data, as can be seen in Figure 4.26. To support these logical communication channels, the entities on the path need to store communication handles that transparently handle these issues. To enable the communication from the tag to the representation, the tag, the tag reader, the tag detection service, the contacted location manager and the hosting service each associate a representation communication channel handle with the identifier for that smart thing. Conversely, the representation, the hosting service, the contacted location manager, the tag service, and the tag reader each associate a tag communication channel handle with the identifier for that smart thing. The situation is similar with the communication channel between the tag and the location manager graph. A tag detection reader has a reference to a default location manager, which is responsible for the location that is referenced by its localization module. The tag reader as well as the tag have a communication channel handle that forwards the identification and localization data to the location manager graph.

Concept 44 (Communication channel) *A communication channel connects a tag with the location manager graph or a tag with its representation. The entities in between have to make sure that such a communication channel can be established. The communication channels between the tag and the location manager graph are used to update the location manager graph with the current location of an identified entity. The communication channel between the tag and the representation is used to exchange sensor data and actuator commands. A communication channel is opened when a tag comes into the coverage space of a tag reader and is closed when the tag leaves the coverage space. If a communication channel is open, the smart thing is* online*, otherwise it is* offline*.*

It is important to note that communication channels are only logical entities. They allow the concrete communication details of all the involved entities to be disregarded. A tag, for example, might directly contact a location manager and transmit its identifier and its location. Nevertheless, the communication is still transparently handled by the other entities in-between.

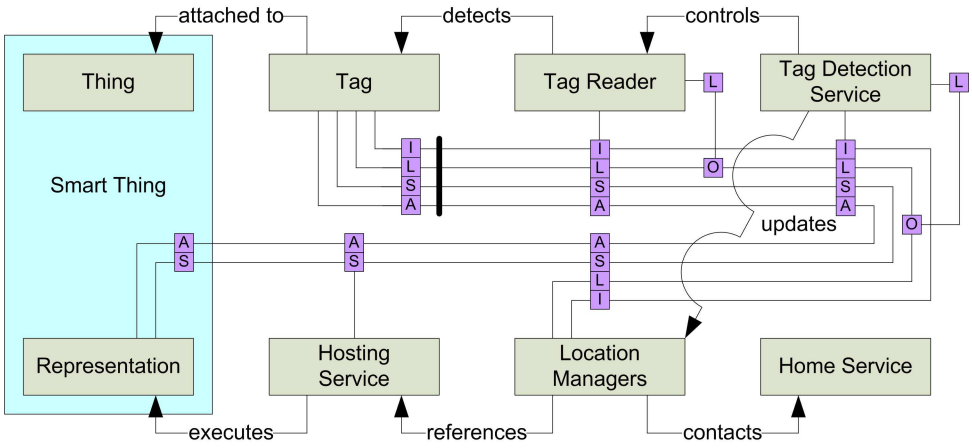

Figure 4.26: Interaction overview

4.5.2 Identification process

Every time a tagged thing comes into the coverage space of one of its technology specific tag readers, the responsible tag detection service has to ensure that it gets to know the identifier of the tagged object, which is used to identify the smart thing. This information is required by the infrastructure to manage the smart thing and also for applications, which in the majority of cases depend on the identity of a smart thing. This is done by intercepting the identification communication channel between the tag and the location manager graph by means of the tag detection service. It uses the *identifier protocol* to retrieve the identifier from the tag:

- `get identifier`
  - `request: <no parameters>`
  - `response: <identifier> or <exception>`

This information is then put back into the identification communication channel and sent to the location manager graph. A request refers to a message that is sent to the tag and a response refers to a message that is sent from the tag back to the originator of the request. If a tag directly contacts a location manager then this protocol is not used.

Concept 45 (Identification process) *If a tag comes into the coverage space of one of its tag readers, the tag detection service asks the tag detection reader for the identifier. It uses the identification communication channel between the tag and the location manager for that purpose. It uses this channel again to send the identifier to the location manager graph.*

The definition shows that reading out the identifier consists of two steps: the tag detection service asks the tag detection reader, which in turn uses a proprietary communication protocol to wirelessly contact the tag to read out the identifier and to hand it over to the tag detection service. We do not define in this scheme who plays the active role although we require that the tag detection service asks the tag detection reader. On the one hand, an active tag could trigger the tag detection reader, which in turn triggers the tag detection service, and on the other hand, a tag detection service periodically

polls a tag detection reader, which in turn periodically scans its coverage space for new tags. These two examples or combinations of both are covered by this process. Another example is an ID module that consists of a WLAN module and a GPS module and the WLAN module is only used as access to the Internet. In such a case, the ID module has to contact the location manager graph directly with the GPS data. This also shows that a communication channel is a purely logical concept.

4.5.3 Localization process

It is up to the tag detection service to find out which of the four localization types exist. By definition, at least the localization module of the tag detection service can be used, since the coverage space of a tag detection system has to represent a symbolic location, and the tag detection service has to determine the location of a tagged thing with at least one localization method. It is up to a *localization policy* for the tag detection service to define which localization types and how many should be used, since it is a trade-off between accuracy, time and power usage. If a tag detection service needs to contact the tag to obtain its location data, it uses the *location protocol* via the localization communication channel:

- `get module names`
  - `request: <no parameters>`
  - `response: list of <module name> or <exception>`
- `get location value`
  - `request: <module name>`
  - `response: <location value>, <timestamp> or <exception>`

This simple protocol defines one statement that returns a list of the module names and another statement that returns the current location value for a certain localization module. A localization module is mounted on the tag, whereby only one location module of one localization technology is allowed. The result is sent to the location manager graph, which uses this information to update its records, among other things.

4.5.4 Update of the location managers

We still need to explain how exactly the registration at the location manager graph works. The whole process can be roughly divided into three phases:

1. Retrieval of old and new nodes of the graph
2. Migration or instantiation of a representation
3. Update of old and new nodes of the graph

In the first phase, we have to determine all the relevant nodes of the graph that are currently involved in managing the representation and that are designated to manage a smart thing, since all of these nodes have to be contacted and updated. These nodes

comprise the current hosting service, the designated hosting service, the current hub, the designated hub, the current managing bases and the designated managing bases.

A tag or a tag detection service contacts the location manager graph with three data: the identification data, the localization data and an event type, which either can be an *entry event* when a tag first comes into a location, an *exit event* when a tag leaves a location, an *update event* when the position within the location changes, or a *detect event* when the location changes without explicitly using entry and exit events. First, we take a look at an entry event and assume that an exit event for the former location has already occurred so that the representation is in the grace state or has already been unloaded. The processing of such an entry event can be roughly divided into three phases:

Phase 1 Independently which node in the location manager graph is contacted, the request will be forwarded by means of the localization data to all bases that contain the specified location and none of their children contain the location, so that all "smallest" bases are known. A designated hosting service can be found in the set of all designated bases' default hosting services. The designated hub can also be found in the set of all designated bases' default hubs. Next, the corresponding nodes of the current managing graph needs to be determined. First, it will be checked whether the representation is already locally managed by asking all designated bases for the state of the representation. Every base has to store whether itself or one of its children manages a representation, and if none of them manages a representation a base can ask its default parent node. If the representation is managed locally then all managing bases can be retrieved, otherwise, the home service will be contacted to ask for the current hub, which in turn will be contacted to find all managing bases in its location domain. In both cases, all managing bases are known and the hosting service as well as the managing hub can be determined.

Phase 2 In the second phase, the new hosting service has to be determined in order to decide on a migration, an instantiation or a remaining of the representation. If the current hosting service is element of the designated hosting services then the hosting service remains. If no current hosting service exists, then the code and the state has to be downloaded from the home service. If the current hosting service is not part of the designated hosting services a new hosting service has to be selected and the representation needs to be migrated from the current hosting service to the selected. The hosting service with the smallest load factor will be selected – the load factor is the quotient of the currently managed representations and the maximum number of manageable representations at a certain hosting service. Normally, a hosting service should be referenced by bases of the same location domain, if not, every hosting service has a default hub, so that the new hub can also be determined.

Phase 3 In the third phase, the nodes will be updated: the old hub has to be notified that it is no longer managing the representation, which also forwards this message to all of its children that have managed the representation. At the new location domain, every managing base and all bases in the known state update their records and forward the message to all parent nodes until the message reaches the hub. If the representation has been newly instantiated, the procedure for the old location domain must be disregarded, since in that case, no old location domain exists. However, in both cases, the home service needs to be updated with the new hub. If the hub remains the same, only a local

migration has to take place, so that the home service does not need to be contacted and only the local branch needs to be updated. After the update, every managing base stores a reference to the new hosting service and stores the localization data it is responsible for, and starting from the hub, all paths to all managing bases will be marked with a reference to the child nodes that contain one of the managing nodes.

Update event In the case of an update event, only the localization data at the corresponding managing bases will be updated.

Exit event In the case of an exit event, the hosting service will be notified so that it will put the representation into the grace state. If within the grace period, a migration request occurs, the code and the state data can be transmitted, or if a remain request arrives, the representation will be put back into the manage state. If the grace period elapses, the representation will be unloaded and the state data is written back to the home service; in addition, the current hub will be unset and the state of the representation at the location domain's bases will be set to unknown.

Detect event A detect event is similar to the combination of an entry and exit event; the old location will be handled as if an exit event had occurred and the new location will be handled as if an entry event had occurred. In this case and all other cases above, the implementation is expected to merge all the steps for performance reasons, e.g. a home service can be asked for the old hub and updated with the new hub at the same time.

4.5.5 Sensor and actuator communication

Besides the actual management of the location, both channels between the tag and its representation also have to be established to allow a representation to retrieve sensor values and to control the actuators. As with the identification and localization channel that is managed by the entities between the tag and the location manager graph, the channel between the tag and the representation is also managed by these entities and the hosting service. The location manager that has been contacted first by the tag detection system stores the necessary handles for the adjective to identify this communication channel to enable a communication channel from the representation to the tag and vice versa. For these two channels the *sensor protocol* and the *actuator protocol* are used:

- direct access
  - `get module names`
    - `request: <no parameters>`
    - `response: list of <module name> or <exception>`
  - `get sensor value`
    - `request: <module name>`
    - `response: <sensor value>, <timestamp> or <exception>`
- subscription
  - `set subscription`

    * request: <module name>, <subscription policy>
    * response: <ack> or <exception>
    * callback: <module name>, list of <sensor value>, list of <timestamp>
  - get subscription
    * request: <module name>
    * response: <subscription policy> or <exception>

- offline callback
  - set offline callback
    * request: <module name>, <offline callback policy>
    * response: <ack> or <exception>
    * callback: <module name>, list of <sensor value>, list of <timestamp>
  - get offline callback
    * request: <module name>
    * response: <offline callback policy> or <exception>

The first group provides two methods to access the sensor modules directly: the first one is used to determine the module names of each sensor module that is integrated into the tag, and the second one returns the current sensor value for a certain sensor module. Often, the sensor data is needed at a certain frequency, so that the sensor protocol provides two methods to set and get a subscription policy that includes rules such as the frequency or aggregation of data. A callback is used to notify the representation about new sensor data. Since we cannot assume that the tag is online all the time, we introduce two statements that allow for recording sensor data when the smart thing is offline. When a smart thing is online, it can set and get an offline callback policy that specifies the rules for a sensor module to record the sensor data when its tag is offline. When a senor module is online again, it uses a callback to notify the representation about the data that has been recorded while the tag was offline.

Concept 46 (Sensors) *A tag can possess sensor modules. A tag can possess at most one sensor module of a specific sensor name. The tag and the representation can communicate via the communication channel using the sensor protocol. The localization modules are a subset of all sensor modules.*

A tag can also directly access the localization modules of a tag. From a tag point of view, there is no difference between a sensor module and a localization module, so that the sensor types list also enumerates all available localization modules that are integrated into the tag. A representation can access a localization module the same way it accesses a sensor module.

Similarly to the handling of the sensors, the actuator protocol defines what kind of data can be exchanged between both entities:

- direct access

  - get module names
    - request: <no parameters>
    - response: list of <module name> or <exception>
  - get state
    - request: <module name>
    - response: <actuator state>, <timestamp> or <exception>
  - set state
    - request: <module name>, <actuator state>
    - response: <ack> or <exception>

- offline state
  - get offline state
    - request: <module name>
    - response: <actuator state>, <timestamp> or <exception>
  - set offline state
    - request: <module name>, <actuator state>
    - response: <ack> or <exception>

One of the two groups of statements refers to the direct access if the tag is online: the get module names statement returns a list of all module names of the actuator modules on the tag. The corresponding get and set statements allow for getting and setting the current state of the actuator – the simplest actuator only provides two states: on and off. Besides the online state that is used when the tag is online, we also provide an offline state that is used when the tag goes offline. This function is mainly intended to turn an actuator off, since the representation cannot turn an actuator module off, e.g. to save energy, when its tag is in the offline state. We do not define a complete list of actuator types either, for the same reasons that were given for the sensors above.

Concept 47 (Actuators) *A tag can possess actuator modules. A tag can possess at most one actuator module of a specific actuator name. The tag and the representation can communicate via the communication channel using the actuator protocol.*

In principle, we could define the interfaces of all introduced entities in the same way as was done with the four protocols for the basic abilities, but we think that the interfaces for the other entities and their interdependencies are straightforward, so that it would unnecessarily overload the model, which is not the case with the basic abilities that have been shown.

4.6 Extensions

In the previous section, we only considered the case in which a smart thing does not contain any other smart things, its tag is only recognized by one tag reader at the same time and, it possesses exactly one tag. In the following, we describe the differences to the section above when these restrictions are canceled.

4.6.1 Containedness

As described earlier, a smart thing can contain other smart things, which can be identified in two ways: first, one localized object is within the range of another localized object, and second, a smart thing has its own tag detection system that detects smart things within it.

In both cases, the containing smart thing spans its own location domain with a hub and optional bases, which have to be registered, or unregistered with the static world location manager graph every time the containing smart thing enters or leaves a location. A smart thing can do that after having received the start or stop event. The bases of the smart thing have to point to the hosting service where the smart thing itself is executed. Independently of where the containing smart thing currently resides, the location domain of the contained smart things does not need to be adapted, since the hub of the smart things remains the same. The whole situation is illustrated in Figure 4.27.

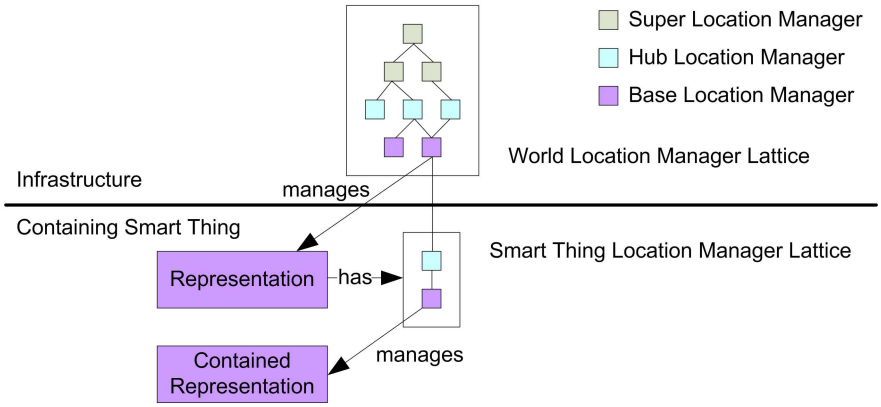

Figure 4.27: Simple containedness

In the second case, which we also refer to as implicit coupling, the smart thing possesses its own tag detection system as well as a home and at least one hosting service. The advantage is that the representation is always and implicitly executed by its own hosting service, so that the home service implicitly and statically points to its own hub. Every containing smart thing is also executed by one of its own hosting services. If another localization system than the internal tag detection systems detects a smart thing and an external hosting service could be used, then one of the internal hosting services must be used to support the mobility of the containing smart things, together with its contained smart things. The more complex scenario is shown in Figure 4.28. We differentiate two application types where implicit coupling is helpful and also possible in the near future: first, a means of transportation as smart thing, e.g. a smart truck, smart plane or smart freighter, and second, smart containers, e.g. a smart fridge or a smart tool box.

4.6.2 Simultaneous detection of the same tag

The principal problem that arises here is that we would have several communication channels to a tag although physically only one tag exists, so that the management would no longer be unambiguous. On the one hand, many tag detection system prevent such situation by using handover schemes, e.g. WLAN or GPRS, or by demanding the whole detection process, e.g. RFID where interferences of several antennae do not allow the

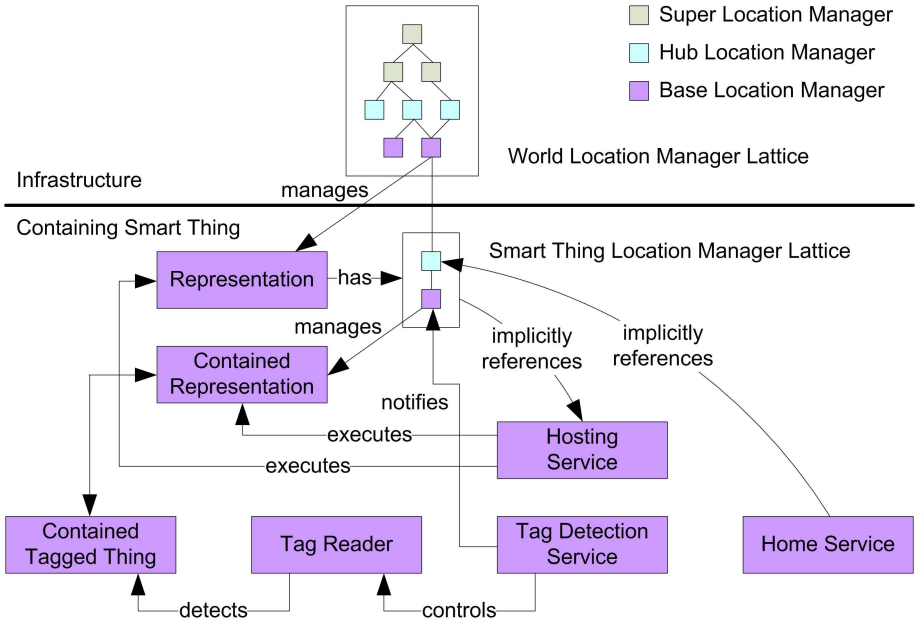

Figure 4.28: Complex containedness

RFID tag to be detected. On the other hand, some systems explicitly allow multiple detection of the same tag, e.g. Bluetooth with a Scatternet or RFID with time-multiplexing of the antennae. We assume that such a situation can only occur within the same location domain. In such a case, the bases of the second tag reader also become managing bases that point to the managing hosting service, and its localization data will be stored too. If the first tag detection reader now generates an exit event, only the old bases need to be unset and the representation can still be managed by the same host.

4.6.3 Smart things with multiple tags

A smart thing may posses several tags, e.g. a bar code and an RFID tag. In such a case, both tags can be detected simultaneously, which can be handled in the same way as described above with the simultaneous detection of the same tag, except that a separate communication between the representation and the tag needs to be opened. If only one tag of a smart thing now leaves the range of its tag reader, only that communication channel needs to be closed, so that the other is still open.

Since all of these three extensions to the simple procedure are complementary, the extensions can be combined, e.g. a smart thing with two tags is contained by another smart thing whose tag is detected by tag readers.

4.7 Application logic

The last aspect we have to consider is the connection of the application logic with our infrastructure. First, the application logic can be put in the representation itself, which is able to communicate with other representations, or separate applications register themselves with the location managers, so that applications can communicate with representations and vice versa. Both, representations and applications can ask the location

manager for other representations or applications, which enables them to communicate among themselves. Third party applications can also be used by connecting them via an application that acts as a proxy at the location manager. Chapter 6 shows how an application logic is distributed over the example application that has been mentioned in Chapter 2 with several smart things.

4.8 Lifecycle

In general, we distinguish between the lifecycle of a thing and its representation. Roughly, a thing's lifecycle consists of *not defined*, *active*, and *inactive* where both transitions are trigged by the end of its production and the final disposal of a product. The corresponding states of a representation are: *not defined*, *online*, *offline* and *inactive*. The states online and offline refer to the fact that the representation is not always hosted by a hosting service. Since a representation is bound to an existing thing, the online and offline states of a representation can only occur if the corresponding thing is in the active state. The online and offline states change if a tag enters or leaves the coverage space of a tag detection system, and the other state transitions are triggered by the activation and deactivation of the tag. The whole state machine for the thing and the representation is shown in Figure 4.29. Although a representation is not inevitably bound to the thing and many pathological situations may occur – for example, the thing is inactive and the representation is online, we do not support these situations, so that they are not defined in the lifecycle model.

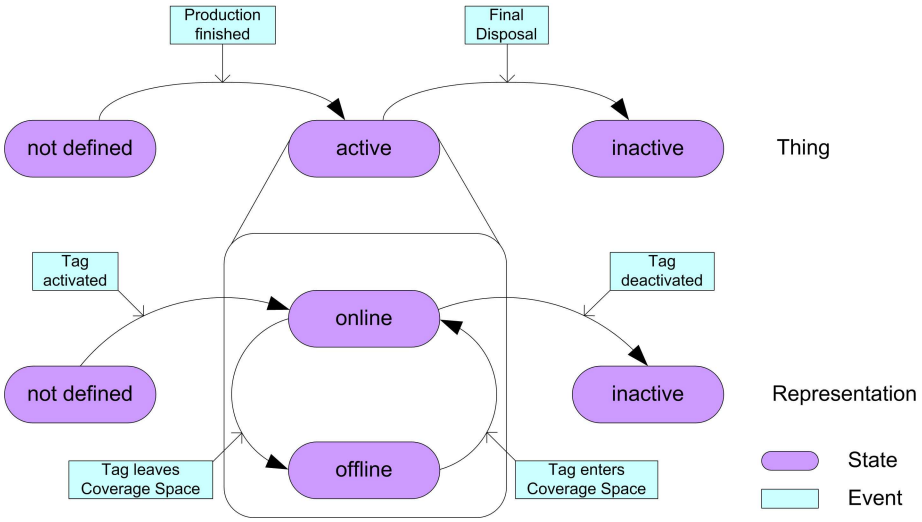

Figure 4.29: State machines of a smart thing for its lifecycle

4.9 Summary

We defined a smart thing as a combination of a thing and a representation. Both entities need to be coupled if a representation wants to provide the additional functionality for its thing. A tag detection system is used for this purpose: tags are attached to things and detected by tag readers which are controlled by tag detection services. These services read out an identifier from the tag and have to make sure that the tag is localized by

at least one of four different methods. Both pieces of information, i.e. identifier and location, are sent to a default location manager of the location manager graph. The contacted location manager has to check whether the representation is already managed in the local location domain. If not, it contacts the home service using the home address part of the identifier, which returns the hub location manager of the location domain where the smart thing is currently managed. A location domain starting from the hub references the path to the managing base location managers. The managing base location managers reference the current hosting service where the smart thing is executed. If the smart thing is either locally or remotely managed, then the smart thing will be migrated from the former hosting service to the new hosting service. The location manager graph has to be correspondingly updated, so that the old location domain no longer references the smart thing and the new location domain correctly references the smart thing. The home service also needs to be updated with the new hub location manager. If the smart thing was not managed before at any location domain, the static code and the dynamic state data need to be downloaded from the home service. A smart thing can use local data storage to keep its data private at this site or it can use the data storage at the home service, so that the data is accessible from any hosting service. After a smart thing has been migrated or newly instantiated, it opens two communication channels to its sensors and actuators to retrieve sensor values or to control its actuators. Smart things as well as applications are registered at location managers so that they can find each other and are able to communicate among themselves. They also use the location managers to be informed about the neighborhood relation. The containedness and composition relation, on the other hand, is managed by the smart thing itself.

Chapter 5

Architecture of the Smart Thing Systems

In this chapter, we show how the concepts of the previous chapter can be implemented in real systems. The presentation of these smart thing systems mainly serves two purposes: first, it can be regarded as proof-of-concept, since our model only makes sense if it can be easily and efficiently implemented, and second, it analyzes different implementation strategies, so that their advantages and disadvantages can be stated. Overall, we present three systems that have been developed iteratively in order to refine the model and to explore new implementation strategies. The model that has been introduced in the last chapter comprises the experiences we have gained with these three systems and represents the ultimate refinement of these experiences. Initially, we briefly present the work of our research group that has influenced the first smart thing system called Voxi. From this system that has been developed bottom-up using Jini as the underlying middleware platform, we extracted some general concepts for our model. With these first concepts, we started to complete the model of collaborating everyday items. Next, we present two systems that implement our model. The main differences between the two implementations are the usage of different middleware platforms, the design decision on whether to support migration of representations, and the focus on different parts of the model. First, we present the Wsst system that builds on Web Services and that does not support migration. Second, the Iceo project is presented, which again builds on Jini due to performance reasons and that supports migration. Besides the actual presentations of the systems, we also look at how each system implements the concepts and we explain the design decisions.

5.1 Previous work

The actual roots of smart things in general have already been discussed in Chapter 1, but first ideas about systems that support smart things were first mentioned by our research group in [70], and the usage of RFID tags as a bridge were first discussed in [118]. The former demand a virtual counterpart that we call a representation for everyday items that spans a fifth dimension besides the traditional four dimensions of our reality that consists of three-dimensional space and one-dimensional time. This fifth dimension allows for additional functionality for the everyday items which need to be offered by an infrastructure that has to enable code mobility. In contrast to it, we see

migration of representation only as optional, since migration of representations makes the systems more complex and does not guarantee the availability of a representation all the time. One focus is on an event-driven communication platform that is designed to enable sensors and actuators to communicate with their virtual counterparts in the fifth dimension. The event-communication can be interpreted as part of our concept of communication channels. This position statement was followed by [73], which describes a first application that builds on a simple event-based infrastructure. Depending on the ingredients that are placed on a kitchen counter, the smart chef application suggests dishes that can be prepared with these ingredients on a nearby display. An RFID antenna is mounted beneath the kitchen counter and all items are equipped with RFID tags. Depending on an RFID tag entering or leaving the range of the antenna, the lowest layer of their infrastructure generates an entry or an exit event. To avoid flickering concerning spurious and fast generated exit and entry events, an event filter at one level above uses a threshold value to filter out such spurious event pairs. Finally, the events are delivered to the application, which looks up what dishes can be prepared with the current ingredients. In this first trial of an event-based infrastructure there is no real borderline between the application and the virtual counterparts. In our model the clear separation between representations and location-dependent services is one of our key assumptions, but it is up to the application developer how to distribute the actual application logic between the representation and location-dependent services. Besides the application and the event infrastructure, they see that location information, the physical proximity of smart things, and the need for a virtual counterpart to communicate with other virtual counterparts or services are general challenges in research into smart thing infrastructures.

5.2 Voxi

Voxi, short hand for *v*irtual *o*bject e*x*tensible *i*nfrastructure, is the first system that has been developed by our research group without the collaboration of this dissertation's author to facilitate the development of applications that make use of smart things. It provides a software framework (FW) consisting of Java classes to facilitate the actual development of applications and representations as well as middleware services (MW) that can be regarded as the infrastructure that is needed by representations and applications. In contrast to the event-based infrastructure the smart chef application uses, the borderline between a virtual object, i.e. a representation, and an application is clearly defined. This system partly uses the approach of our model to distribute the application logic between representations and location-dependent services, but the location-dependent services are rather like the representations, so that they do not use the full potential of a clear separation as we will show later. [27] provides a more detailed description of the system.

5.2.1 Overview

As Figure 5.1 shows, Voxi works independently from the actual tag detection systems through the usage of an event source that transparently detects objects in the real world and transmits their objectIDs as well as the locationID of the event source together with a timestamp to a virtual object manager, which is the central element of the system. The virtual object manager checks at the lookup service whether the corresponding virtual

object and the corresponding virtual location have already been instantiated. If not, it contacts the virtual object repository to download the code in order to instantiate them and to register them at the lookup service. Finally, the virtual object manager forwards the entry event to the virtual location, which in turn forwards the event to the virtual object. If a virtual object repository receives an exit event with an objectID and locationID, it tries to unload the corresponding virtual object as well as to unregister it at the lookup service – a virtual location cannot be unloaded. The artefact memory can be accessed by virtual objects to persistently store its state. Since this system is built to be deployed locally and not worldwide, we had to extend the entities that Voxi proposes to more general concepts, e.g. Voxi only uses non-hierarchic locationIDs to manage locations, whereas STILM uses a hierarchic approach to support mobile smart things and static locations as well as symbolic location names and physical positions.

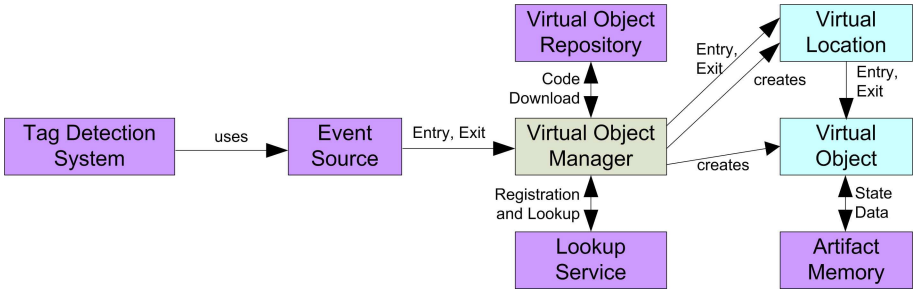

Figure 5.1: Overview of the Voxi system

5.2.2 Components

In the following, we describe every entity that has been mentioned in the overview in more detail.

Event source The event source can be roughly regarded as the tag detection service that builds the connection between the real and the virtual world, so that the deployed tag detection system needs to be coupled with the event source that delivers the generated events from the tag detection system to the virtual object manager. The interface of an event source as a Java object is quite simple: first, the event type has to be specified, i.e. a tag detection system must specify an entry event if a tagged thing comes into the range of a tag reader and an exit event if it leaves the range. Second, the objectID must be stated, the intention of an objectID is the same as with the identifier except that an objectID provides no internal structure, so it can be any Java string. Third, the locationID must be stated. In this case, location only refers to the symbolic location of a tag reader. There is no location hierarchy and there is no differentiation between a tag reader in the static world or within a mobile smart thing as proposed by STILM. A truck or a warehouse are modeled in Voxi as simple locations. Fourth, a timestamp when the event occurred is necessary to enable a virtual object to record its history. Finally, the virtual object manager that is supposed to receive this event needs to be specified. Since the whole system relies on Jini, the event source asks the lookup service for a proxy of the given virtual object manager, which then is used to notify the virtual object manager about the event. The structure of an event source is shown in Figure 5.2. The main differences to our more general concept of a tag detection service are that the event

source only supports symbolic location names, does not allow a worldwide distribution and has a strong focus on RFID as the underlying identification technology.

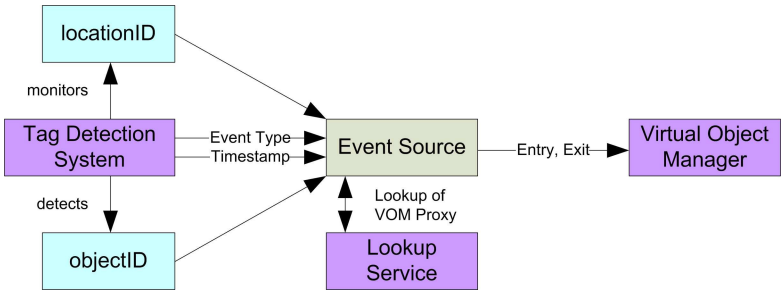

Figure 5.2: Structure of an event source

Virtual object manager The virtual object manager can be compared with the hosting service in the model, since the main task of the virtual object manager is to provide an execution environment for the virtual objects and virtual locations. A virtual object manager is realized as a Jini service that has to register itself at the lookup service, so that other entities, such as the event source, can contact it. For every entry or exit event that a virtual object manager receives from the event source, it creates a thread to process the event. In the model, the processing of these events is handled by the contacted location manager, but Voxi has no location hierarchy so that the processing can be directly carried out by the virtual object manager. In the case of an entry event, the entry thread first checks at the lookup service whether the virtual location and object of the corresponding reported locationID and objectID have already been instantiated. On the one hand, a virtual location might already been instantiated because another virtual object has already been there, on the other hand, a virtual object might already been instantiated since it might have been detected at another location before. If one of them needs to be instantiated, the virtual object repository will be contacted first to download the corresponding Java class file that is found by simply appending .class to the objectID, or locationID. After the instantiation of the virtual location or the virtual object, they can be registered as a Jini service at the lookup service. Finally, the entry thread forwards the entry event to the virtual location, so that the latter can update its records. If a virtual object manager receives an exit event, it starts a new exit thread that processes this event: first, it gets a proxy of the corresponding virtual location from the lookup service to forward the exit event to the virtual location. After the corresponding virtual object has been stopped, the virtual object manager will be notified and unregisters the virtual object at the lookup service. The main difference to our hosting service is the missing option to distribute the whole system. In our model, the location managers are responsible for finding an appropriate hosting service. In this case, the hosting service can be contacted directly by the tag detection system.

Virtual object repository The virtual object repository is similar to the code service in the model. In the Voxi implementation, the virtual object repository is just a regular HTTP server that stores the Java class files. The name resolution is implicitly done by the entry thread of the virtual object manager, since it appends the .class to the objectID, or locationID.

Lookup service Since Voxi relies on Java, RMI and Jini, the lookup service is taken from Sun's Jini implementation. There is no direct match with a concept from our model, since RMI and Jini are subsumed as underlying network technology and therefore only an implementation detail. Every virtual object manager, virtual location and virtual object is registered at one central lookup service. Due to this one central lookup service, a system can only be deployed locally which does not fulfil our requirements.

Virtual location A virtual location in Voxi represents the coverage space of a tag reader. Every location has an identifier: the locationID, which is used by entry and exit events to state the virtual location. Dependent on the tag reader, a virtual location can either be static, e.g. the tag reader is mounted in a room, or it can be mobile, e.g. the tag reader is installed in a truck. In our model, only a truck as a smart thing can have its own application logic, but in Voxi all virtual locations, including the static locations, can have their own application logic. A virtual location that has its own execution thread is instantiated by an entry thread the first time the virtual location is referenced. Two main tasks have to be fulfilled by a virtual location: first, it stores the references of all smart things that are currently within its corresponding coverage area. With these references, a virtual location is able to provide the absolute neighborhood concept, so that smart things can communicate with their neighbors at this location. The second main task is the provision of an event bus for the smart things that are managed by this location. A virtual location receives entry and exit events from the corresponding threads of the virtual object manager to update its own records and to notify the virtual objects about their own events. Besides forwarding the entry and exit events, virtual objects can publish and subscribe for arbitrary events at a virtual location. A developer is free to extend the standard virtual location class to implement a location-specific behavior. In Voxi, a virtual location comes closer to our representation than it does to an actual location in our model. In our model, where we can have several more complex and powerful services that can register themselves at a location manager for a static location, Voxi allows only for simple virtual location implementations that we only use for representations.

Virtual object Virtual objects, in our model referred to as representations, encapsulate the additional behavior of a smart thing. Every virtual object has its own execution thread and is registered as a Jini service at the lookup service. It is instantiated by the entry thread of a virtual object manager and it gets its entry and exit events through the virtual locations, which it can also use to publish or to subscribe for application-specific events. The Java interface of a virtual object is lean: it provides three methods to start and to stop the virtual object as well as to get a timestamp when the virtual object was used recently. The stop method is only a signal for a virtual object to shut down as soon as possible, but there are no real restrictions as to whether or when a virtual object really shuts down. After receiving the entry and the exit event, a virtual object can use the artefact memory to retrieve or to store its state data. To learn about its environment, a virtual object has a list of references to all virtual locations where it currently resides. The references to the virtual locations can be used for event communication or to get to know the neighbors at these locations. Besides its objectID, a virtual object has a description that can be used to encode additional information about the type and the instance of the virtual object. A developer has to extend the basic virtual object class to implement a smart thing's specific behavior. Figure 5.3 shows the relation of a virtual location and a virtual object. From all entities in the Voxi system, the virtual object

comes closest to one of our concepts: the representation. A representation has an identifier, is able to communicate with other representations and locations, i.e. with registered services. It gets a start and a stop signal from the hosting service. A smaller difference here is that we propose a grace period after the stop signal, where representations get a chance to save their state and to properly release used resources before they are definitely shut down by the system. In the Voxi system, it is up to the representations whether they shut down or not after the stop signal.

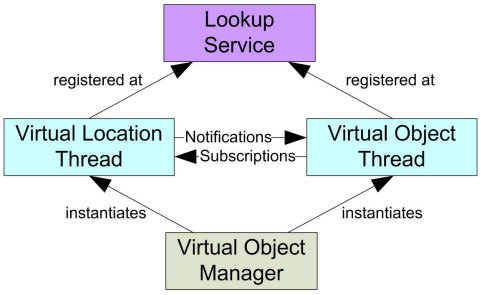

Figure 5.3: Relation of a virtual location and a virtual object

Artefact memory Although the artefact memory can be regarded as independent service, it is modeled as a virtual object that resides at the "pure" virtual location "Virtopia". Its main task is to implement the history concept, and since it is centrally available, every virtual object can use it to store received events or its state data, so that the artefact memory can also be regarded as data storage. Due to its central availability, applications can query the artefact memory for past information, such as which smart thing was at a certain location together with another smart thing. The artefact memory uses a standard database management system that is contacted via JDBC, so that SQL queries can be easily posted to the artefact memory. Another important difference to our model is that we have two kinds of data storages: one at the home service, where a representation can store public data, and another at the hosting service, where a representation can store private data. Due to the central availability of the artefact memory, it can only provide public data.

Communication The basic communication paradigm in Voxi is event communication, which is supported by virtual objects, virtual locations and virtual object managers. These entities provide a lean communication interface that consists of a request and a notify method that both accept a Voxi specific event. The request method additionally returns an event as an answer. An entity that needs to communicate with another entity has two options: it can use a virtual location as event bus, it uses an event delivery manager object that hides the lookup process at the lookup service and directly delivers the event to the corresponding entity. An entity can also directly make use of RMI to call methods on other entities – prior to this, the lookup service can be used to get a reference to that entity if it is not already known. The actual event management also relies on RMI. In our model, we also propose event bus services at hosting services for local asynchronous event communication and the option to use synchronous communication between representations themselves, and between representations and location-dependent services.

Extensions Besides the actual virtual objects and locations, Voxi provides virtual meta objects and virtual meta locations. A meta object is a regular virtual object that manages several things in the real world, so that mapping between their objectIDs and the responsible virtual meta object is required. This task is fulfilled by the virtual object repository where static files with the objectID as filename denote the virtual object class that needs to be instantiated. Thus, the implementation of a virtual object that acts as a virtual meta object has to make sure that it can process several entry and exit events for every thing it is responsible for in parallel. Analogously, the same also applies for virtual locations. These meta-concepts are contradictory to our definition of a smart thing that is a one-to-one mapping between things and representations, so we explicitly do not support such concepts.

5.2.3 Comparison with the smart things model

In comparison with the smart things model, the Voxi system already implemented or partly implemented some of the concepts of the smart things model, as has been shown above, but some important aspects are missing in this first system: one of these concepts is the support of sensors and actuators that are not generically supported by Voxi, or the structural aspects of a smart thing concerning containedness and composition or of the location management in general. Migrations of virtual objects, e.g. by serializing the object state, is not supported, but virtual objects can use the artefact memory to store state data when they receive the exit event and in turn, after having received the entry event, can retrieve their state from the artefact memory. The main problem is that one Voxi system can only work as stand-alone system and is not able to work with other Voxi systems on the Internet. In fact, the virtual meta object approach conflicts with our definition of a smart thing, which states that it is a unique association of exactly one thing and exactly one representation, and also the artefact memory as a service cannot be modeled as a smart thing in our model. Table 5.1 summarizes the main tasks of the different elements of the Voxi system.

5.3 Wsst

Wsst, short hand for deployment of *w*eb *s*ervices to model *s*mart *t*hings, is the second smart thing system of our group that has been developed under the supervision of the dissertation's author. It builds on Web Services as an underlying service discovery platform, in contrast to Voxi, which builds on Jini. Besides the different service discovery platform, in Wsst we added some of the missing concepts and also analyzed different design decisions. In comparison to Voxi, Wsst has a hierarchy of location managers that allow for smart things to be tracked world-wide, a hierarchy of UDDI servers that allows users to find every representation on the Internet instead of supporting migration and finally, Wsst allows for composition of smart things so that the location information can be handed down. Similar to Voxi, Wsst is a system that consists of a software framework, in this case with C# classes, and some middleware services. A report [101] provides a more detailed description of the system.

Element	Type	Realization	Main tasks
Event source	MW	Java object	connection for tag detection systems
Virtual object manager	MW	Jini service	download code for virtual objects and locations, instantiates them, registers them
Virtual object repository	MW	Web server	provides code (Java class files)
Lookup service	MW	Jini service	manges references to virtual objects and locations and virtual object managers
Virtual location	FW	Jini service	neighborhood relation, event bus for representations
Virtual object	FW	Jini service	encapsulate additional functionality

Table 5.1: Summary of Voxi elements

5.3.1 Overview

As Figure 5.4 shows, a tag detection system is responsible for detecting the tagged things in its range. When such a tagged thing comes into the range of a tag detection system, the latter has to read out an identifier (URI) stored on the tag. Using this URI, the tag detection system contacts a hierarchy of UDDI servers to resolve this URI into the corresponding URL of a web service that acts as the representation for the smart thing. Note that in Figure 5.4 smart thing refers to its representation. After the resolving process, the tag detection system calls a method on the representation that updates the location of the tagged thing. Finally, the representation registers itself at a hierarchy of location managers, which implement the neighborhood concept. If a smart thing is part of another smart thing, then its location does not need to be set, since it inherits this information from its parent node in the composition tree. In comparison to Voxi, this procedure has no central elements, so that it can be deployed globally.

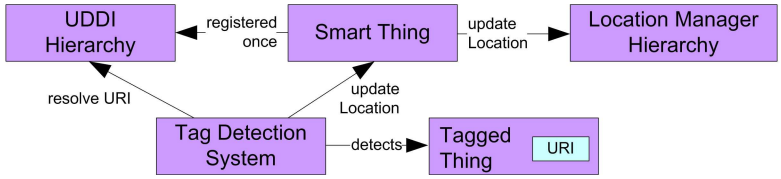

Figure 5.4: Overview of the Wsst system

5.3.2 Components

Tag detection system Similarly to Voxi, Wsst does not define the actual tag detection system either, but the interface which a tag detection system can use to register a tagged thing which has newly appeared. Unlike in our model, the tag detection system does not need to update the location manager, but the representation directly. This is possible

since migration is not supported, so that the static representation is always reachable. To update the representation about its location, the tag detection system has to get to know the identifier of the tagged thing first, in order to be able to look up the representation as well as to determine the position of the tagged thing. The identifier in Wsst is similar to the one we defined in the model. It consists of a name and an address part: the name is an unique and random UUID that has been generated by a UDDI server where the representation is registered, the address part consists of a DNS-like name that denotes the UDDI server where the representation is registered, e.g. *uri:pharma.apotis:40a96d21-ee00-0000-0080-e698e3243f5a* with *pharma.apotis* as DNS-like name and *40a96d21-ee00-0000-0080-e698e3243f5a* as UUID. The tag detection system consults a hierarchy of UDDI servers that returns the URL where the representation can be accessed. Before the tag detection system can update the representation, the location still needs to be determined. This can be done in an arbitrary way, since the location model supports the WGS 84 standard and a symbolic naming scheme similar to the one defined in the symbolic world location model. Finally, the symbolic location of the tag reader can be used to call the set-location method of the representation.

UDDI Hierarchy As explained in 3.1.2, UDDI servers normally build a service cloud. Since they are originally intended to store references to business services on the Internet, a service cloud is an appropriate means for their management. In our case, we have to handle many more entities, since every representation needs to be registered at the service cloud, the service cloud might break under the registration load, so that a structure within the UDDI servers would be helpful. For this purpose, we developed a DNS-like naming scheme and algorithm that allow UDDI servers to partition the registrations of representations. The DNS-like address consists of UDDI server names that are separated by dots, e.g. "pharma.apotis" whereby "pharma" as well as "pharama.apotis" denote a UDDI server. This approach is compatible with the identifier concept in our model, where we only state that the address part of the identifier depends on the underlying network technology. Since we use UDDI as underlying technology, we have to use an addressing scheme that is compatible with UDDI. If a representation should be looked up, the algorithm first checks whether the current UDDI server is already the right UDDI server. If not, it forwards the request to one of the UDDI servers on the path to the right one. After the right UDDI server has been found, the UUID of the name can be used to look up the URL of the representation. The UDDI hierarchy, i.e. the actual UDDI server that stores the reference of a representation, can be regarded as a simple home service, although it does not provide data and code storage. These are only necessary if migration is supported. Since the UDDI server requests delay execution, every entity can make use of a UDDI cache that only contacts the UDDI server hierarchy if the URL for the requested URI has not already been cached. Figure 5.5 shows an example UDDI hierarchy.

Web server Since Web Services normally communicate via HTTP, a web server is needed that is able to process the SOAP messages for the representations in order to invoke their methods. The web service is comparable to the hosting service, since it acts as an execution environment for representations. This also means that the representation is available although the thing might currently not be in the coverage range of any tag reader. On the one hand, since migration is not supported, the system is much simpler to realize and the representation is always available. On the other hand, the delay between

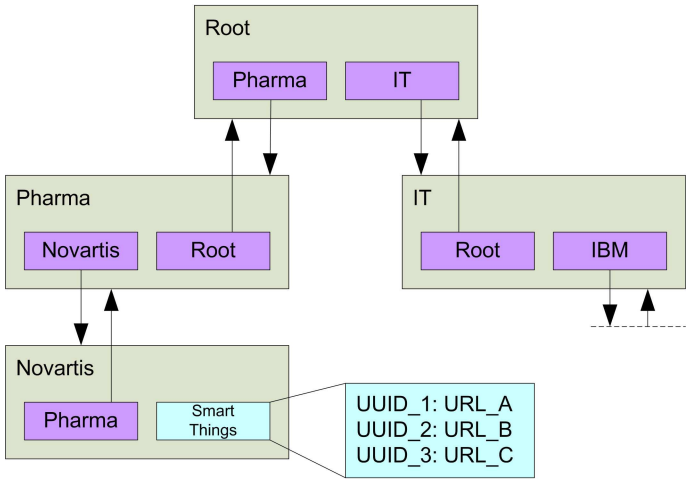

Figure 5.5: Example UDDI hierarchy

the tag and the representation is higher and temporary network failures can break the connection between tag and representation. The Iceo system that will be described at the end of this chapter supports migration. Since the support of migration is one key decision when implementing the model into a system, we are able to discuss the pros and cons.

Smart Thing In Wsst, the smart thing also stands for the actual representation of a smart thing. A smart thing is realized as a web service without a separate execution thread and even with a complicated procedure to save the current state data between two method invocations. Its URL is stored in the UDDI server hierarchy so that the web server where a smart thing is executed can be contacted by any other smart thing or application. A smart thing implements a lean C# interface that provides the basic abilities of a smart thing, which also include the possibility to retrieve its WSDL document where all the methods of a smart thing are listed. That way, a developer is able to easily extend the generic smart thing interface to include smart thing-specific behaviour, which is described in that WSDL document. Common to all smart things is that they know their identifier, an arbitrary and additional name, their current location, and the location managers where they have to register and to unregister themselves if they have been updated with a new location. Since smart things implement the composition relation, they also know their parent and child nodes in the composition tree. This relation needs to be explicitly set by external applications, for example, the new location is set externally by the tag detection system. Additionally, this provides a method to retrieve the history of a smart thing that mainly refers to the visited locations, but can also contain any past state information. There is no dedicated artefact memory in Wsst, so that every representation stores its state and its history data in a file in an XML format where it is executed. If a new location is set by an external tag detection system, a smart thing first has to check if it is the root of the composition hierarchy. If this is the case, it first unregisters itself at the old location and then registers itself at the new location at the location manager hierarchy. The registration process returns all responsible location managers where the smart thing has been registered. If a smart thing is not the root of the composition tree, it does not need to update its location, since this information is handed down from the root in the composition tree. Normally, smart things are rather passive in contrast to

the representations in Voxi which have their own execution thread, but a smart thing in Wsst is able to create its own execution thread so that more active behavior is possible. Figure 5.6 shows the structure of a smart thing. As these explanations have shown, a smart thing in Wsst is a full implementation of the representation concept of our model.

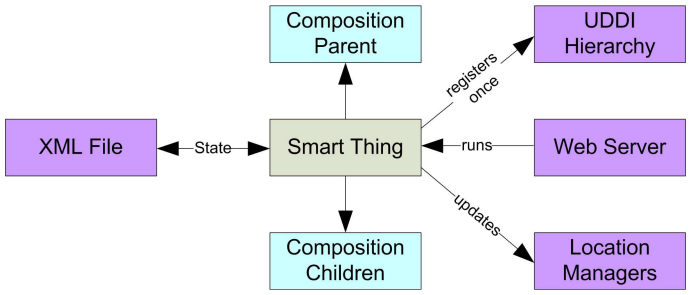

Figure 5.6: Structure of a smart thing

Location managers The location managers, which are the last piece in the Wsst system, come close to the definition of the location manager graph in the model. A location manager that is implemented as a web service, such as the representations, is responsible for a certain location. A location always has one symbolic name and contains one contiguous space of WGS 84 positions. Other location managers can be its children if their locations are real subsets of its location. Two restrictions on the location manager graph are that exactly one root node with the symbolic name world exists that comprises all WGS 84 positions and the second restriction is that the graph can only be a tree. A smart thing needs to be registered at this world node that hands down the registration request to all location managers that contain the new position of the smart thing but none of its children. These location managers can be asked for the absolute neighbors of a smart thing. One main difference to STILM is that the Wsst location model does not support a dedicated localization within smart things, but allows for smart things that contain other smart things to register themselves in the mobile branch of the location tree. Since locations have no behavior, locations can have a reference to a host service that is a regular smart thing, e.g. the location truck has a reference to the host service truck. If the location and the smart thing interface is implemented by the same class, the host service reference points to itself. Since all these functions are listed in a lean C# interface that must be implemented by every location manager, it can be contacted by any smart thing or application. Figure 5.7 shows an example location manager hierarchy. Since migration is not supported, the concepts of super, hub and base location managers are not as important as they are with systems that support migration. Therefore, these concepts are implemented in the Iceo system, which explicitly supports migration.

Communication In Wsst, we have no dedicated event communication, since the communication is handled by explicit method invocations. Applications and smart things can access the location manager tree to get to know the identifiers of the smart things at a certain location, which need to be resolved first by the UDDI hierarchy before other smart things can be contacted. All communication issues are transparently handled by SOAP's method invocations. Since the focus on supporting sensors, actuators and different localization technologies is put in the Iceo system instead of the Wsst system, we have no real counterpart of communication channels in this system.

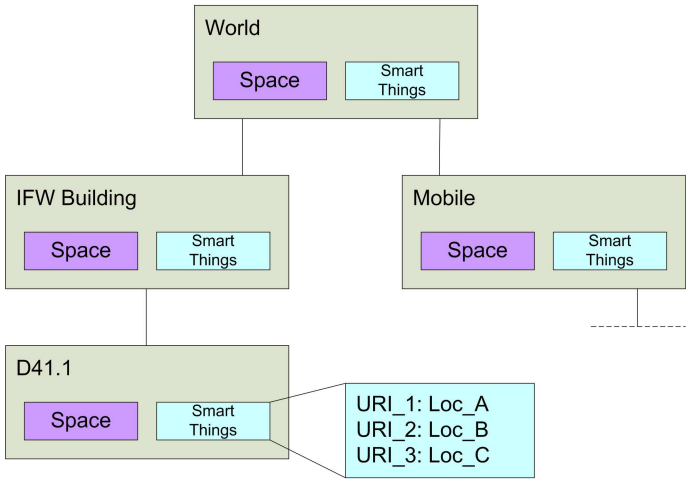

Figure 5.7: Example location manager hierarchy

5.3.3 Comparison with the smart things model

In comparison with the Voxi system, Wsst comes much closer to our smart things model with the restriction that migration is not supported: a UDDI server acts as a simple home service, a web server as a hosting service, a web service as the representation and a restricted location manager hierarchy is also available. The main restriction of the location manager hierarchy is that it only allows a tree instead of a more general graph structure. Location domains and the differentiation between hub, base and super location managers are not necessary since they are only needed to support migration. The option to use a smart thing as a host service at a location manager allows for a restricted containedness relation, since smart things do not have their own location model, but the Wsst location model is a hybrid location model that uses the WGS 84 standard as well as a symbolic naming scheme. With the combination of a UUID and the DNS-like address of a UDDI server, the structure of an identifier corresponds to the identifier introduced in the model. Besides the implementation of the neighborhood and the history concept, Wsst also implements the containedness relation that allows for handing down location information to make the system more efficient. Although Wsst implements more of the concepts of the model, there are still some important aspects such as the support of sensors and actuators or the generic support of tag detection systems missing. Due to the fact that migration is not supported, the implementation of some concepts becomes much simpler, e.g. a home service does not need to provide the code and the current state of an object. A better support of the location graph and a better support of the containedness relation is also missing in Wsst. In comparison with the Iceo system, Wsst supports no migration. Wsst has its focus on smart thing relations like composition and Iceo has its focus on the abstraction of different tag detection hardware which includes sensors and actuators. Table 5.2 summarizes the main tasks of the different elements of the Wsst system.

5.4 Iceo

Iceo, short for *i*ntelligent *c*ommunicating *e*veryday *o*bjects, is the last smart thing system that has been developed under the supervision of the dissertation's author that, on

Element	Type	Realization	Main tasks
UDDI hierarchy	MW	UDDI server	maps URI to URL of smart thing web service
Web server	MW	Web server	execution environment for smart things
Smart thing	FW	C# web service	encapsulates additional functionality
Location managers	MW	C# Web Services	supports composition neighborhood concept supports mobile locations

Table 5.2: Summary of Wsst elements

the one hand, implements the missing concepts and, on the other hand, considers performance aspects. In comparison to the previous systems, it has a dedicated support for tag detection systems, sensors and actuators, and it allows for efficient migration of representations. Due to the poor performance of the web service approach, Iceo again builds on Jini as the underlying service discovery platform. The implementation is much more complex since it relies on the work of three master theses in contrast to Wsst and Voxi, each of which only builds on the work of a single master thesis. Thus, Iceo can go deeper into the analyzed aspects. The reports [30, 107, 124] provide a more detailed description of the system.

5.4.1 Overview

In this overview, we assume that a smart thing is detected the first time. A hardware specific scanner detects the tagged things within its coverage space and informs its base about this fact with the identifier that has been read out and the symbolic location of the scanner system. The identifier consists of a name and a home address as described in the model – the location can either be a symbolic location or a physical position of the detected tagged thing. Bases are comparable with base location managers in the model that are arranged in a tree structure with a hub as the root of this tree. Since the base as well as its child nodes process this identifier the first time, they forward the execution request toward the hub. In our case, the hub has not processed the identifier before, either, so that it uses the home address part of an identifier to contact the home service, in this case called producer. The producer knows the hub where the representation is currently executed as well as the static Java code. Since the representation is currently not managed anywhere, the producer sets the requesting hub as managing hub and returns the static Java code to the hub. The hub and the other bases on the path to the base with the scanner that detected the smart thing, hand down the code and mark the way to the managing base with the identifier. The managing base has its own object manager as a hosting service that instantiates the representation. The scanner module creates two communication channels to the sensors and to the actuators that are given to the representation. After the instantiation, a representation can ask the base for neighbors and can use both communication channels to control the sensors and the actuators. If the smart thing leaves the range of scanner A.1 and enters the range of scanner B.1, the representation will be migrated from base A to base B and the paths

from the hub to the old and the new managing base will be updated. Figure 5.8 shows the important components of Iceo. As these explanations show, the main concepts of the model, especially home service, hosting service, location managers, communication channels and the tag detection system including sensors and actuators are implemented by Iceo. Relations of smart things such as containedness or composition are not part of Iceo, since these concepts were already focused on in Wsst. Since the implementations of these concepts are not interdependent, one implementation is sufficient.

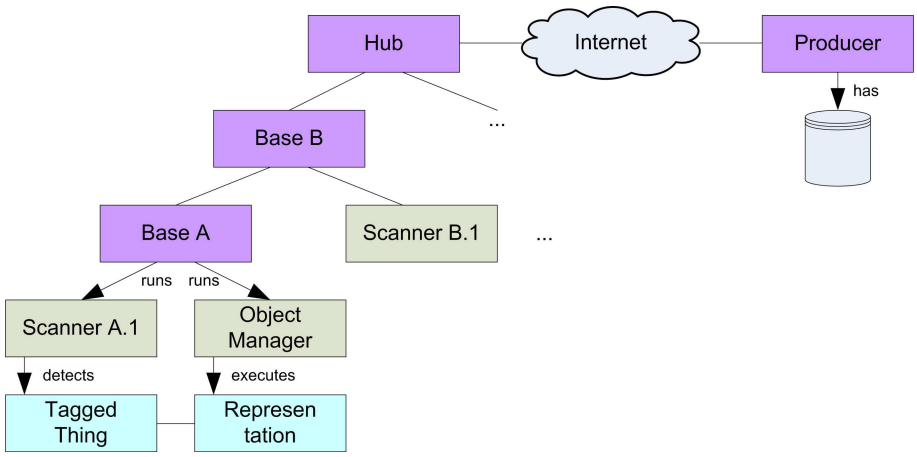

Figure 5.8: Overview of the Iceo system

5.4.2 Components

Scanner In contrast to the previous two smart thing systems, Iceo's framework part provides the basic classes to model tag detection systems. In Iceo, scanner stands for the tag reader that scans for tags in its coverage space. Although the control of the tag detection hardware is hardware specific, some common properties of different tag detection hardware can be extracted: first, a tagged thing is modeled as a real object that possesses an identifier and a hardware-specific address that is used by the tag reader. Then, these real objects are managed in a real object pool that states whether a real object is detected or not. The scanning interface represents the actual tag detection service that detects these real objects. The four localization combinations, i.e. the symbolic location or the physical position of a tag or its reader are modeled and integrated into the tag detection process. A hardware specific solution, e.g. a bar code solution, only needs to extend these basic classes with the hardware specific behavior. Currently, Iceo has implementations for bar codes, BTNodes and RFID i-Code tags, which provide their functionality as Jini services. Besides the identification and localization issues that are handled by the scanning component, the communication with sensor, actuator and localization modules is also supported. For every component that is supported by a specific tag detection system, a correspondent Jini service will be instantiated that handles the communication with the modules on the tag, e.g. if a tag possesses several sensors and actuators, a sensor and an actuator Jini service will be created that handles the communication with all tags concerning the component's sensors, or actuators. Thus, Iceo allows for communication channels, as mentioned in the model, between a tag and its representation. Figure 5.9 shows the four module groups of a tag that provide their functionality as Jini services.

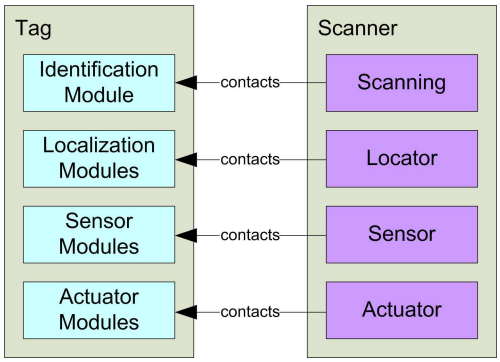

Figure 5.9: Tag modules provided as Jini services

Base A base in Iceo, in fact, is not the same as a base location manager in the model, but comes close to it. One base can control several scanners, and it provides the execution environment for the representations, for all of its scanners. Bases that are higher in the tree hierarchy should be responsible for scanners that cover a larger area, which in turn contains the areas of its child bases' scanners, e.g. a WLAN tag detection system that covers a floor can have several RFID tag detection systems as child nodes that cover only single rooms. From the hub to the hosting base, every base stores information on whether itself or one of its children is hosting a representation as mentioned in the model. The setup of the base tree is not predetermined, but rather takes into account which and how many scanners are installed and how many representations need to be hosted. As rule of thumb, it can be stated: the more scanners or representations that have to be managed, the larger the base tree. Every base which is a Jini service, has its own lookup service, where all services belonging to this base are registered, i.e. the hardware specific scanners as well as the optional sensor, actuator and localization services are also registered at the lookup service of the base where the scanner is managed. Every base has its own base location service, another Jini service, which manages the locations of the identified smart things of the managed scanners, so that queries about smart things' neighbors can be posted to that service. A base has exactly one parent and several child nodes. These nodes do not directly reference the corresponding node, but its lookup service. This can be advantageous if one of the nodes crashes, since the lookup service manages such situations. The actual execution of a representation is handled by an object manager that is described below. Figure 5.10 shows the lookup services of a base, hub and a producer and the minimum set of Jini services that are registered there.

Hub As the root of the base tree, a hub builds the connection to the producer (home service). Like a base, it knows which of its child nodes is hosting a certain representation. If a hub is asked for a representation that is not managed by its tree, it contacts the producer service which returns the hosting hub or if no hosting hub exists, a producer service can provide an initial version of a representation. Unlike the model, a hub is not a special base, but a separate entity that cannot control scanners or manage representations. Its main task is just to build the bridge to the producer. Like a base, a hub is also registered at its own lookup service, where its own hub location manager is registered, too. Since a hub cannot host a representation, the hub location service contacts the corresponding base location services, e.g. a hub location service can be requested to return references to all smart things of its base tree.

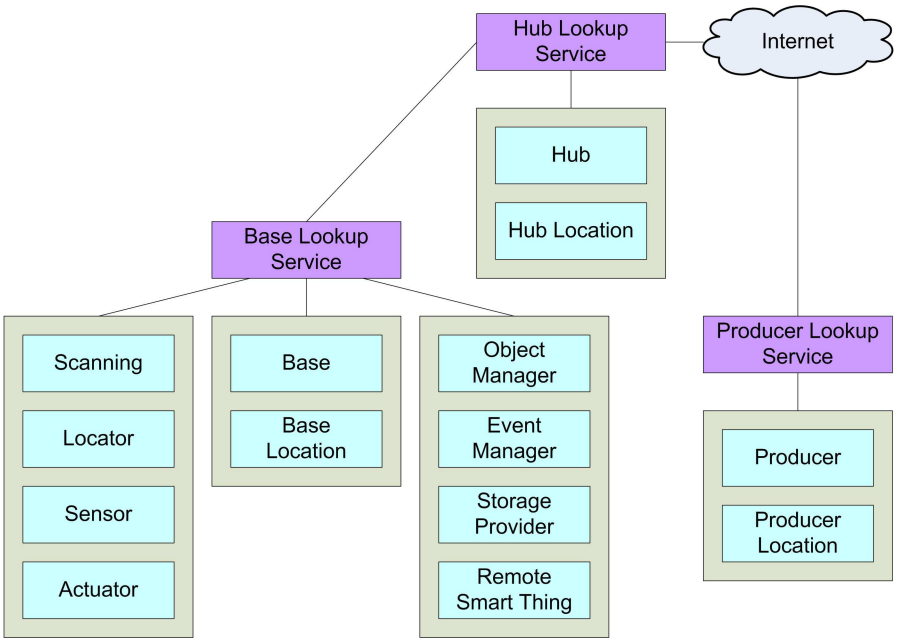

Figure 5.10: Lookup services with registered Jini services

Producer A producer can be compared with the home service in our model, since it is referenced by an identifier and it knows at which base in the tree its representations are currently managed. Analogous to the base and the hub, a producer is a Jini service that is registered at its own lookup service, which also contains a proxy for a producer location service that, for example, returns the current location of an object by consulting the corresponding base location service. The main task of the producer service is to update its hub entry, so it can be asked for it, and to provide an initial instance of a representation whose code is retrieved by a mapping between identifiers and Java classes.

Object Manager The object manager can be regarded as the heart of the whole system, since it provides the actual execution environment for the representations, so that it is comparable with the hosting service in our model. An object manager registers itself at the base location manager to be informed about the appearance and disappearance of tagged things. These asynchronous notifications are handled by a dedicated event manager. In comparison with our model, the event manager corresponds to the event bus service. Since one of our requirements for the Iceo system was that it needs to be efficient, the object manager uses a task manager to process the events instead of creating an individual thread for every event and every representation as in the Voxi system. For every event there is a corresponding task, e.g. an entry task for an entry event or a migrate task if a representation needs to be migrated. Depending on the task load, the task manager can create or shut down worker threads that process the tasks. An object manager itself generates events to inform other object managers about the treatment of a representation: a manage event indicates that it started to manage a representation, a migrate event says that a representation will only be managed for the started grace period and finally, an unmanage event is generated after the grace period has elapsed and the representation has been locally deleted. With these three events and the two events from the base location service, every object manager administrates a state machine for every representation that has been mentioned in one of these five events. Finally, an

object manager retrieves a representation instance from another object manager where the representation is in the grace state, or an initial version from the producer. After having received the representation via RMI, an object manager triggers the representation with a manage event to start and a migrate event to stop its execution. Figure 5.11 shows the structure of an object manager. The management of representations, including starting and stopping of a representation, implements the procedure of our model.

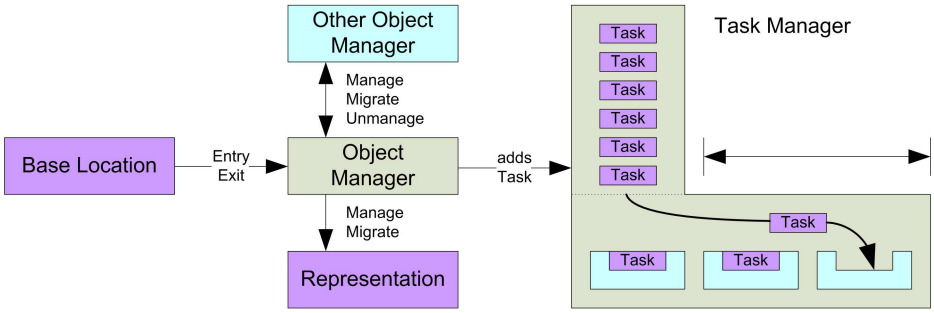

Figure 5.11: Structure of an object manager

Element	Type	Realization	Main tasks
Tag detection system	FW	Jini services	basic support for identification, localization, sensors, actuators
Base	MW	Jini services	location management for representations
Hub	MW	Jini services	bridge between bases and producer
Producer	MW	Jini services	provides code knows current hub
Object manager	MW	Jini services	executes representations
Representation	FW	Java object	encapsulates additional functionality

Table 5.3: Summary of Iceo elements

Representation The actual representation is a simple Java object that must be extended for smart thing specific behavior: it has references for two optional communication channels to control its sensors and actuators, it knows its identifier, it has a reference to its object manager and an additional reference to a storage provider that is comparable with the data service in the model which can be used to store and retrieve state information. Since the characterized functionality is quite limited, a representation provides a generic way to inform others about its capabilities by stating the Java interfaces it implements. In the same way, it provides a generic mechanism so that applications and remote representations can access the representation: if an entity needs to be contacted that is not executed in the same JVM as the representation, an RMI object is created that handles the communication with the local representation. A last important aspect of a representation is the event notification: a representation provides a method to be informed about starting and stopping and about any other event it has registered for.

Normally, a representation does not have its own, costly execution thread, since the event notification is handled in the event handler thread, which can be used by the representation for short actions. If a representation or an application wants to communicate with another representations, both can contact the bases for a reference. Thus, the representation in Iceo is a complete implementation of the representation concept of our model. In contrast to Wsst, Iceo supports migration of representations.

5.4.3 Comparison with the smart thing model

The third smart thing system, Iceo, comes very close to the smart thing model. In comparison with the other systems, it provides generic support for the tag detection system and provides communication channels between the tag and the representation to control sensors and actuators. The base tree roughly corresponds to the location manager services of the smart thing model, the object manager represents the hosting service, and the producer stands for the home service. In a final implementation of the smart thing model, the base tree of Iceo and the location managers of Wsst need to be merged. Additionally, the full STILM model and the three extensions: containedness, simultaneous detection of the same tag, and smart things with multiple tags needs to be properly supported. The containedness and composition relation has not been implemented in Iceo, since it has already been tested in Wsst, but smart things with multiple tags are already partly supported in Iceo. Table 5.3 summarizes the main tasks of the different elements of the Iceo system.

Chapter 6

Evaluation of the Smart Thing Systems

The overall goal of smart thing systems as has been explained in Chapter 2 is to enable and to facilitate the development of applications that make use of smart things. In this chapter, we want to evaluate the smart thing systems presented in Chapter 5 and the concepts of Chapter 4, which these systems implement, with the smart supply chain application that has been introduced in 2.3. Our evaluation consists of two parts: first, we want to evaluate qualitatively the development of an application in order to determine whether and how the systems actually facilitate the development of smart thing applications. Second, we want to evaluate the systems in a quantitative manner to find out whether they are suitable for deployment in real world scenarios.

6.1 Qualitative evaluation

The qualitative evaluation considers all aspects concerning the setup of middleware services and the actual development of the business logic in smart things and dedicated applications. As a generic example, we chose the smart supply chain application, since it covers different locations and hierarchies of smart things, and this application comes close to an application that can be deployed in reality. First of all, we describe the common procedure used in every smart thing system for developing an application. We then describe how we modeled the business logic that is independent from the smart thing system deployed. Since Voxi and Wsst do not provide any support for a tag detection system, we describe the RFID framework that was originally developed for simple RFID applications to work independently of an arbitrary underlying RFID hardware. We connected the RFID framework with all three systems in order to receive entry and exit events from monitored RFID tags. After this preparation, we describe the deployment of the smart supply chain application and the setup of the middleware services, first with the Voxi system, followed by the Wsst system and finally with the Iceo system.

6.1.1 General deployment scheme

Besides the development of the business logic, we have to consider some more aspects that mainly refer to the installation and the setup of the tag detection system. The following list summarizes the important aspects concerning the tag detection system:

- attaching the tag to the thing
- loading an identifier on the tag
- installing the tag readers in the physical environment or within a smart thing
- associating a symbolic location name to the tag detection service
- setup the localization policy of the tag detection system

It also depends on the deployed technology whether the attachment of the tag or the upload of the identifier comes first. A bar code, for example, always needs to encode and print the identifier first before it can be attached to a thing. An RFID label, on the other hand, might be programmed after it has been attached to a thing. The installation of the tag readers normally require that a symbolic location name is associated with the tag detection service that acts as address for all detected tagged things in the detection space, but this is not absolutely necessary, since a tag detection system or the thing itself could determine the physical position. Which localization technology is finally used is determined by the localization policy, which has to be interpreted by the tag detection service. The environment can be extended step by step by attaching new tags to things and by installing new tag readers. This means in turn that a start with only one tag reader and several tagged things is possible. While the installation of the tag readers, which should occur rather rarely, always requires manual work, the programming of the tags and their application on the thing can be automated within the production process of a thing.

After the hardware issues have been addressed, we next have to setup the middleware services. This roughly comprises the following tasks:

- startup and setup a home service on a computer
- download the representation code to the home service
- association of the identifiers with the code at the home service
- startup and setup of hosting services
- startup and setup of bases, hubs and super location managers
- connecting location managers, tag detection services and hosting services

The startup and setup process of one of the above mentioned entities also includes the setup of the underlying network technology. If Jini is used, the necessary components such as the RMI daemon or the lookup service need to be started. In general, the computers where the services run should be configured in such a way that they restart the services after a reboot. The startup of a home service is a rather rare event, since it is only necessary when a new provider of representations comes on the market or if an existing provider of representations gets another home address for its representations. In this case a new home service is required, i.e. for every home address that is part of the stored identifier on a tag, there is exactly one home service. More frequent is the introduction of a new representation. In that case, the static code has to be uploaded to the home service. Most frequent, though, is the registration of a new identifier at the

home service. Such a registration is needed to map an identifier to its code. The startup of hosting services as well as location managers normally also occurs rather rarely, since such a startup is mostly a consequence of the installation of a new tag reader. The setup of location managers requires the allocation of symbolic names and physical positions: while the former is mostly straightforward, e.g. one takes the room number as name, the latter can be more complicated, since the exact physical coordinates of a certain space are not always easy to determine. The actual connection between the location managers, the tag detection service and the hosting service is straightforward and can be comfortably made with a GUI by an administrator that is responsible for the deployment of that system at a site (location domain). The only task that might be automated is the association of the identifier with the code at the home service: if a tag is automatically applied on the thing and if the production process knows which code should be used for the products it is producing, the production process can automatically register the identifier at the home service.

The final and last block in the deployment of a smart thing system is the actual development of the business logic which roughly comprises the following tasks:

- development of the representation
- development of the application
- connection with proprietary systems

First, it is necessary to determine how to distribute the business logic over representations, smart things applications and existing applications. Normally, it is clear whether a function is part of a smart thing or of an application that solely communicates with smart things, but this is not really predetermined: in the smart supply chain example, the smart things can either check themselves in with the existing warehouse management system, or an application that monitors the check-in area checks them in. Since existing systems normally cannot communicate with smart things, an additional application that acts as bridge is necessary. However, existing systems and even applications are not mandatory: the whole business logic could be realized only within the smart things. Normally, the development of new representations occurs more frequently than the development of new applications, since applications normally cooperate with classes of different smart things: the application that monitors the check-in area is developed once, but it is able to communicate with smart things that have been developed later.

In summary, we can state that the deployment of a smart thing system comprises three major tasks:

- installation of the tag detection system
- startup and setup of middleware services
- development of representations and applications

6.1.2 Implementation of the business logic

No matter which smart thing system is used to develop the smart supply chain application, the same aspects have to be considered and the inherent business logic that has to be modeled is the same for every system. The goal of the smart supply chain application

is the automation and verification of all steps within a supply chain in order to reduce the costs and to increase the operation speed. In our small supply chain, a retailer can send orders for mineral water from different bottlers to a wholesaler. The wholesaler that processes the order has to check the availability of the ordered mineral water and to initiate the transport from its storage areas to the check-out area. There, the goods are loaded onto an ordered truck that drives the goods to the retailer where they are unloaded. Finally, the retailer has to move the goods from its check-in area to its storage area. In every step, we want to check whether the right product instances have passed a predetermined location, and the products should be able to ascertain the temperature in order to check whether they are stored at the right temperature. The same procedure also applies to the orders between the wholesaler and the two bottlers – the detailed procedure is described in Chapter 2.3. In order to implement the smart supply chain application, we differentiate between five components:

1. warehouse management systems at different locations
2. orders that are sent between these locations
3. order management
4. monitoring application
5. representations

We need a warehouse management system at every location that manages the products at this location, i.e. which products are currently stored there. We also need to check whether the right product instances come in and go out. The second point means that the orders that are sent between the different entities need to be modeled, and since they have to be processed, the order management has to be modeled separately. We need a monitoring application that allows us to see what happens at the different locations and that allows us to send the orders between a retailer and the wholesaler or between the wholesaler and the bottlers. The last point requires that the products either have to communicate with the warehouse management system - e.g. a bottle has to communicate what its content is, in order to be properly stored - or they have to communicate among themselves, e.g. a bottle asks a thermometer for the current temperature.

Orders We differentiate between four messages that are sent between the different locations: a *goods order* states which products and how many are ordered, a *transport order* states that a truck needs to drive the goods from A to B, a *transportOK* is used to signal that a truck has arrived at a certain location, and finally an *ASN* containing the identifiers of the products that have to be shipped notifies all involved parties which product instances to expect. As Figure 6.1 shows, the common feature of all orders is that they have a sender and a receiver as well as a sequential number. A goods order additionally knows which products and how many have been ordered. An ASN additionally knows the identifiers and the corresponding goods order. A transport order just has an additional reference to a goods order that triggered that transport and a transportOK only needs to state to which transport order it belongs to.

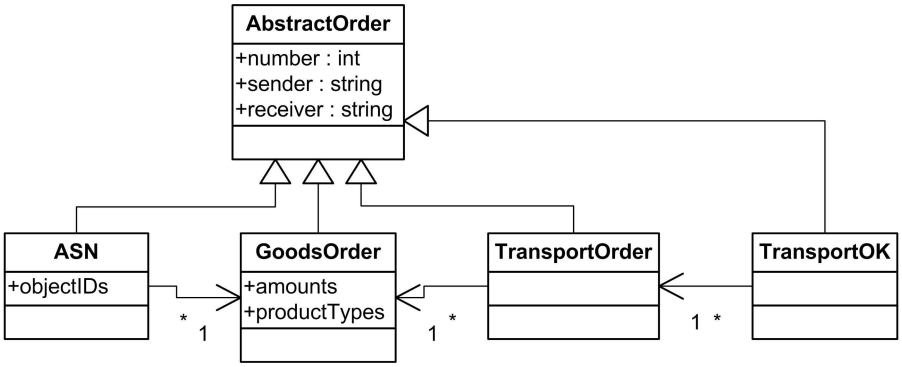

Figure 6.1: UML diagram of all order types

Order management Every location that is able to receive at least one of the above orders needs an order management system that stores and processes the orders. Such an order manager differentiates between four states of an order, which can be seen in Figure 6.2: pending, processing, processed and current. Pending means that the order has been received but still needs to be processed, processing means that the order has been partly processed and some information is still missing to complete the processing of this order, processed means that the order has been finally processed and current refers to the message that is currently being processed by the order manager – only one order can be the current order at any one time. A second task of the order manager is the generation of the sequential number for an order: starting from 0, after every request, the counter will be increased by one. This scheme only allows the sequential number to be unique for one sender, since each sender has its own order manager, so that the same sequential number might be generated many times by different senders. A third task of the order management system is to check whether the right number of product instances or product types enters or leaves the location it is responsible for. Normally, the order manager works on the instance level. If it expects some products to arrive, then it knows the concrete product instances from the ASN. If it expects some products to leave, then it can recognize the product instances that are leaving the field. There is one exception when it cannot work on instance level: if a goods order arrives at the storage area where the product instances are actually stored, the order manager can only check whether the right amount of the ordered product type has been taken out, but this reflects the intended behavior in such a case. In order to fulfil this task, an order manager differentiates between the three states in, out and none: in means that it expects products to enter its location, out means that it expects products to leave its location and none means that currently no order is active. If it works on the product instance level, it uses a set of identifiers, otherwise, it uses a counter for every product type. Additionally, it knows which product instances are in the field to compare these with the set of identifiers or the product type counters.

Warehouse management system Every participant in our supply chain has different internal locations that it has to manage: the wholesaler has one check-in area, two storage areas and one check-out area, a bottler has at least a check-out area, a retailer has at least one check-in area and a truck has a loading space. Every location can be the sender or the receiver of the four different order types and therefore every location uses an order manager to handle the orders. Although the order manager takes care of a lot of

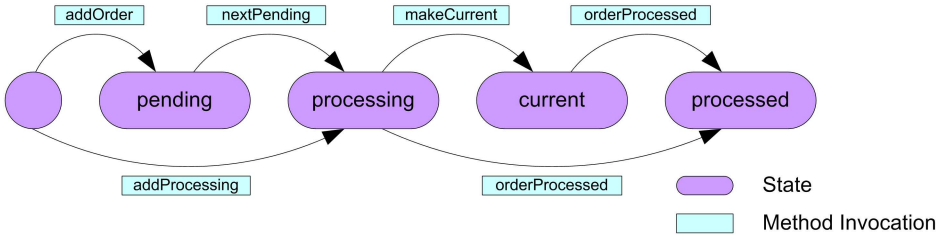

Figure 6.2: State machine for an order

the management work that is needed to manage the goods at a certain location, every location has its own procedure for the treatment of goods. A check-in area, for example, always tries to forward the goods to the storage area, in order to be able to process the next delivery, or another example is a truck that cannot load or unload goods while it is driving. Thus, every location needs its own warehouse management system that takes into account the special needs of a certain location. To handle the general management aspects, a warehouse management system uses an order manager.

General tasks that need to be fulfilled by a warehouse management system comprise the notification of interested parties about certain events, e.g. the reception of an order, the completion of an order, or concrete instructions, such as the request to move the goods from the check-out area to the truck. In our case, these messages will be displayed by the monitoring application. Another task is the subscription for entry and exit events of smart things that enter or leave the location the warehouse management system is responsible for. After the reception of these events, the warehouse management system forwards these events to its order manager, which determines whether the order has been processed. In such a case, the warehouse management system can signal that all products are either in or out, depending on the type of order, and the next order can be processed. In addition, the monitoring application will be notified. All other functions are then location-specific. Figure 6.3 shows the location-specific state machines of the warehouse management systems concerning the mentioned states: in, out and none.

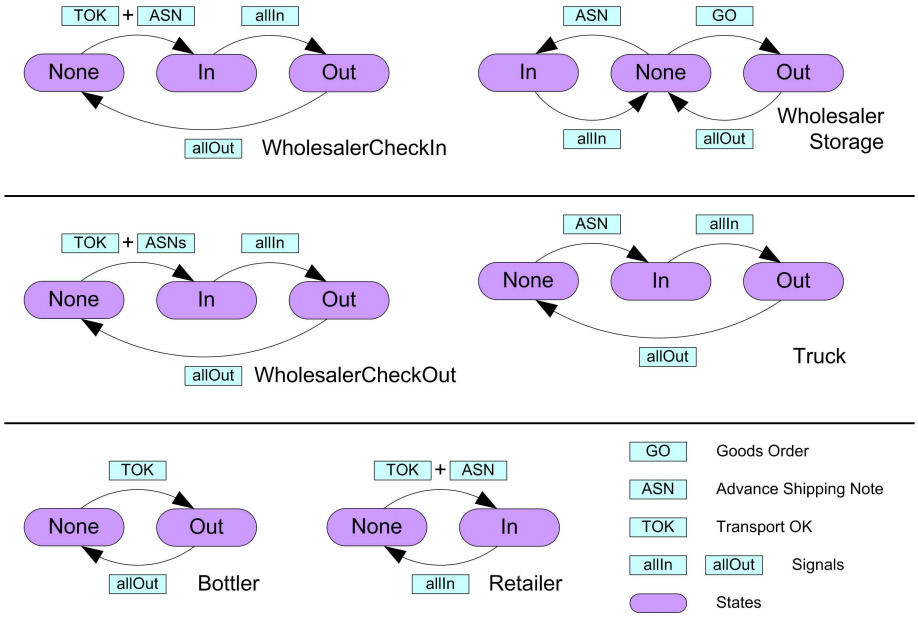

Figure 6.3: State machines for warehouse management systems

After a bottler has received a goods order, it performs several tasks: first, it calculates the amount of boxes and containers that are needed to transport the goods, second, it checks whether it has enough bottles, boxes and containers, third, it sends a transport order to its forwarder. The reception of a transportOK from the ordered truck changes the state from none to out. While the goods are loaded into the truck, the bottler checks whether the right amount of product types have been taken out. Finally, the allOut signal again changes the state from out to none and the bottler sends an ASN with the identifiers of the smart things taken out to the truck and to the wholesaler.

Trucks that receive a transport order start driving to the receiver of the goods order associated to the transport order. When they arrive at the destination, they send a transportOK there to signal that they have arrived. They then wait for an ASN that states which product instances have to be loaded. The ASN also triggers the change of the state from none to in. After receiving the allIn signal, the truck drives to the orderer of the goods and changes the state from in to out. Having arrived at the final destination, it again sends a transportOK and waits until the goods are unloaded: this event is signaled by an allOut which in turn triggers the state from out to none.

A wholesaler's check-in area can receive a goods order from the monitoring application that the wholesaler only needs to forward to its bottlers. A human user can trigger the goods order with the monitoring application. Next, the check-in area waits for an ASN that corresponds to the forwarded goods order and for the transportOK of the truck that is delivering the ordered goods in order to change the state from none to in. It checks whether the right product instances have arrived by means of the ASN. The allIn signal triggers the state change from in to out. Next, the goods have to be moved to the different internal storage areas. Depending on the smart thing, the check-in area knows the right storage area for each product type. Every storage area receives an internal ASN that states which product instances have to be moved from the check-in area to each storage area. Finally, the allOut signal changes the state from out to none and the check-in area can process the next delivery.

Storage areas of a wholesaler are the connecting location between the goods order from the wholesaler at its bottlers and the goods order from a retailer at its wholesaler. In the former case, the goods are brought into the wholesaler's storage area and in the latter case, the goods are taken out of the storage area. After the reception of an internal ASN, its state changes from none to in, and the storage area checks whether the right product instances have arrived at its location by means of the internal ASN. The allIn signal triggers the state change from in to none. If an internal goods order from the check-out area arrives, the storage first checks whether it has the requested number of product types available and changes its state from none to out. The allOut signal changes the state back to none and an ASN with the smart thing identifiers is sent to the check-out area.

If a wholesaler's check-out area receives a goods order from a retailer, it sends a transport order to its forwarder, calculates the number of boxes and containers that are needed to transport the goods, and depending where the product types are stored, it sends internal goods orders to its storage areas. After reception of the internal ASNs from its storage areas and the transportOK from the ordered truck, the state changes from none to in. Until the allIn signal occurs, the check-out area checks the product instances by means of the internal ASN. If the allIn signal then occurs, the state changes to out and the check-out area waits until the products have been loaded onto the truck. Finally, the allOut signal triggers the state change to none, so that the check-out area

can process other goods orders.

The last location that needs a warehouse management system is the retailer. Analogously to the check-in area of the wholesaler, the retailer forwards goods orders coming from the monitoring application to the wholesaler. It receives an ASN as well as a transportOK as the answer to the forwarded goods order that triggers the state change from none to in. By means of the ASN, the retailer checks whether the right product instances have been unloaded from the truck. The allIn signal then triggers the state change back to none and the retailer is able to receive new deliveries.

Representations Much of the business logic is already covered by the warehouse management systems, which use order managers to manage the orders sent. In order to cooperate with a warehouse management system, the identification and localization abilities of a smart thing are necessary, but these are rather passive abilities of a smart thing, since the identification and localization are mainly actively driven by the infrastructure. Both abilities are necessary, since the warehouse management system has to ascertain the identifiers of the smart things within the location it is responsible for. Besides these passive abilities, a smart thing has to store and retrieve its state, and it must be able to calculate a statistic concerning the retention period at its visited locations. It should also be able to inform the monitoring application about its state.

A bottle, additionally, has to know its content, i.e. whether it is mineral water either from LidWaters or from OpenWaters. The warehouse management system needs this information to properly store and retrieve the bottles. A bottle also should record the temperature for quality reasons. This can be done in three ways: either it asks the warehouse management system for the current temperature, or it controls a sensor, or it asks one of its neighbors.

Monitoring application The final component that we need for our smart supply chain application is the monitoring application. This serves three purposes. First, it can be used as browser for the locations that are managed by a warehouse management system and the representations within, e.g. the monitoring application shows the current goods order at a certain location or it shows the statistical analysis of a representation. Second, it displays the messages of the warehouse management system, such as the request to start loading a truck or an alarm message of a representation stating the temperature is out of a predefined range. Third, the monitoring application provides the means to cause a goods order to be sent from the retailer to the wholesaler or from the wholesaler to both its bottlers.

6.1.3 RFID framework

Since only the Iceo system provides direct support for tag detection systems, i.e. for bar codes and BTNodes, we used a system that has been previously developed to support local RFID applications. By local we mean the RFID hardware, the RFID framework and the application are connected, or run on the same machine. One application we developed was the smart surgical kit: a surgical kit contains many small swabs that can be left behind inside the patient during surgery. To detect such situations we equipped every swab of the kit with an RFID tag and installed one RFID reader under the surgical kit and another under the waste bin where the used swabs are disposed. Since the smart surgical kit application knows the content of one surgical kit, it can infer whether an item

is still in the kit, has already been thrown away, or it can infer that a swab is still in use. In such applications, the RFID framework allows the application to work independently of the underlying RFID hardware and allows applications to register for entry and exit events. The assembly and a screenshot of this application can be seen in Figure 6.4. This application has been developed for one of the M-Lab partner companies as a technical demonstration. It does not make use of the concepts presented here. Its purpose is to show how an application can make use of the RFID framework.

Figure 6.4: Assembly and screenshot of the smart surgical kit application

The framework itself consists of three layers: the hardware abstraction layer (RFID Controller), the dispatcher layer and the filter layer as can be seen in Figure 6.5.

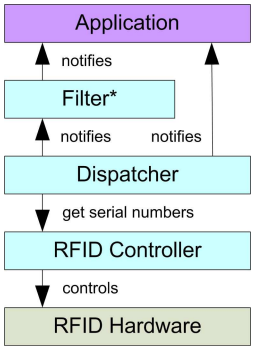

Figure 6.5: Layers of the the RFID framework

The hardware layer abstracts from the underlying RFID hardware by providing an abstract controller class that needs to be extended for a specific piece of hardware. This controller continuously polls the hardware-specific RFID reader for RFID tags in its field, reads out their serial numbers and puts them into a queue. On the next layer, the dispatcher layer, the dispatcher takes the serial numbers out of the queue and generates an entry event if the tag was not seen in the last reading cycle. For every tag of the last reading cycle that has not been detected in the current reading cycle, an exit event will be generated. These events are delivered to all registered filters of the filter layer. Several filters can be plugged together, since a filter only needs to understand the two RFID events. In our case, we only used one filter to smooth spurious exit/entry event combinations. One source of these spurious event combinations is the nondeterministic behavior of most tag detection algorithms: first, the tag reader announces the number of time slots in which the tags can transmit their identifier. Second, the tags randomly choose one time slot to transmit their identifiers. Due to the probabilistic choice of the

time slot, two tags might choose the same time slot to answer and a collision occurs, so that the tag reader cannot detect both tags in this detection cycle [114, 115]. In the next detection cycle, both tags might be detected properly, so that a combination of an exit and an entry event occurs due to the collision in the second reading cycle, as Figure 6.6 shows. The smoothing filter delays the forwarding of an exit event: it puts all exit events in a list and forwards them only after a predefined time window no entry event for the same tag has occurred. If this happens, both events, the previous exit event and the entry event, will be disregarded. This scheme means a trade-off between real-time requirements and the occurrence of spurious event combinations. Finally, the applications of the application layers will be notified of the entry and exit events that have gone through the filters.

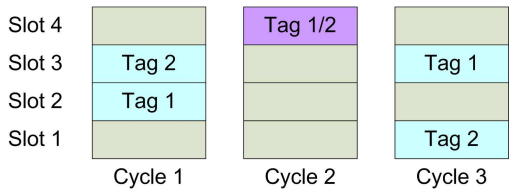

Figure 6.6: Collision of two RFID tags

6.1.4 Voxi

The Voxi system, which is written in Java and builds on Jini, uses a PostgreSQL database to enable its artefact memory to store the data persistently. In order to run the Voxi system, we installed the following software and hardware systems:

- Java 2 SDK, Standard Edition, Version 1.4.0
- Jini Technology, Version 1.2.1
- PostgreSQL, Version 7.2.3
- Softronica RFID reader
- Philips i-Code compatible RFID tags

In the following, we will describe the deployment of the smart supply chain application step by step, as has been proposed in the general deployment scheme. This consists of three phases: installation of the tag detection system, startup and setup of the middleware service, and development of representations and applications.

Installation of the tag detection system Identifiers in Voxi (objectIDs) denominate the Java class that encapsulates the additional functionality of a smart thing. First, we glued the RFID tags on the 24 bottles, 4 boxes and 2 containers and then electronically wrote the corresponding class names on the tags: Bottle1..24, Box1..4 and Container1..2. Next, we had to develop a tag detection service that controls the RFID reader and generates the entry and exit events that are processed by the virtual object manager. For that purpose, we simply developed an application that acts as a bridge between the RFID framework and the event source of the Voxi system, as Figure 6.7 shows. This bridge

registers itself as an application in the RFID framework and receives the corresponding entry and exit events with the above mentioned objectIDs. One of the bridge's startup parameters is the locationID, so that the bridge can use the locationID as well as the objectIDs from the RFID frameworks to notify the virtual object manager about entry and exit events. In total, we need nine locationIDs: DistributeAllCI, DistributeAllCO, DistributeAllStorage1, DistributeAllStorage2, FastDelivery, OnTimeDelivery, OpenWatersCO, LidWatersCO and MigrosCI. Since we have only five RFID tag readers for nine locations, we have to run the application in two phases, so that one RFID reader has to manage two locations one after the other. We also need to configure the bridge with the name of the virtual object manager which is the receiver of the two event types. In our case, we use the locationIDs as symbolic location names of the tag detection service.

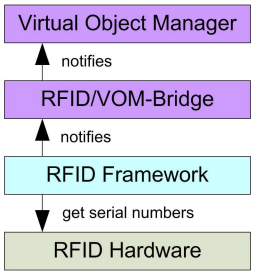

Figure 6.7: RFID/VOM-Bridge

Startup and setup of the middleware services In Voxi, the home service is realized as an HTTP server. Since we use Sun's Jini implementation, we can use the HTTP server that is shipped with the Jini implementation. One of the HTTP server's startup parameters is the directory that contains the files offered by the server. Thus, all the downloadable code for the Jini infrastructure as well as the static code of the representations needs to be copied there. Next, the RMI daemon and the Jini lookup service can be started, in order to allow the virtual object managers to register themselves there. Every participant in the supply chain has its own virtual object manager: OnTimeDeliveryVOM, OpenWatersVOM, LidWatersVOM, MigrosVOM, DistributeAllVOM, and FastDeliveryVOM, which are all started up then. In order to allow a bottle to retrieve the current temperature, we registered a smart thermometer at every one of the nine locations. Since we have no physical thermometer, we simulated such a smart thing with a random number generator. The smart thermometers can be registered by creating an entry event with the objectID of the thermometer and the locationIDs of the nine locations. Then, we have to start the monitoring application for every participant in the supply chain, to allow the inspection of their locations. Finally, we have to start the artefact memory at the location Virtopia by generating a corresponding entry event – the PostgreSQL server has to be started in advance.

Development of representations and applications The orders, the order management and the monitoring application are independent from the Voxi system and can be implemented straightforwardly, so that we only introduce the representation and the warehouse management system.

First, we developed a virtual product class as an extension of the virtual object class that is provided by the framework. When this virtual product class receives the entry

event, it contacts the artefact memory to retrieve its state, and when it receives an exit event, it writes its state to the artefact memory. Additionally, this class provides the option to ascertain the actual state and the statistical data of the smart thing in order to allow this information to be displayed by the monitoring application. From this class, we derived the three specialized classes Container, Box and Bottle. Since all three classes introduce new fields, the storing and retrieval of the state data at the artefact memory needs to be extended. A bottle additionally registers itself for temperature events at its location after it has received an entry event and a stop event triggers a bottle to unregister for temperature events. Temperature events are published by the smart thing thermometer that is available at every location. The bottle records the temperature data and broadcasts a temperature alert if the temperature exceeds a certain limit. This shows the neighborhood concept in Voxi: a bottle and a thermometer that are registered at the same location can communicate through events. A bottle can also state its content in order to be properly handled by the warehouse management system. For every smart thing instance we had to extend a separate class that corresponds to the objectID, i.e. we had to extend the Bottle class 24 times, with only the class name differing.

Next, we had to develop the warehouse management system for every one of the nine locations. Since virtual locations are a specialization of virtual objects, we can implement the warehouse management system as an extension of the basic virtual location class that is provided by the framework. The virtual product location class extends the virtual location class and provides the basic capabilities of a warehouse management system. It instantiates an order manager to handle the orders. The orders are sent as regular Voxi events between the warehouse management systems. Additionally, the virtual product location generates the allIn and the allOut signals by calling two abstract methods that need to be implemented by subclasses. Besides managing the identifiers, this class also counts the smart things of a specific product type. Boxes and containers are classified as boxes and containers by their objectID, which contains the corresponding string. If a bottle enters a location, the bottle will be asked for its content and classified according to its content: LidWaters or OpenWaters. Next, we extended the virtual location class six times: Producer, Retailer, Truck, WholesalerCI, WholesalerCO and WholesalerStorage and implemented the warehouse management location-specific behavior. These six classes again had to be extended with the nine locationIDs. Here, we only had to specify instance-specific behavior, e.g. LidWaters takes OnTimeDelivery as its forwarder and OpenWaters takes FastDelivery as its forwarder. Figure 6.8 shows the monitoring application at DistributeAll with its four locations and detected smart things at the DistributeAllStorage2 location.

Experiences It was not possible to model the composition relation with the support of the infrastructure, so that we do not include the four handles. Since the containedness relation is not supported either, we did not model the containedness of the box and the containers, but modeled them as regular smart things. We only used event communication, which on the one hand, was very easy, since we only had to specify the objectID or the locationID as the receiver of the event and all communication details were transparently handled by the event delivery manager. On the other hand, if we wanted to send complex objects, we had to do the marshaling and unmarshaling of the complex objects by hand, which is a very cumbersome task. Another problem is the distribution of the application. Normally, each one of the five participants has its own subnet with its own lookup service, so that our example application cannot work in a really distributed

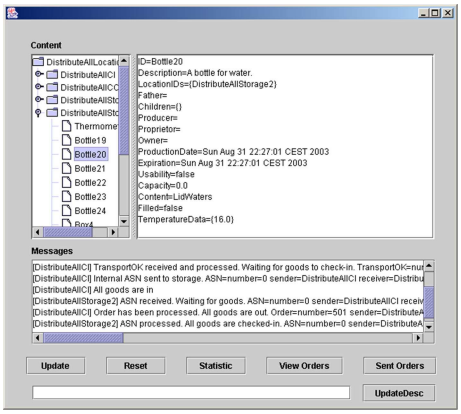

Figure 6.8: Screenshot of Voxi's monitoring application

way for several reasons: first, the artifact memory is a central service, second, the event delivery manager can send events only to local entities, and third, the virtual object repository is a central service, too. However, we were able to show that the middleware services work on principle, and that the concepts virtual objects (representations) and virtual locations (location-dependent services) fit and provide an appropriate means for developers to program smart thing applications.

6.1.5 Wsst

The Wsst system that is written in C# builds on Web Services that make use of SOAP and UDDI. In order to run the Wsst system, we installed the following software and hardware systems:

- Microsoft .NET Framework SDK, Version 1.1
- Microsoft Internet Information Services, Version 5.1
- SoapUDDI
- Microsoft Access ODBC drivers
- Apache Tomcat, Version 4.0.4
- IBM Web Services Toolkit, Version 3.0
- Softronica RFID reader
- Philips i-Code compatible RFID tags

Installation of the tag detection system Since we already glued the tags on the things for the evaluation of the Voxi system, we were able to use the tagged things again. Due to the fact that the Wsst system supports composition, we tagged the four handles of the two boxes, too. In order to write the URIs on the tag, we first had to register the representations at the UDDI server to retrieve the UUID that is part of the URI. The URL part of the URI refers to the UDDI server where the representations are registered. In our case, all representations are managed by packaging.plast, which refers to the producer of the smart things that are all made of plastic. In total, we had to

write the URIs of 34 smart things on the corresponding tags. Similar to the approach in Voxi, we developed a bridge between the RFID framework and the Wsst system: if the bridge receives an entry event from the RFID framework, it interprets the serial number as URI and resolves the URI at the UDDI hierarchy to a URL of the corresponding representation. Since the bridge is developed in Java, we used the IBM Web Services toolkit classes to call the set-location-method of the corresponding representation based on SOAP. Each bridge needs to be configured with a symbolic name of the tag reader's coverage space and the physical position of the tag reader. Both items of information are used to call the set-location-method of the representation. Since we now have twelve locations and only five RFID readers, we developed a GUI where the user can select one of the twelve locations that is used to update the representation.

Startup and setup of the middleware services First, we had to start up the twelve UDDI servers for the following domains: Ω, trucks, trucks.man, packaging, packaging.-plast, retailer, retailer.migros, food, food.openwaters, food.lidwaters, wholesaler, wholesaler.distributeall, which are also illustrated in Figure 6.9. This list shows that every participant in our supply chain has its own UDDI server where at least its locations are registered and we have the producer of the trucks where both smart trucks are registered as well as the producer of the packaging, where our 34 smart plastic parts, i.e. boxes, containers, bottles and handles are registered. We developed a script that automates the registration at the UDDI server hierarchy. For each of these twelve UDDI servers we had to configure the Apache Tomcat web server and to create a Microsoft Access database which is accessed via JDBC/ODBC by the SoapUDDI implementation to persistently store the UDDI data. Ω refers to the root UDDI server.

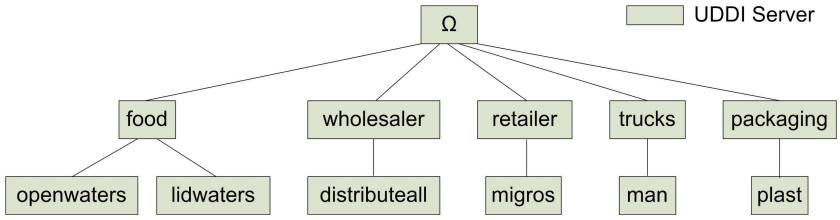

Figure 6.9: UDDI hierarchy in the Wsst smart supply chain application

Next, we set up the super location managers that do not represent coverage spaces of tag detection readers, as Figure 6.10 shows. As in every ordinary web service, we also had to register the eleven super location managers at the UDDI hierarchy: ω at Ω, mobile at Ω, ch at Ω, ch.zurich at Ω, ch.zurich.migroscity at retailer.migros, ch.zermatt at Ω, ch.zermatt.openwaters at food.openwaters, ch.stmoritz at Ω, ch.stmoritz.lidwaters at food.lidwaters, ch.olten at Ω, and ch.olten.distributeall at wholesaler.distributeall. The very first ω refers to the world location manager. As the names indicate, MigrosCity is located in Zurich, OpenWaters is located in Zermatt, LidWaters is located in St. Moritz and DistributeAll is located in Olten.

In the Voxi implementation, we had nine locations in the supply chain that were extended to twelve locations with host services in the application for Wsst - with the storage of the two bottlers and the retailer - to make the supply chain complete. These twelve location managers, whose locations are covered by tag readers, also need to be registered at UDDI servers: mobile.fastdelivery at trucks.man, mobile.ontimedelivery at trucks.man,

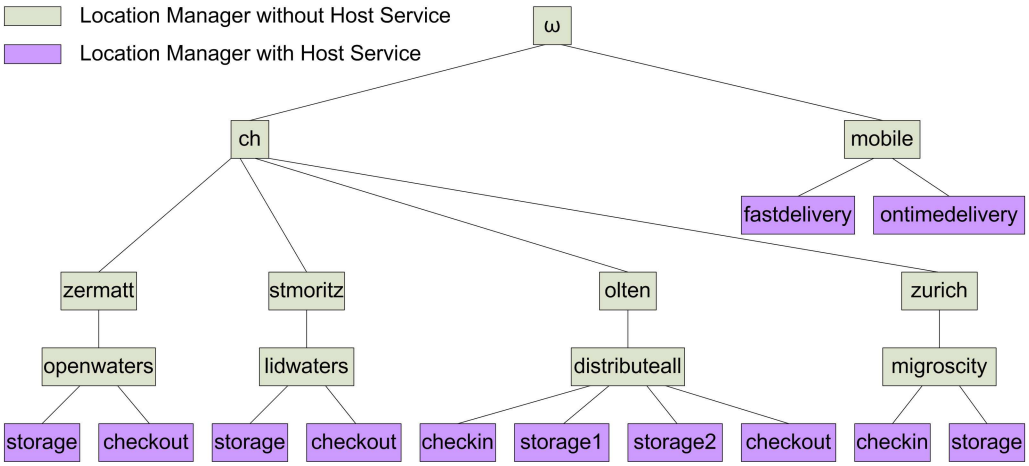

Figure 6.10: Location managers in the Wsst smart supply chain application

ch.zuerich.migroscity.checkin at retailer.migros, ch.zermatt.openwaters.checkout at food.-openwaters, ch.stmoritz.lidwaters.checkout at food.lidwaters, ch.olten.distributeall.checkout at wholesaler.distributeall, ch.olten.distributeall.checkin at wholesaler.distributeall, ch.olten.distributeall.storage1 at wholesaler.distributeall, ch.olten.distributeall.storage2 at wholesaler.distibuteall, ch.zermatt.openwaters.storage at food.openwaters, ch.stmoritz.lidwaters.storage at food.lidwaters, and ch.zuerich.migroscity.storage at retailer.migros.

Now, every smart thing and every location is registered at a UDDI server. The setup of the location managers was a little bit cumbersome due to the fact that we had to find out the WGS 84 coordinates for all locations except the both trucks. The location managers are configured in such a way that they automatically register themselves at their parent node when they are started for the first time. This means that a location manager is initialized with a symbolic name, its WGS 84 coordinates, the UDDI server address and the address of the parent location node where it has to register itself in both cases.

The Internet Information Services that host the Web Services are integrated into the Windows operating system. Normally, the web server runs by default and only the directory that contains web services needs to be marked to accept the execution of Web Services. Thus, all web services need to be copied to this directory.

Development of representations and applications The actual development of the representations and applications comes close to the one in Voxi, so that we only state the differences. In contrast to Voxi, we have no artefact memory where we can store the state data, but every representation is able to write its state data into an XML file which it can use again to retrieve an old state. Since Web Services rely on the stateless HTTP protocol, a web service object cannot use members to store its state between two method invocations. As work-around, we used the so-called application object that is generated for every web service once. It can be used to save the internal state between two method invocations. The availability of the location manager hierarchy enables us to determine the relative neighborhood concept. This function is provided by the monitoring application. Although the locations in Wsst have no behavior like the virtual locations in Voxi, we can use the host service of a location manager that specifies a smart thing,

which in turn has a behavior. Effectively, a location-specific warehouse management system can be implemented by a class that implements the location manager as well as the smart thing interface, and whose host service references itself. Each of these smart locations receives it own execution thread so that the implementation of Voxi with the virtual objects and the virtual locations can be transferred to Wsst. One main difference is in the communication: in Wsst, an entity first has to resolve the URI to retrieve the URL of the web service. This URL can then be used to instantiate a proxy object that transparently invokes the remote method of the web service.

The four handles were configured to have one of the two containers as their parent node in the composition hierarchy, so that the location update of the handles could be disregarded, since this information can be requested by the handles at their parent node.

Experiences Due to the fact that the development of the application in Wsst is quite similar to the development of the application in the Voxi system, the code of the application could be generated in a much shorter time. Although the remote method invocation is transparently handled by SOAP, which does not require the manual marshaling and unmarshaling of complex objects - unlike in Voxi with the event communication - it takes substantially longer. The URI resolution at the UDDI server is particularly time consuming, so that we used a caching scheme for the URI resolution. The download of the WSDL document enables a check at runtime whether a smart thing or a web service supports a certain method, so that the interface of a smart thing can be kept simple, since the WSDL document provides information about further protocols a smart thing supports. Although the setup of the UDDI hierarchy as well as the location manager hierarchy is complex, it has to be done only once and it allows for the distribution of the whole system, which was not possible with the Voxi system. The location manager hierarchy has proven that the relative neighborhood concept can be realized. It could also be shown that the support of the containedness relation allows the location information to be handed down. Another difference is the fact that the system only uses entry events from the tag detection system, since it only allows the new location to be set. A problem of both frameworks is that every smart thing instance needs its own class or web service. It would be desirable if one static class could be instantiated with an identifier. The main problem of the Web Services approach is the delay of the URI resolution and the SOAP invocations. The Jini lookup service and RMI are more efficient, as will be shown later.

6.1.6 Iceo

The Iceo system has been completely written in Java and again builds on Jini as the service discovery platform. One service, the producer, uses a JDBC connection to a mySQL database to persistently store its data. Besides the RFID support through the RFID framework, Iceo additionally provides support for bar codes and BTNodes. The following software and hardware systems must be deployed in order to run the Iceo system:

- Java 2 SDK, Standard Edition, Version 1.4.0
- Jini Technology, Version 1.2.1
- Java Communications API, Version 2.0

- MySQL, Version 3.23.58
- Softronica RFID reader
- Philips i-Code compatible RFID tags
- Symbol CS 2000 bar code scanner
- AnyLabel, Demo Version 1.12
- BTNodes with sensorboards

Installation of the tag detection system Besides using the RFID framework to support RFID, we also used the built-in support for bar codes and BTNodes. We mainly used RFID tags and bar codes, since we did not have enough BTNodes or sensor boards to equip all the smart things in the smart supply chain application. We again used the tagged things in the Voxi and Wsst applications, except for the containers, since they are tagged with bar codes. First, we had to write the identifiers of all the smart things onto their tag. An identifier consists of an arbitrary name and the DNS address of the producer service. As in the Wsst example, all smart things are made of plastic and have the same producer named Plast whose producer service can be reached under producer.plast.com. For the bottles, boxes, containers and handles, we used the following names: bottle1..24, box1..4, con1..2, handle1..4. Although we wanted to use the names container1..2 instead of con1..2, this was not possible, since the encoded Code 39 bar code container1@producer.plast.com was too long to be read with our symbol bar code scanner. This meant that we had to shorten the bar code to encode con1@producer.plast.com. We used the RFID framework to write the identifiers on the tag and AnyLabel to print out the bar codes that we then attached to the containers and boxes. We also attached one BTNode besides the RFID label to a bottle with the identifier bottle1@producer.plast.com. We had to change the identifier in the C source code of the BTNode code, then we had to compile the code and upload the compiled code onto the BTNode.

Next, we had to setup the tag readers. For the RFID support, we again built an RFID framework application that implements the Scanning interface, so that it can be contacted by the Iceo system. For every entry and exit event of the RFID framework, the application uses the serial number as an identifier and the location that has been given as the startup parameter to update its base location service. Since we again have twelve locations that need to be monitored, as in the Wsst system, we developed a GUI that allows the location for one RFID reader to be selected, so that one RFID reader can simulate several locations, as the screenshot in Figure 6.11 shows. Although the support of the Symbol bar code scanner was already integrated into the system, we had to develop another GUI that allows one of the nine locations to be selected, since we only have one bar code scanner. We used another BTNode as a tag reader, by connecting it via the serial interface to a computer. We also had to change the name of the location in the C code to fastdelivery.truck1, to adapt it to the location that we wanted to monitor. This changed C Code then had to be compiled and uploaded to the BTNode.

In summary, we have nine locations, which are all equipped with an RFID reader and a bar code scanner. The containers and boxes are equipped with a bar code and all the other smart things are equipped with an RFID tag. Additionally, we equipped one bottle with a BTNode tag and one location, and the FastDelivery truck with a BTNode tag reader.

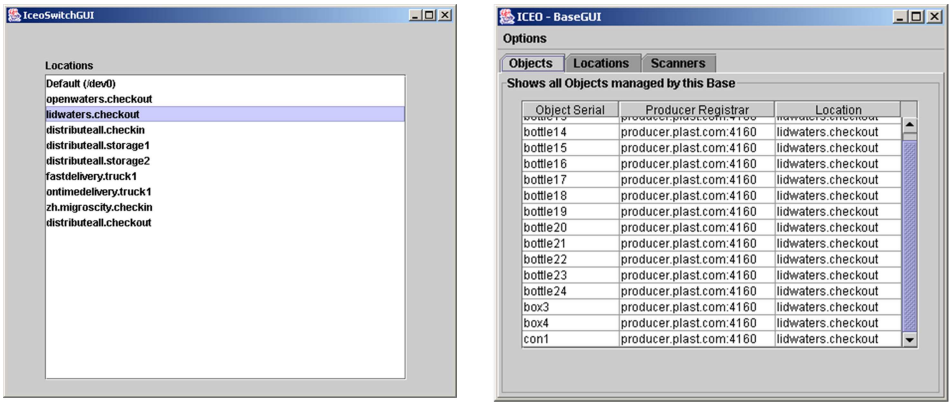

Figure 6.11: Screenshots of Iceo's RFID switcher and monitoring application

Startup and setup of the middleware services Since Iceo uses Jini, we first have to start the HTTP server for the dynamic code download as well as the RMI daemon that allows for remote method invocation on each computer. In our case, we simulated the whole supply chain on three computers. Additionally, we start its own lookup service for every base, hub and producer service, where they can register themselves as well as the services they instantiate. Next, we have to start the different services that comprise bases, hubs and producers. First, we start with the sole producer service that is responsible for all identifiers that contain its address: producer.plast.com. For this purpose, we need to start the mySQL database server. One table in the database contains the mapping between the identifiers and the Java class files, which we add with a simple SQL script running in the mySQL user client. For the startup of the producer service, we only need to specify the address of the lookup service where it can register. Every service has an additional XML configuration file where the additional services are specified. The standard XML configuration file of the producer contains the producer location service. Since we do not need any additional services at the producer site, we do not need to change this configuration file. Figure 6.12 shows a screenshot of the producer GUI, which contains the mapping between the serial number and the Java class.

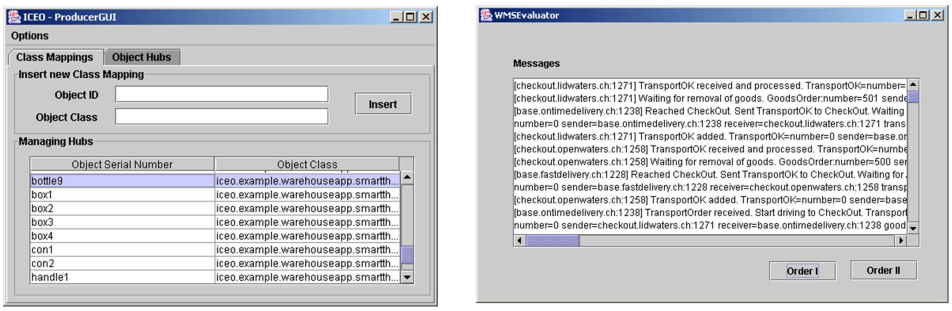

Figure 6.12: Screenshots of Iceo's producer GUI and monitoring application

Next, we start the six hub services for every participant in our supply chain, which can be accessed at the addresses: hub.fastdelivery.ch, hub.ontimedelivery.ch, hub.openwaters.ch, hub.lidwaters.ch, hub.zh-city.migros.ch, and hub.distributeall.ch. Analogously to the producer service, we need to state its lookup service and do not need to update the configuration file, which only contains the hub location service. Finally, we have to start the nine bases. In this application, every base is responsible for exactly one location. In

the following, we state the locations and the DNS addresses of each base: fastdelivery.-truck1 and base.fastdelivery.ch, ontimedelivery.truck1 and base.ontimedelivery.ch, zh.-migroscity.checkin and checkin.zh-city.migros.ch, zh.migroscity.storage and storage.zh-city.migros.ch, openwaters.checkout and checkout.openwaters.ch, openwaters.storage and storage.openwaters.ch, lidwaters.checkout and checkout.lidwaters.ch, lidwaters.storage and storage.lidwaters.ch, distributeall.checkin and checkin.distributeall.ch, distributeall.-checkout and checkout.distributeall.ch, distributeall.storage1 and storage1.distributeall.-ch, distributeall.storage2 and storage2.distributeall.ch. In this application, we have a flat hierarchy of bases, i.e. every base registers itself directly at its hub. For the actual startup of the base, we have to state the lookup service of the base as well as the lookup service of the hub, so that the base can register itself at its hub. In the case of the bases, we had to extend the standard configuration file, which already contains the object manager and the base location manager, together with the warehouse management system that is responsible for the location that is managed by this base. Finally, we need to start the scanners, i.e. for every location one bar code tag detection service and one RFID tag detection service. Besides the technical parameters on which port to find the hardware, the tag detection services also need to know the address of the lookup service of the base, so that they can register themselves at the corresponding base that is responsible for the location they monitor. Figure 6.13 shows the entities described above.

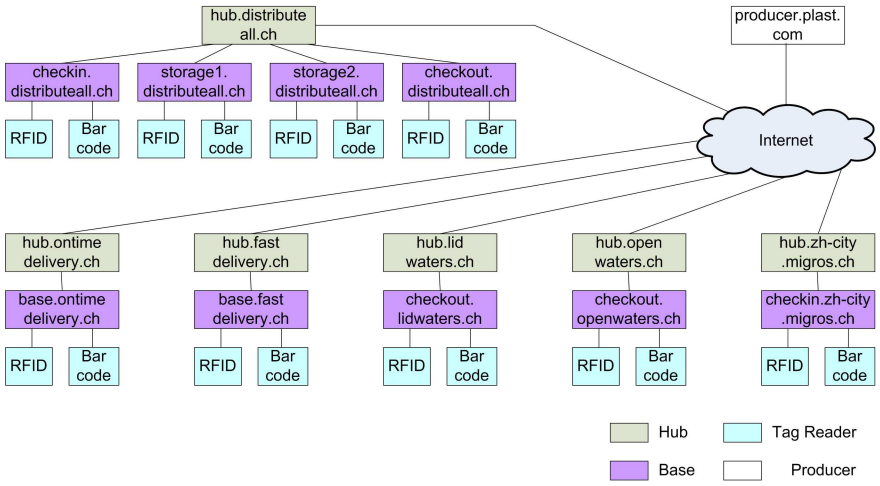

Figure 6.13: Main entities of Iceo's smart supply chain application

Development of representations and applications The actual development of the application was quite simple, since the Java code of the Voxi implementation could be reused with only minor changes. Since Iceo already provides a browser for each base (see Figure 6.11), the monitoring application (see Figure 6.12) only had to implement the other two functions: providing a window for messages of the warehouse management system and providing the buttons to send the two kinds of orders. The classes for the orders and the order manager could be used without any changes. As stated above, a location-specific warehouse management system is instantiated by a base as an additional Jini service. In this case, the super class of all specific warehouse management systems, called LocalWMImpl, short for local warehouse management implementation, asks its base for the base location service and registers itself for entry and exit events of smart things. It also has a reference to the base's object manager, so that it can contact the

representations, e.g. a bottle has to be asked whether it contains LidWaters or OpenWaters in order to classify a bottle. It creates a separate thread that handles the incoming orders. The specific code for each location could be copied with minor changes: instead of the event communication, we explicitly call methods at the other warehouse management systems to hand over an order. The implementation of the representations was even simpler, since the super class called VirtualObjectImpl handles the whole interaction with the system, so that we only had to develop the smart thing specific behavior. The actual behavior of a smart thing can be found out dynamically by checking which Java interfaces a smart thing implements. In our case, we use the following interfaces: Describable, Container, Composable, and Bulk. Describable means that a smart thing has a name, Container refers to the containedness relation, Composable refers to the composition relation and Bulk means that a smart thing can have a content. There is a default implementation for each of these interfaces, so that the actual representations only need to extend these default implementations. A handle implements Describable, a bottle implements Describable and Bulk, a box implements Describable and Container, and finally, a container implements Describable, Container, and Composable. Figure 6.14 shows the whole class hierarchy.

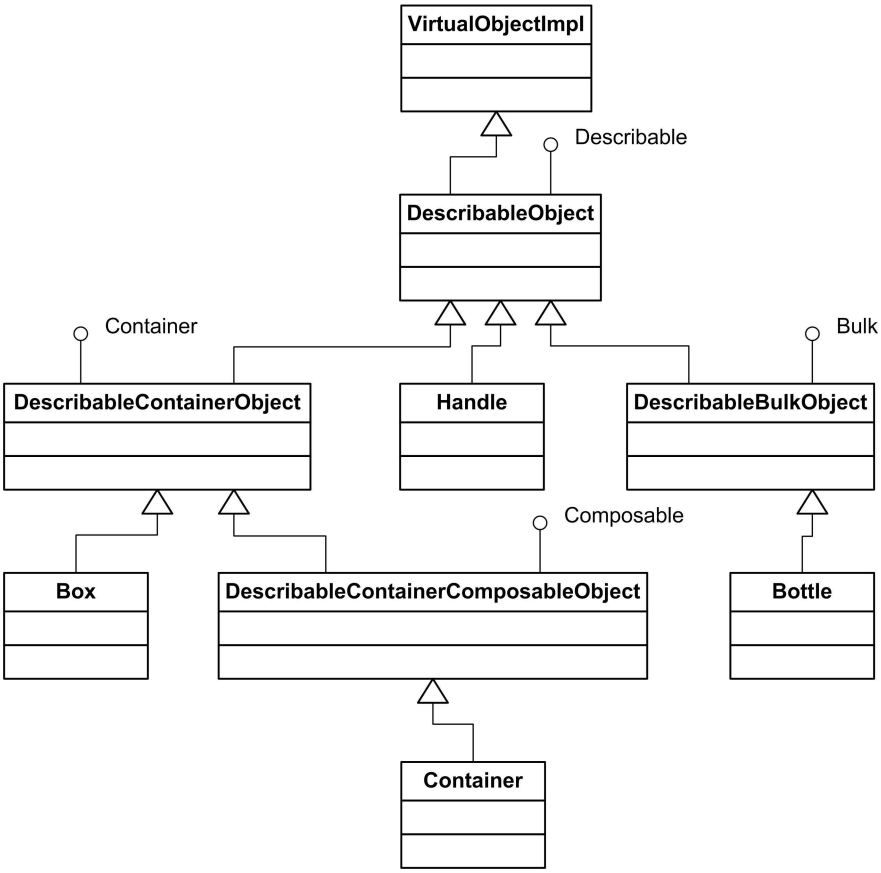

Figure 6.14: UML diagram of smart things in the Iceo application

The test of the Bottle with the BTNode has been carried out separately from the actual smart supply chain application, since we do not have enough BTNodes. Instead of measuring the temperature, we measured the brightness, since we also have an LED as an actuator that has an impact on the current brightness. First we have to extend our class from the virtual object class and to override two methods: the init-method uses

the reference to its sensors, which is able to support the sensor protocol we defined in Chapter 4 to register itself for brightness events. The notify-method checks for brightness events and if the brightness is too low, it uses the actuator reference and the actuator protocol to switch an LED on.

Experiences One missing component is the data storage at the producer site, which is able to save the state data of a representation. As long as a representation migrates from one base to another base, the state and the code is transparently migrated by RMI. Although we have a storage provider at each base, a representation can save its state to this storage provider when it gets to the grace period, but this storage provider is only locally accessible. The implementation of the location specific warehouse management systems as services that are loaded by a base is conceptually better than having smart locations, since a location might have to run several location-dependent services instead of implementing the functionality into one smart location. From a performance perspective, we think it is better that a representation is modeled as a simple Java object without thread, since locations that have to handle several thousand smart things might not have the computing power to handle several thousand active Jini services or even worse, web services. Another positive aspect is the simple extension of the provided abstract VirtualObject class for smart thing specific behavior, especially the possibility to query sensors and to control actuators. In a final implementation, one has to merge the advantages of the Wsst framework, i.e. location managers and composition, with the advantages of the Iceo framework, i.e. migration and the modeling of representations as simple Java objects.

6.2 Quantitative evaluation

Besides the qualitative evaluation of the systems and concepts, we also wanted to evaluate the quantitative aspects in order to check whether our systems can be deployed in real world scenarios with thousands of smart things at an arbitrary location. As the Web Services approach showed, the performance can be a problem, but in general, we had no performance problems in the smart supply chain application, since the only limiting factor was the detection speed of the tag detection systems, which took most of the processing time.

In order to compare our different smart thing systems, we do not measure the actual smart thing systems, but the service discovery platforms they rely on, since these results are easier to compare. Due to the fact that they also provide the basic building blocks of our systems, we can use the qualitative analysis of these basic building blocks to make inferences about the behavior of complex patterns consisting of these basic building blocks.

The subject of our quantitative analysis is all aspects of a service discovery platform that have an impact on the performance of our systems and the applications that make use of them, including the memory usage of the runtime environment as well as the services, time and network traffic needed to register a service, the time and network traffic to lookup a service, the time and network traffic to invoke a remote method and finally, the overall time and network traffic for a small test application. [100] provides a more detailed description of the analysis.

In the following, we describe the test application, the test environment, the measurement of the relevant values and finally present the test results.

6.2.1 Test application scenario

In the test application scenario, we have smart milk bottles whose things are tagged with an arbitrary tagging technology. The representations of the smart milk bottles all run on the same server, e.g. a server of their producer, and the tagging technology allows recognition of whether a milk bottle has been put into the shelf or has been taken out of it. A shelf notifies the representations of the milk bottles, which are capable of processing such events, about the events. In the test application scenario, the distribution of the milk bottles is realized by one national and two regional distribution centers as well as four stores. We have a shelf in every distribution center and in every store, which is illustrated in Figure 6.15.

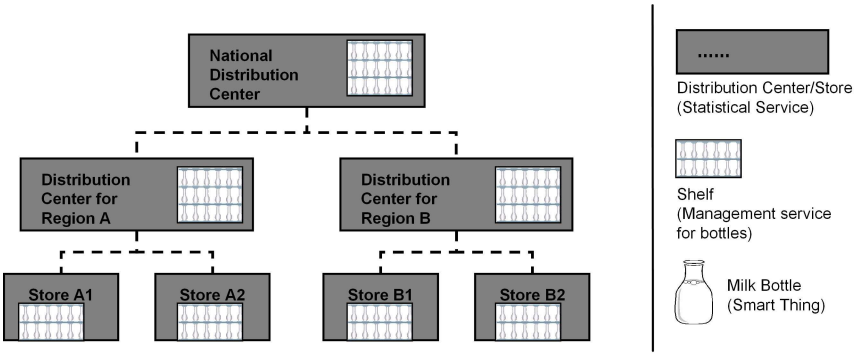

Figure 6.15: Summary of the client application

Besides the representation, the shelves and the distribution centers or the stores are also realized as services. It is possible to ask the shelves for the number of bottles and a list of all bottles they host. The distribution centers provide a service that generates a statistical analysis of all bottles in circulation, e.g. one method returns the number of bottles in the distribution center and all of its sub distribution centers, or the average storage period of the milk bottles on every distribution level can be determined by means of the data each bottle has recorded before. Effectively, we simulate the scenario only with the means of services that are, on one hand, web services and, on the other hand, Jini services.

6.2.2 Test environment

As a test environment we used three Windows computers that are networked by a 100 Mbps Ethernet. The directory service ran on one computer, on a second computer we ran all the services of the test application, i.e. smart things, shelves and statistics, and the third computer was used for the requests to the directory service and the remote method invocations of the client application. A fourth computer, which was also connected to the network, carried out the analysis of the network traffic.

Computer/network All computers were PCs using the Windows XP operating system, they had a 451 MHz Intel Pentium III processor and 256 MB RAM. The local

network was a 100 Mbps Ethernet and all the computers were equipped with a 100 Mbps network interface card. During the execution of the tests, we made sure that no other applications had an impact on the network traffic. For the measurement of the network traffic we used Finisar Surveyor, which allows for such measurements.

Jini We used the Java 2 SDK, Standard Edition, Version 1.4.0 as well as Jini Technology, Version 1.2.1.

Web Services Web Services are a platform and programming language independent standard. There are several frameworks that provide a runtime and development environment for Web Services. For our tests, we used the Microsoft .NET Framework as well as the so called Java Web Services. We used Apache SOAP as runtime environment and IBM's Web Services Toolkit as the development environment for Java Web Services. We deployed SoapUDDI as a private UDDI server that contacts a Microsoft Access database for its directory entries and queries.

6.2.3 Measurement of the relevant values

We divided the measurement into five categories. First, we measured the memory usage of the runtime environment as well as for the services. Then, we conducted performance measurements of the three phases: service registration, service lookup, and service invocation. Finally, we analyzed the performance of the described test application. Before we present the actual results, we first describe how each value is measured.

Measurement procedure In the case of the runtime environments, the measurement of the memory usage was carried out with the Windows Task Manager. The memory usage of the services could be realized through methods provided by the runtime environment for that purpose.

The measurement of the time was always carried out in the client application: the current system time was taken directly before and after a service invocation. Thus, the answer time is the difference between the two measurements, including the time for marshaling and network delays besides the actual time for the service invocation. We conducted every time measurement twenty times and calculated the average as well as the standard deviation for every test.

The measurement of the network traffic was carried out with the above-mentioned network analysis software, which measured the total network traffic between client and service, including the IP and TCP headers.

Memory usage The memory usage is intended to give information about how many resources are used by the infrastructure in which the services can be deployed. For Jini and Java Web Services, we measured the memory usage of the JVM in which we start the services. For the .NET Web Services, we measured the corresponding memory usage of the .NET runtime environment.

The memory usage of the services is determined in the following way: from time to time we started services of the same type and measured the average increase of the memory usage within the runtime environment. This gives a hint of how many resources are necessary per service and how many services can be deployed on one host. Since

the memory usage of services heavily depends on their complexity, we developed services with a minimum functionality in order to allow them to participate in each platform.

Service registration First, this performance test has to show the time needed and the network traffic generated by a service invocation. In our case, we compared Sun's lookup service implementation and the SoapUDDI implementation. To be able to compare the performance, we did not include the time needed and the network traffic generated to perform the discovery process in Jini.

Lookup of a service We analyzed how long it takes to find a service, when we use on the one hand the unique service ID and, on the other hand, the service name. Additionally, we measured the network traffic that occurs in this phase. Again we did not consider the discovery process in Jini.

Another important criterion is the scalability of both platforms. To determine this, we measured the response times for the lookup of a service based on the ID and based on the name after more and more other services have been registered at the directory service.

Service invocation This issue shows the performance of remote method invocations in Jini with RMI and Web Services with SOAP. For this purpose, we call methods of the representation: one method returns a simple integer value and another method returns a complex array of objects, i.e. the list of locations where a smart milk bottle already has been. In the latter case, we vary the number of objects. In all cases, we measure the time and the network traffic.

Test application This last test analyzes a typical client application that includes the lookup of a service and the following service invocation. Due to the complexity of the application, we only implemented this test with Jini and .NET Web Services and left out an application with Java Web Services.

In a first test, we wanted to find the statistical service of a store; in addition, its name should be returned. In this test, we determined the time and the network traffic and included the discovery phase in Jini in the test results. In a second test, our client application has to find the statistical service of the national distribution center and to ask for the total amount of bottles on all distribution levels. For this purpose, every statistical service has to contact the both statistical services on the next level and to query its own shelf in order to determine the number of bottles. These requests take place concurrently and the results can be added up after all responses have been received. In total, a statistical service and a shelf service each need to be looked up seven times; in addition, the method that returns the number of bottles has to be called accordingly. In this last test, we measure the response time for the client invocation and the total network traffic.

6.2.4 Results

For each of the five test categories, we present the results in a table and as a figure. Additionally, we give a short interpretation of the results.

Memory usage Table 6.1 and Figure 6.16 show the results of the test concerning the memory usage. Although the memory usage of the Jini runtime environment is approximately half of the memory usage of the .NET Web Services runtime environment, both are high, since they are not optimized for optimal memory usage, so that an optimization could reduce the memory usage in all cases. The memory usage of a single service in both Java environments is much lower than a service in the .NET environment, since in Java, the instantiation of another Java object is cheap in comparison with .NET Web Services, where every web service possesses a broad spectrum of functions that is inherited from its class hierarchy.

Measurement	Jini	Java Web Services	.NET Web Services
Runtime environment (KByte)	9564	22824	17332
Service (KByte)	1.84	38.42	1640.97

Table 6.1: Memory usage

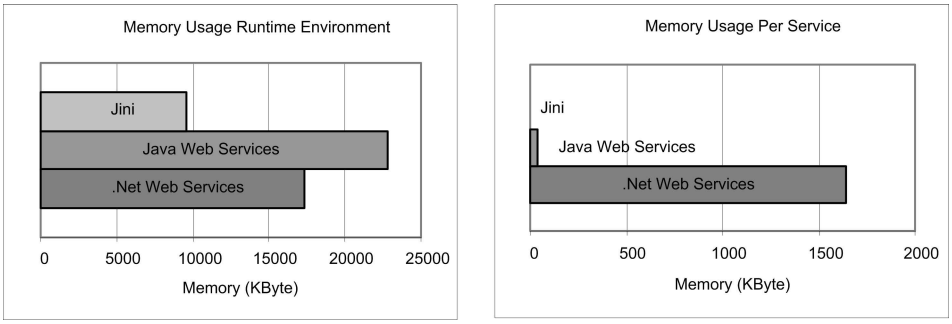

Figure 6.16: Memory usage

Service registration Table 6.2 and Figure 6.17 show the results of the test concerning the service registration. The service registration in Jini, where the discovery process is not included, is faster in comparison with UDDI. The longer registration time in UDDI results from its three SOAP requests: one to register the service, another to register the so-called tModel and one to register the binding template. In Jini, on the other hand, one request carries out the whole registration, including the registration of the proxy object and the attributes. The network traffic is all about the same, but one has to consider the continuous traffic of renewing the leases in Jini, which occurs after registration for the whole lifetime of a service. Our tests showed that the time for the registration is independent of how many other services have already been registered at the directory service. We registered up to 20,000 services and noticed no substantial changes to the values in the table.

Lookup of a service As Table 6.3 and Figure 6.18 shows, the network traffic for a service lookup by means of the identifier is approximately the same, but the response time with the Jini LUS is much faster than the response time of SoapUDDI. The poor performance of the latter is due to two circumstances: first, the processing of the SOAP

Measurement	Jini LUS	SoapUDDI
Response time (ms)	137.6 ± 18.9	421.1 ± 82.7
Network traffic (Byte)	4067	3446

Table 6.2: Service registration

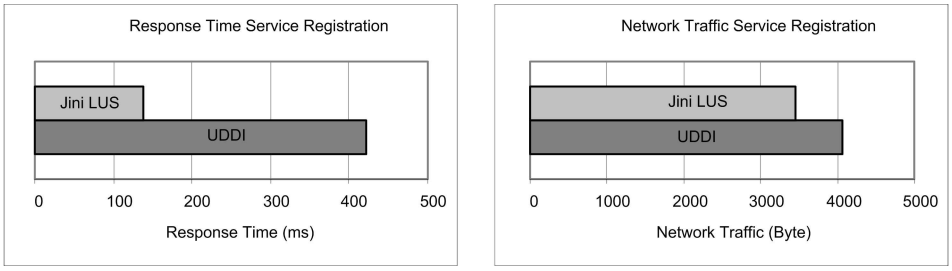

Figure 6.17: Service registration

protocol is more complex than RMI and the SoapUDDI implementation has to contact a database, while the Jini LUS holds its information in its own memory's address space.

Measurement	Jini LUS	SoapUDDI
Response time (ms)	9.0 ± 3.3	284.4 ± 21.6
Network traffic (Byte)	2040	2217

Table 6.3: Service lookup by means of a service ID

Table 6.4 and Figure 6.19 show that both numbers for the service lookup based on the service name are only a little bit higher in Jini, but both numbers for SoapUDDI are approximately doubled. The reason for the latter consists in the two phases of the SoapUDDI lookup process: first, the service name is used to lookup the UUID and then the UUID is used to lookup the service URL.

Figure 6.20 shows that Jini's service lookup scales up to 20,000 services. If a service lookup based on the service identifier is used it constantly takes 10 ms, and if the service lookup based on the service name is used, the response time increases linearly but very slowly up to 20 ms. In the case with SoapUUDI, the time to look up a service based on the service ID also stays constant, but at a much higher time - around 300 ms. The service lookup based on a service name scales very poorly: it is a linear relation that takes about 1,200 seconds when 20,000 services are registered. The problem is that the corresponding field in the database in not indexed. In general, a better implementation of UDDI would lead us to expect better performance.

Service invocation As Table 6.5 and Figure 6.21 show, a service invocation in Jini with RMI is much faster than a service invocation with the Web Services counterparts. Although the .NET Web Services implementation is around 4 times faster than the Java Web Services implementation, the Jini implementation is still around 10 times faster than the .NET Web Services implementation. The network traffic caused by RMI is approximately a fourth of the traffic generated by the SOAP counterparts, since SOAP relies on ASCII/XML documents that are much larger than the compact serialization

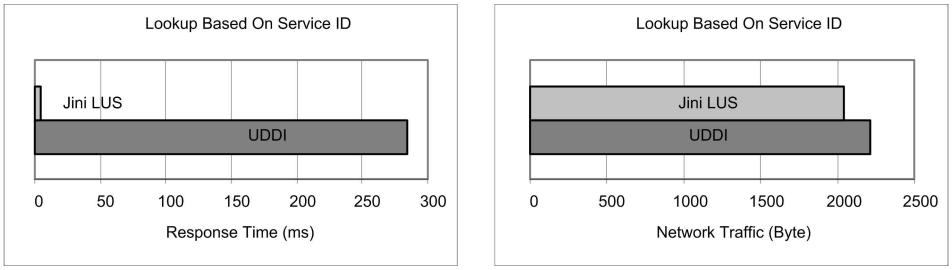

Figure 6.18: Service lookup by means of a service ID

Measurement	Jini LUS	SoapUDDI
Response time (ms)	11.2 ± 3.6	546.3 ± 129.0
Network traffic (Byte)	2269	4220

Table 6.4: Service lookup by means of a service name

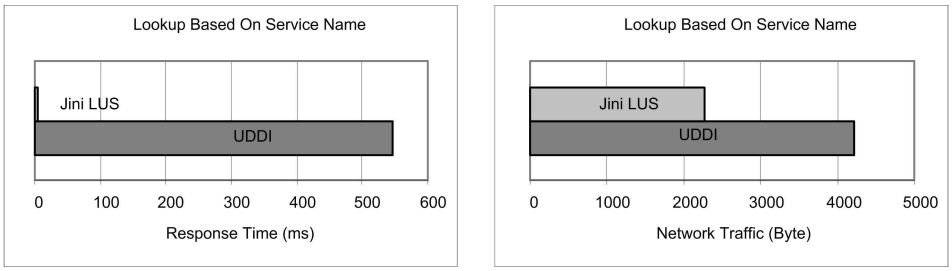

Figure 6.19: Service lookup by means of a service name

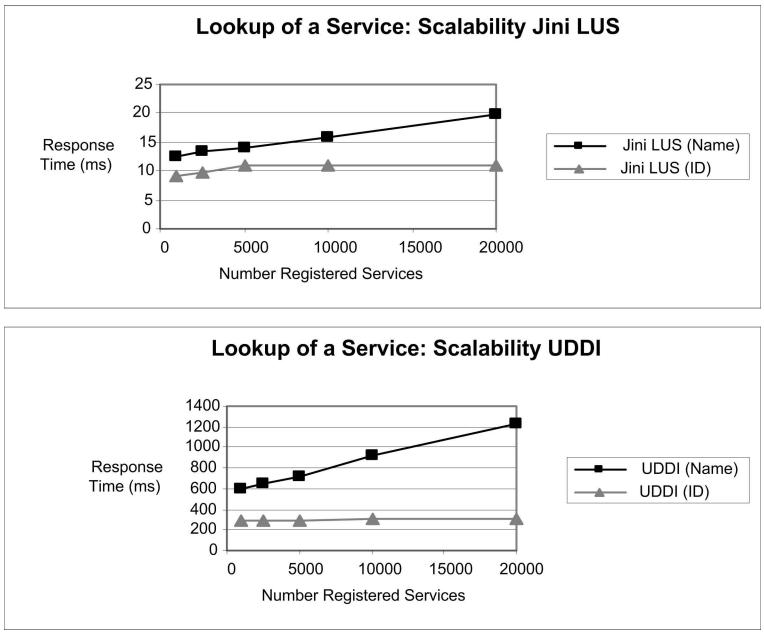

Figure 6.20: Scalability of the directory services

stream generated by RMI.

Measurement	Jini	Java Web Services	.NET Web Services
Response time (ms)	14.4 ± 1.7	596.0 ± 20.3	159.1 ± 5.4
Network traffic (Byte)	575	2044	1894

Table 6.5: Service invocation

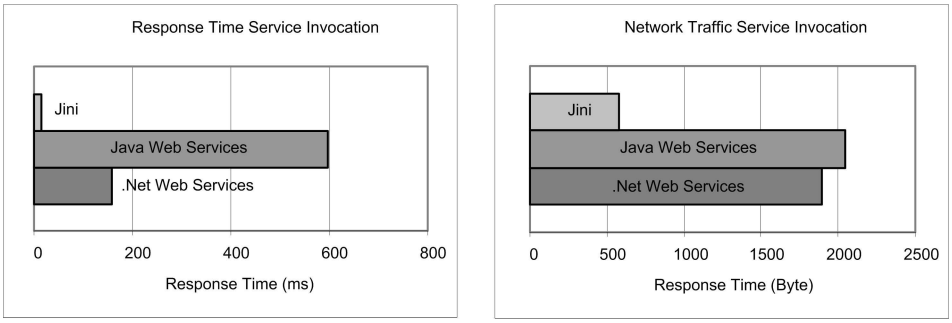

Figure 6.21: Service invocation

Again, Jini is superior to both Web Services approaches regarding the scalability of return values, as Figure 6.22 shows. The response time in Jini stays almost constant, but both Web Services implementations increase linearly but relatively steeply. The same is true for the network traffic: the increase is linear at every solution, but Jini's increase is much flatter than the increases at the two Web Services implementations. This mainly results from the complex marshaling scheme used in SOAP, resulting in large XML documents which need to be processed at length.

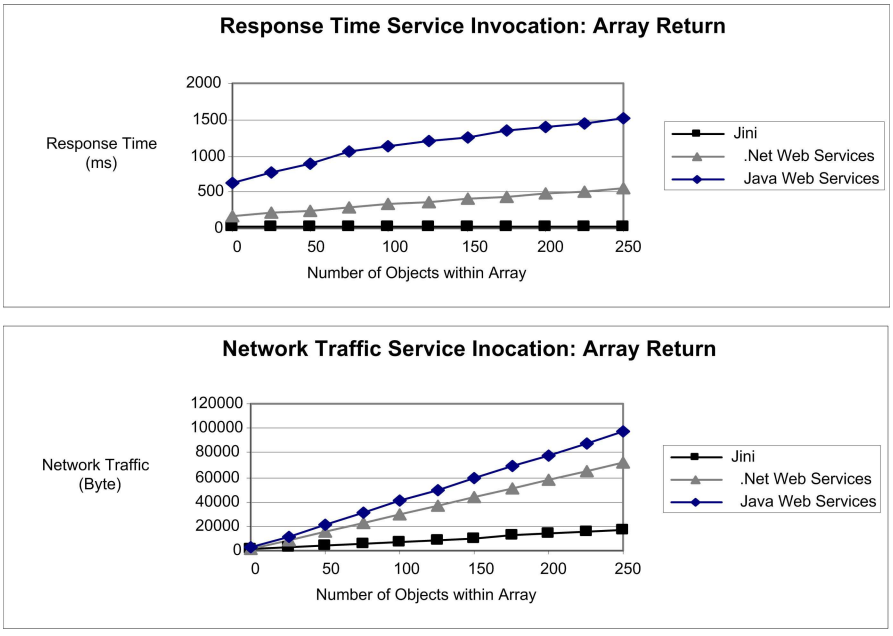

Figure 6.22: Scalability of service invocations concerning array size

Test application Table 6.6 and Figure 6.23 show that Sun's Jini implementation is more efficient than the .NET Web Services implementation, especially concerning the response time, as the previous tests led one to expect. The discovery process is included in the numbers, except for the code download of the LUS proxy, which occurs only once. It normally takes 1300 ms and generates 70 KBytes of network traffic, so if we included this in the simple method invocation, the .NET Web Services approach would be better.

Measurement	Jini LUS	.NET Web Services
Response time (ms)	198.8 ± 7.2	814.8 ± 121.8
Network traffic (Byte)	5087	9325

Table 6.6: Simple invocation of the client application

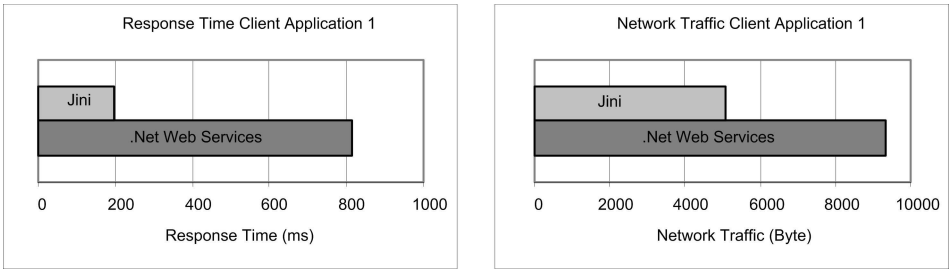

Figure 6.23: Simple invocation of the client application

Table 6.7 and Figure 6.24 analogously show the same results for the complex invocation of the client application: Jini is around four times faster than .NET Web Services. If we included the LUS download here, Jini would anyhow be more efficient than the .NET Web Services implementation.

Measurement	Jini LUS	.NET Web Services
Response time (ms)	857.5 ± 32.8	4935.6 ± 260.8
Network traffic (Byte)	49756	66876

Table 6.7: Complex invocation of the client application

6.3 Conclusions

The tests confirm the concerns that we already had at the stage of qualitative evaluation of the systems: method invocation, and especially service lookup, is more efficient under Jini than under one of the two Web Services frameworks that make use of SoapUddi implementation. Although another UDDI implementation might be much more efficient than the current SoapUDDI implementation, one cannot expect a SOAP implementation to get the same performance results as RMI, since SOAP relies on many more protocol layers, which require more resources for processing than the much simpler RMI.

Although the Jini data states that a representation can be realized as a Jini service, we think that this still involves too much effort for very simple smart objects that might

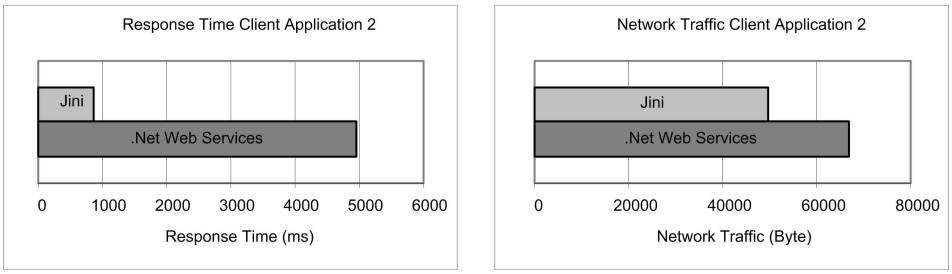

Figure 6.24: Complex invocation of the client application

not even have a state. A simple Java object also allows for migration by RMI, since simple Java objects are transparently serialized and deserialized by RMI. Jini services as remote objects, on the other hand, cannot be serialized, since RMI uses a corresponding stub object in such a case.

From a qualitative and quantitative view, we recommend modelling representations as simple Java objects and the middleware services as Jini services. Jini services that run on the same machine should be started within the same JVM, so that RMI is not necessary for interactions between services on the same machine. Since we expect that most communication will take place locally, i.e. communication between representations among themselves and with other applications, in these cases the whole communication could be managed locally. Besides the performance aspects, the concepts Jini provides, such as discovery, leases and remote events, are all needed by an implementation of the model.

The concepts we proposed in our model have all be proven to be helpful to structure, implement and deploy smart thing applications. Since we did not implement all the concepts into one smart thing system, a final system can be easily composed of the three systems we introduced. It is easy to integrate, since we already have efficient implementations of all concepts and the different implementations are not interdependent, so that a final system mainly has to bundle the different implementations into one.

Chapter 7

Related Work

The aim of this chapter is to show the overlap of the work presented in this dissertation with the related work in the domain of smart thing systems and ubiquitous computing in general, as well as adjacent domains, for two purposes: first, this comparison allows the original contribution of this work to be emphasized and second, the related work offers the chance to focus on some of the concepts that are outside the main scope of this dissertation. In Chapter 3, we already introduced some other work with the intention to show how we build on these technologies and standards in order to be independent from them in our model. The related work presented in this chapter, on the other hand, partly overlaps with the concepts and the smart thing systems we presented. The following remarks do not provide a general introduction to the related work, but concentrate on the common aspects it shares with the work we have carried out. For a more comprehensive overview of the systems, we refer readers to the referenced literature.

We classify the related work into three classes: first, the related work of adjacent domains that are not primarily part of ubiquitous computing research, but provide concepts that are useful for our model. Second, the related work of smart thing infrastructures that share our goal to support smart thing applications. Third, we consider some middleware infrastructures for ubiquitous computing applications in general.

7.1 Adjacent domains

Adjacent domains refer to research areas that do not focus primarily on ubiquitous computing issues, but these ubiquitous computing issues have to be considered in order to model smart things.

7.1.1 Naming and addressing

We introduced an identifier as consisting of a unique name and a home address in order to uniquely identify a smart thing. The name part only has to be unique within its namespace, i.e. the home address. Further, we defined an address as the name of the location where an entity resides. Thus, the home address is the name of the location in the virtual world where the home service is located.

Hauzeur [46] generally considers the relation between names, addresses, and routing in computer communication. A name is a linguistic object which uniquely identifies an entity within a certain domain. A route is a list of names that denotes the path from the source entity to the sink entity via which a message has to be transmitted. An address

is a concept between name and route that is used to efficiently find a route. It can be regarded as the name of the communication object that an entity uses to transmit its message in the direction of the sink. Such addresses are normally numerical to enable them to be processed efficiently, and a mapping between the entity and its communication object is needed. IP addresses, for example, are numerical and the DNS system maps a domain name of a host into an IP address. Hrachovec [54] and Su [110] also discuss these issues.

Berners-Lee et al [12] introduce a Uniform Resource Identifier (URI) as a string of characters to uniquely identify an abstract or physical resource. A URI can either be a Uniform Resource Name (URN), a Uniform Resource Locator (URL), or both. While a URL primarily identifies a resource via a representation of its primary access mechanism, e.g. their network "location", a URN refers to URIs that remain globally unique even when the resources cease to exist or become unavailable.

With our definition of a location in both worlds, i.e. real world and virtual world, we are a bit more general and subsume the other definitions. Although an identifier consisting of a name and a home address is similar to a URI that consists of a URN and URL, we do not demand that the name must be globally unique, but only within its namespace. In our case, this is not a restriction, since the home address as namespace provides the uniqueness of the whole identifier. One could argue that it might be a drawback that the home address cannot be changed, but we cannot require that, because some technologies, such as printed bar codes or read-only RFID tags, do not the support the modification of an identifier.

7.1.2 Location models

We presented a hybrid location model consisting of four parts that are connected through fourteen translation functions. Two of the four parts refer to the world location model that models static locations in the real world and the two other parts refer to the smart thing location model that models the location within mobile smart things. Both parts of each model refer to a symbolic location model as well as a physical position model, since both kinds of models provide different advantages.

Leonhardt [75] proposes a hybrid location model to support location-based services in mobile computing. This thesis considers all relevant aspects concerning location modelling: these comprise location models, service models, acquisition of location data, uncertainty, predication and interpolation, and security considerations. There are two functions in the combined model that connects a symbolic model with a geometric model: the area function maps every symbolic location to a geometric area and the leastMatchingArea function maps an area to all the best matching symbolic locations.

Jiang et al [59] also propose a hybrid location model that allows for a tree of locations that have a symbolic name and positions. Since every node in the tree can have its own coordinate system for its positions, they require a function enabling every edge to convert the position of one coordinate system into the other. The recursive application of that scheme allows for conversion between two arbitrary nodes in the location tree.

Hightower et al [51] classify localization technologies according to several criteria: technique, physical vs. symbolic, absolute vs. relative, localized location computation vs. recognition, accuracy and precision, scale, cost, and limitations. Besides this classification, the authors also consider several other aspects in localization, e.g. localization techniques [51, 52, 53].

Many other projects also make use of symbolic or hybrid location models. The innovation of our model is the clear distinction between a hybrid location model for the static world and one for mobile smart things. Another aspect is the combination of the two criteria: symbolic locations vs. physical positions and localized location computing vs. recognition. Although both criteria have been already mentioned by Hightower et al [51] to classify localization technologies, we make use of both these criteria and explicitly built them into our tag detection model.

7.1.3 Cellular IP

Although Cellular IP [17, 18] refers to routing aspects of IP packets, we mention it, since it adopts the concepts of cellular wireless networks to support mobility of IP-based mobile computers. In our work, we adopt the concepts to support the management of representations for mobile smart things. The reason for the introduction of Cellular IP is to have the advantages of IP routing and the mobility management of cellular. In fact, Mobile IP has been introduced to support mobility in IP routing, but the signaling overhead for fast moving mobile nodes within a particular area is much higher than the one needed by the management of cellular. Cellular IP extends the traditional Mobile IP scheme, which uses a home agent to manage a care-of address with a gateway. In a Cellular IP environment, the care-of-address denotes a so-called gateway that acts as a gateway from the traditional Mobile IP enabled network to a Cellular IP network. The latter consists of base stations that are the wireless access points for mobile hosts. Within this network, the routing always takes place between the gateway and the base stations where the mobile host is detected. If a mobile host changes to another base station within the Cellular IP network, it does not need to update the home agent, since the care-of-address of the gateway stays the same.

We chose a similar approach with the home service and the current address that references the managing hub location manager, which is the root of a location domain of base location managers. Although the concepts seem to be quite similar, our situation is more complicated, since we have to manage smart things that can also act as a hub location manager and we have a location manager lattice, so that the hub location manager is not predetermined.

7.1.4 Artificial intelligence

We explicitly mentioned that smart does not refer to intelligent and therefore does not cover aspects of artificial intelligence (AI) research. Typical issues of AI [93] comprise intelligent agents, solving problems by searching, knowledge representation, reasoning, planning, making decisions under uncertainty, learning and communicating, perceiving and acting. However, some of the aspects have a slight connection to smart things. The definition of an agent in [93] comes close to our definition of a smart thing: agents interact with environments through sensors and effectors. We required a smart thing to have an identity, to know its location, to be able to perceive its environment through sensors, and to be able to manipulate the environment through actuators. Franklin et al [38] discusses a taxonomy for autonomous agents. If we compare our requirements with the requirements of an autonomous agent, the following factors are missing: autonomy concerning its own actions, goal-orientation, i.e. the execution of actions to achieve its own goals, and temporal continuity concerning its execution. A representation might be

programmed to act autonomously and to achieve its own goals, and temporal continuity is given for smart things that make use of implicit coupling. This means that implementation of our smart thing model might be deployed as an infrastructure for autonomous software agents, with the restriction that they have to provide the functionality for a real-world thing. Three further issues are communicating, perceiving and acting. Although AI focuses on a broader spectrum in the three domains, i.e. natural language processing vs. event communication, computer vision vs. brightness sensors and robotics vs. LEDs, the foundations of these concepts remain the same. In general, intelligent vs. smart focuses on a broader spectrum, but in principle our smart things could be extended with AI methods in the future. We only provide the minimum infrastructural support for a thing to become smart; everything else is up to the developer of the representations and applications.

7.2 Smart thing systems

To the best of our knowledge, five other projects or systems also try to support smart thing applications that are mainly commercial products. The main difference to our model is that the other systems only partly consider the identification and localization aspects of smart things, allow only a passive behavior of smart things or do not scale to world-wide usage.

7.2.1 Cooperating Smart Everyday Objects

Cooperating Smart Everyday Objects (CSEO) [102, 103, 104] is another approach of our research group to enable smart things. The main abstraction of CSEO is an active tag, which possesses its own microprocessor, power source, communication, and sensor modules. By adding such active tags to everyday objects, cooperation among the tagged everyday objects and with mobile user devices becomes possible. One key assumption is that these active tags are resource-limited, such that only the cooperation between them allows for powerful applications.

Context-awareness is the basis for smart everyday objects to cooperate. The concept context-awareness refers to the recognition of the current situation of a smart everyday object. In order to support such context-aware applications, CSEO introduces a software platform consisting of a programming language for describing context-aware services, a compiler for this language, and a tuplespace-based infrastructure for distributing data among collaborating entities.

CSEO sees handhelds as cooperation partners for smart objects and identifies six usage patterns for the communication between smart objects and handhelds: mobile infrastructure access point, user interface, remote sensor, mobile storage medium, remote resource provider, and weak user identifier. In order to make use of the computing power of a handheld, a smart object might upload a so called Smoblet, i.e. a piece of code, to the handheld where it can be executed.

The main difference to our approach is that CSEO focuses on the user interface and the direct interaction of active tags only. In our approach, we explicitly do not support user interaction, since we want to support applications with myriads of smart things among themselves where user interaction is no longer possible. In fact, we also allow for implicit coupling, i.e. the whole functionality is placed on the tag itself, but we focus on

a background infrastructure that enables smart things.

7.2.2 Auto-ID Center

The former Auto-ID Center and its successor EPCGlobal[1] attempt to establish standards in the RFID domain in order to enable companies to exchange information about tagged products. The elements of their infrastructure are: the Electronic Product Code (EPC), the Reader Protocol, the Product Markup Language (PML), PML servers, the Object Name Service (ONS), the Savant [85], and standards that define the air protocol between RFID tags and readers. A Savant controls the tag readers with the Reader Protocol, and the tag readers use the air protocols to communicate with the tags in their coverage space, which transmit their stored EPC via the tag reader to the Savant. The Savant uses the EPC to look up the address of a PML server in the ONS system, which it uses to contact the PML server based on PML. Applications might be integrated via Savant or by implementing the Reader Protocol directly.

An EPC can be roughly regarded as the traditional bar code on products, i.e. an EAN or UCC bar code, extended with a serial number. For additional information on a product, the ONS service has to be contacted: this maps the EPC to a URL whose DNS part needs to be mapped to an IP address of a PML server, which stores the additional information about a product. In our model, we directly store the DNS address of the home service on the tag in order to prevent having another name service besides the DNS. Although an EPC is shorter than our identifier and an additional indirection layer provides more flexibility, we assume that the size of a tag memory is not a bottleneck, since 64 Bytes or less are in most cases sufficient to store our identifier. The additional mapping is not needed in our model, so that we cannot justify setting up a world-wide ONS system that maps EPCs to URLs.

Since our smart thing systems work independently of the underlying tag detection hardware, we do not consider aspects such as the air protocol or the Reader Protocol. In general, it is not intended to support active smart things, but passive data about detected tagged things.

The Savant is the central element of the infrastructure, but at the time of writing, the whole specification of the Savant system is changing, so that we can only describe the version 0.1 that we have also tested. At the moment, it consists of three independent subsystems: the event management system (EMS), the task management system (TMS), and a real-time in-memory database (RIED). Savants should be arranged in a tree concerning organizational aspects of a company in order to be able to aggregate the data. Leaf Savants control RFID readers and internal savants aggregate data. Since the data exchange between Savants is not specified, we can only describe the Leaf Savants. The EMS provide interfaces for adapters that communicate with the reader, queues that decouple producers and consumers of events, filters that disregard events, and loggers that finally store or process the events. Implementations of these interfaces need to be specified in a startup script of the EMS, which plugs all the modules together. A logger might use the RIED to store the tag reads in the memory. User tasks that can be registered at the TMS are scheduled like a cron job. They access the RIED in order to perform their task. After testing the Savant, we came to the conclusion that the concepts of the EMS fit well, in contrast to the RIED and the TMS. In our opinion, RIED and

[1] www.epcglobalinc.org

TMS do not provide the means to model smart thing based applications. It is easier to write location-based services that register for entry and exit events, as proposed by our model.

In general, it is not possible to evaluate Savant conclusively, since most of its components are changing. Additionally, it is difficult to compare Savant with our smart thing systems, since it is intended to manage the data of smart things and does not provide a representation.

7.2.3 Smart Items Infrastructure

The Smart Items Infrastructure (SII) [69] is SAP's approach to a middleware for smart things, in order to couple them with its enterprise resource planning (ERP) systems. An official announcement[2] concerning the adoption of SII was published at the time of writing. For SII as a commercial product, the amount of information is limited, so that we are only able to roughly describe the architecture, which consists of communication services and four layers that use them: a hardware abstraction and a third party adaption layer, an information processing layer, a smart item manager layer, and a business adaptation layer. To enable communication between the modules of the four layers that are normally distributed, the communication services comprise content-based messaging, publish/subscribe communication, and point-to-point communication. The hardware abstraction layer is used to disengage from the identification technology and the third party adaptation is used to control other applications such as production control systems. The next layer, the information processing layer, mainly has to filter out events from the hardware layer that are not required by the other layers, such as spurious exit/entry event combinations. At one level above, the smart item manager provides the main functionality in four separate modules: the system management module, used to manage the entire SII, a data management module, which decides on what data to store in the data storage module, and finally an event handling module, which filters or aggregates events based on rules in predicate logic or if/then-statements. At the top is the business adaptation layer, which builds the bridge to ERP systems such as SAP R/3.

Since the focus of the SII is similar to that of the infrastructure of the Auto-ID Center, in terms of the passive modeling of a smart thing, it is difficult to compare it with our smart thing systems, where every smart thing has its own representation, which is able to control the sensors and actuators on the smart thing. Both systems are fairly complementary to our systems, in that they both support smart thing applications.

7.2.4 VisuM

VisuM, short for Visualisation und MapMatching, is an operative middleware system that supports smart things applications in the Volkswagen production process. As a proprietary development of Volkswagen, there is no public documentation available, so that this description relies on slides of a talk, given by the developer[3] about this system at one of the M-Lab events[4].

The overall goal of VisuM is to identify and localize objects in the real world in order to provide this information to applications. VisuM allows for the support of different Auto-

[2] http://www.sap.com/company/press/press.asp?pressID=2609

[3] Christoph Pelich, Volkswagen AG, christoph.pelich@volkswagen.de

[4] http://www.m-lab.ch/events/ws1-2003.html

ID technologies by uncoupling from them through the VisuM protocol. A message from the VisuM protocol is sent from a tag detection system to a predefined VisuM server, which stores the content of the message in a relational database. Another server, the VisuM App-Server, provides access to the information in the database. A visualization tool, for example, queries the database through the VisuM App-Server and shows on a scalable map how many products of a certain type are located at the locations that the map covers. A message of the VisuM protocol contains the identifier of the tag, the location as either a GPS coordinate or symbolic location name, consisting of four parts describing a place within the Volkswagen location hierarchy, a timestamp of the detection, and an application ID that maps the tag to a specific application. These items of information and the association of the tag with its thing are stored in the database, so that applications can make use of them.

The system and the application work properly and reliably. As with the previous two systems, it is difficult to compare them with our smart thing systems, since these systems manage the passive data of tagged things and do not provide a representation, so that VisuM can also be regarded as complementary to our approach.

7.2.5 RAUM

The RAUM (German for room) system [10, 11, 55] of the Karlsruhe University of Technology aims to enable communication between smart artefacts that are collocated. In order to enable this communication, it defines a location and a communication model. The former defines a root tree with four levels of symbolic names as wells as the physical position of an artefact within the location that is denoted by the symbolic location. Artefacts that are resource limited only have to store their own position as a path of the location tree in order to save memory. Every artefact spans a co-called RAUM, i.e. the space in which it communicates with other artefacts. These spaces should be covered by the location tree in order to support communication within a RAUM. A smart artefact can span one of the three RAUMs: a Listener RAUM enables a smart artefact to receive messages from other smart artefacts in the RAUM, a Speaker RAUM enables a smart artefact to send messages to other smart artefacts in the RAUM, and a Discussion RAUM can be regarded as combination of the two other ones. Besides its location path, a smart artefact has to manage two sets: one set contains all the RAUMs that are defined by it and another set contains all the RAUMs which contain the smart artefact, in order to decide on the reception of messages. Several operators are defined to manage the RAUM. The open operator, for example, is used to announce the definition of a new RAUM. An artefact that is not able to determine its own position uses the location stuffing of the infrastructure, i.e. the smart artefact leaves the location field in a message out in order to allow the infrastructure to fill out this field. Route bridges and routers are the infrastructural services that manage the communication between smart artefacts that use IR or RF as a communication medium.

The RAUM system comes closer to our smart thing systems than the three other systems described above, since it allows for active representations that can communicate among themselves. The main difference to our smart thing systems is that it only allows for implicit coupling, i.e. the representation has to be integrated or attached to the artefact. In our smart thing systems, we provide an infrastructure that allows for the execution of the representation on a hosting service in the background infrastructure.

7.3 Ubiquitous computing systems

Besides the five smart thing systems, we considered eleven other ubiquitous computing systems in terms of their middleware support, in order to check whether we can adapt some of their concepts for our model. This was part of a general analysis about middleware [2, 19, 23, 24, 78, 111] for ubiquitous computing [1, 32, 47, 66]. We briefly sketch the goal of each system and provide a summary of the functions they support. Not all projects have the same relevance for our model, so that we describe five projects in a little bit more detail that target similar issues to those involved in our smart thing systems.

7.3.1 Overview

This overview simply states the name of the project, the bibliography references, the organization that carries out the project and a short description of the project's goals:

- *Aura* [59, 105, 112, 117], Carnegie Mellon University: Aura's goal is to minimize the distraction of a computer for a human user. The main abstractions are user tasks that can be transparently executed on heterogeneous hardware and software.

- *Cooltown*, [9, 25, 63, 64, 65, 67, 88], HP Invent: Cooltown aims to support nomadic users with electronic devices that move from one location to another. For this purpose, web technologies need to be extended in order to allow for pervasive computing applications.

- *EasyLiving*, [14, 15, 79], Microsoft Research: This project aims to support users in so-called intelligent environments. Users have to be recognized and tracked in order to allow the system to select appropriate input and output devices.

- *GaiaOS*, [48, 49, 90, 91], University of Illinois at Urbana-Champaign: Concepts of operating systems are transferred by this project to the ubiquitous computing domain in order to support ubiquitous computing applications.

- *Hive*, [80, 89], Massachusetts Institute of Technology: Originally, Hive is a system which provides an infrastructure for mobile software agents that can also be deployed to support ubiquitous computing scenarios.

- *iWork*, [60, 61, 87], Stanford University: This project provides the necessary infrastructural means in order to support user interaction with various hardware and software components.

- *Nexus*, [42, 74, 83, 84], University of Stuttgart: As an extension of a geographical information system, the Nexus system models static and dynamic location information of objects and their environment in order to provide location-based services.

- *one.world*, [43, 44, 45], University of Washington: With three principles to make changes in the environment explicit, dynamic composition of applications, and the separation of code and data, one.world provides a system framework for ubiquitous computing.

- *ParcTab*, [119, 120, 121, 122, 123], Xerox PARC: The oldest of all the projects mentioned here, which has been initialized by the ubiquitous computing pioneer Marc Weiser, considers an infrastructure for pager-sized computers, which can be localized at a room level.

- *Stitch*, [6], Xerox Research Centre Europe: Stitch is a middleware that is intended to support ubiquitous computing applications with the two concepts event handling and distributed tuple spaces.

- *Sylph*, [22], University of California at Los Angeles: Sylph provides an abstraction layer in order to provide sensor values with an arbitrary service platform such as Jini. Sylph explicitly considers single sensors and not sensor networks: the latter are also subject to current research [4].

7.3.2 Common functions

Besides the analysis of each individual system, we were also interested in factorizing the tasks that are commonly supported by these middleware projects, especially in comparison with traditional middleware systems. In the following, we give a short summary of our results:

Provision of environment information: the main goal of the monitoring of the environment is to recognize human users in order to enable applications to use this information to personalize their services. Some projects support sensors to retrieve general data concerning environmental phenomena.

Support of implicit and explicit user interaction: these projects provide a rich selection of new interaction schemes. Besides the sensor technology, which is mainly responsible for user interaction in the background, some new input devices and techniques are supported.

Support of migration of software components: mobility of software components can be understood in two ways: on the one hand, software components can spontaneously appear and disappear. On the other hand, they can migrate from one host to another host.

Support of location awareness and mobility of physical objects: a physical object can either be a mobile electronic device or a smart thing. Since hardware components normally have a software proxy, their appearance and disappearance can be handled the same way as with the software components. Besides this binary information about whether an object has been detected, different location models, technologies, and systems are used. If additional information is required that is not available on the object, the information needs to be looked up. Sometimes objects cannot sign off, so that timeouts, leases, or heartbeat messages are necessary.

Modeling: every project needs to define what parts of the world it wants to model. Projects that do not explicitly consider this aspect have the semantics implicitly encoded in their methods and protocols. Other projects provide XML languages such as the Resource Definition Language (RDF) or a class hierarchy in order to dynamically describe relevant parts of the world.

Communication mechanisms: most projects make use of the means provided by the operating system and other middleware systems, such as CORBA. For performance reasons and for appropriate modeling of their domain, some projects define their own communication mechanisms and protocols.

Synchronous vs. asynchronous communication: most projects provide both kinds of communication: for synchronous communication they mainly rely on traditional middleware systems. Asynchronous communication is mainly realized through publish/subscribe systems.

Transparency: traditional middleware systems tried to abstract from the distribution of the entire system. This has changed in middleware for ubiquitous computing systems that explicitly consider the distribution and the locations of the system's entities. The main abstraction now rather refers to transparent access to heterogeneous hardware and software.

Decentralized vs. centralized architecture: the decision whether the architecture should be laid out in either a centralized or decentralized way mainly depends on the availability of a background infrastructure. If such a background infrastructure is available, the systems use a centralized design, since the background infrastructure is able to provide centralized services. If no background infrastructure is available, the design must be decentralized, since no central services are possible.

Realization of the data storage: normally, the realization of the data storage should be transparent to the middleware, but some projects explicitly focus on tuple spaces [20]. Besides this option, regular database management systems are used to store and to retrieve data.

Support of the developer: in general, the projects provide three different means to support the developer: the provision of new software patterns, such as the extension of the model-view-controller, scripting languages, and software frameworks with a generic class hierarchy for a certain application domain.

Integration with existing systems: to support integration with existing applications, the projects provide three different approaches: first, the systems do not consider this aspect and it is up to the developer to integrate existing software. Second, the systems provide integration with the existing software of a certain domain, as in meeting rooms, where only web browsers and presentation platforms are necessary. Third, the projects provide a general abstraction to integrate existing systems.

7.3.3 Nexus

Nexus [42, 74, 83, 84] provides a middleware for mobile and spatially aware applications. Its main abstraction is the augmented world model that, on the one hand, models the real world and, on the other hand, augments this model with virtual information. The architecture consists of three layers: the top layer consists of the actual application that uses the application interface to access the next layer, the federation layer. This layer comprises nodes that use the Area Service Register (ASR) in order to access the next layer, the service layer, via the service interface. Such a node splits a request into several sub-requests and after the reception of the sub-results, it aggregates the response and

returns it to the application. The service layer consists of the Spatial Model Server (SMS), which stores the actual information about static objects. For mobile objects, on the other hand, a location service is used. One key idea of this system is that everyone can set up its own SMS, which has to be compliant with the Nexus interfaces in order to extend the Nexus system; for example, everyone could set up a web server in order to extend the WWW. Nexus also provides a generic and extensible class hierarchy in order to model the static and mobile objects of the real world.

An important feature, in comparison with our model, is the use of a location service. Location services have to track objects using positioning systems or tracking systems. The location services are distributed and communicate among themselves via wireless or wired communication networks. A sighting of an object consists of the object's identifier, the WGS 84 coordinates, an area of coordinates that specifies the inaccuracy of the localization system, a timestamp and information on whether the data comes from a positioning system or a tracking system. An object references an object register that knows the location service where the object has signed in. Each location service has a location register that stores all objects at a certain location. Building on these components, a location service can answer queries concerning the neighborhood relation and it also provides event notifications, for example, it provides entry and exit events.

The static modeling of the world with the SMS can be regarded as complementary to our approach, since we only focus on mobile smart things. The location service approach, on the other hand, comes very close to our location model and forms a subset of it, since we support additional features, such as locations within smart things, as well as the location domain concept that provides better scalability with local handovers.

7.3.4 Cooltown

Cooltown [9, 25, 63, 64, 65, 67, 88] extends well-known web technologies to support mobile users with electronic devices such as PDAs to allow for the support of the mobile user's activities. Its main abstractions are people, places and things. One generic Cooltown scenario is a user that carries a PDA with an Auto-ID system which it uses to scan identifiers that are attached to objects. These IDs can be URLs, or else a resolver has to map them to a URL, which in turn is used to contact the corresponding web server to show the "homepage" of the object, which provides additional services around this object. Another Cooltown scenario refers to the e-squirt concept. In contrast to the content pull with the homepage of an object, the e-squirt concept enables a content push by means of scanning a URL from a source and sending it to a sink with its PDA. Besides the homepage of an object, a place can also possess a homepage that offers location-aware services that can be linked together to hierarchies, so that Cooltown also provides a hierarchical management of locations that can contain objects.

In Cooltown, the scanning and the resolution of identifiers is a central issue since it has to retrieve the homepage of an object or support the e-squirt concept. Thus, Cooltown has concentrated on this issue, too. First, it states the tasks associated with an ID: creation, binding to a physical or virtual entity, and capturing of an ID, followed by a resolution process where additional data can influence the resolution process. The result of the resolution process can be a resource or a list of links. An ID itself has to fulfil several requirements: uniqueness, non-exhaustive supply of new IDs, integration with existing schemes, human tractability, and a simple generation of new IDs. Based on these properties, Cooltown proposes an ID called 'tag' that should not be mixed up with

our definition of a tag. Such a tag is unique over time and space and easy to generate, since an already unique name, e.g. an e-mail address, is combined with a timestamp.

We think that the use of the web as platform to enable smart things is an appropriate approach in order to enable human interaction with smart things, since the web is intended to support user interaction and to provide linked information. Since we put the focus on the communication between the smart things, the Cooltown approach can be regarded as complementary to our model. The issues concerning the IDs are also covered by our smart thing systems, as has been explained in the deployment schemes of Chapter 6.

7.3.5 Hive

Hive [80, 89] is a middleware project that aims to support the concepts of the Things That Think project that we introduced in Chapter 1. Its main abstractions are agents, cells, and shadows. Cells are the execution environment for agents that can migrate from one cell to another. Agents can communicate locally within one cell or communicate remotely with other agents. Shadows represent access to local resources and they can be contacted by agents only locally. A hive network is a loose bunch of cells and ideally, one cell would be located on one object or device and would host one agent. Such a scenario might be too costly concerning the resources on an object, so that a cell can also be executed as a process on a workstation and would be responsible for several devices. The support of migration considers the static code as well as the dynamic state of an agent.

In comparison with our model, an agent would be a representation, a cell would be a hosting service, and a shadow represents the access to a tag's sensors. Implicit coupling would be realized as a cell that is hosted directly on the object. In summary, the concepts they introduce can also be found in our model, but as our model shows, they are a subset of the whole model.

7.3.6 Sylph

Sylph [22] is a middleware that registers sensors as services at arbitrary service discovery platforms. The main component of this system is the proxy core that communicates with both sides, i.e. it queries the sensors and it registers the sensors as services. Additionally, it handles the communication between the application and the sensors after an application has downloaded a service module that has been created and registered by the proxy core beforehand. At the startup of the service proxy, it reads a configuration file in order to initialize the sensors – a sensor can either be contacted directly or it can be contacted via a base station, so that wireless sensors and sensor boards are supported, too. An application that has downloaded a service module from a service discovery platform can specify a sensor request in an SQL-like Transducer Query Language that allows for the specification of the sensor types, a predicate clause that must be fulfilled, an interval for how often the value is needed and the duration for how long the sensor value should be recorded. A simple sensor cannot process such complex queries, so that the proxy core maps these queries into corresponding simple get and set-methods that can be understood by a sensor.

Sylph provides concepts concerning the communication between an application and the actual sensor that are comparable with our approach. Although we do not need a service discovery platform, since the association between a representation and the sensors

on the corresponding tag is unique, we introduce a tag service. This tag service is comparable with the proxy core since, on the one hand, it understands the sensor protocol and, on the other hand, it maps the sensor protocol messages to simple commands that are communicated to the sensors. A major difference is that we also consider the offline state, where the representation has no communication channel to the tag.

7.3.7 ParcTab

In conclusion, we want to look at the first ubiquitous computing project from Marc Weiser. The ParcTab project [119, 120, 121, 122, 123] considers pager-sized computers with a small display and three buttons that are intended to provide their users with services, includingg location-based services. Besides the pager-sized computers, Weiser also envisions notebook-sized computer and larger displays on walls. Due to the low computing power of the ParcTabs, they have a proxy in the background infrastructure: the tab agent. The communication between the two is based on IR, i.e. every ParcTab has an IR transceiver and another IR transceiver is mounted at the ceiling of an office room, which additionally has a connection to the background infrastructure. The tab agent can send commands from the tab remote procedure call to its tab, which acts accordingly, but the other way round is also possible through event communication from the tab to the tab agent. An IR gateway that receives such an event first has to consult a name service to look up the tab agent, so that the gateway can forward the event to the tab with additional symbolic location information that can be processed by the tab agent.

Although Weiser used a small electronic device instead of considering smart things, and did not explicitly mention a tag detection system and a representation, the system implicitly implements these concepts. Compared with our model, the ParcTab is the thing, the tab agent is the representation, and the IR system acts as a tag detection system, so that Weiser unwittingly implemented our high level concepts in this very early stage of ubiquitous computing.

7.4 Summary

The goal of this chapter was to show how our work is related to other work that also has to consider the same or similar aspects as this work. Since a smart thing infrastructure as we envision it did not exist, we had to develop our concepts from scratch. The related work is either complementary to our approach or provides a subset of our concepts, so that our model as a whole and each of its concepts are innovative and provide a new and overall view of systems that enable smart thing applications.

We divided the related work into adjacent domains, smart thing systems, and ubiquitous computing systems. First we looked at the adjacent domains that refer to naming and addressing, location models, Cellular IP, and artificial intelligence. Although these aspects do not primarily refer to smart thing systems, they provide features that are also necessary in our domain. We then looked at four other smart thing systems. Three of them, i.e. Savant, SII and VisuM, do not provide a representation, but manage the static data that occurs during the tag detection process. The RAUM project, on the other hand, provides a representation and enables location-aware communication between smart artefacts, so that this system can be regarded as the first system that goes in the direction of our vision. Finally, we briefly described eleven ubiquitous computing

middleware projects and summarized the common task they support. Five of them, i.e. Nexus, Cooltown, Hive, Sylph, and ParcTab, have been described in more detail, since they cover aspects that are relevant for our model.

Chapter 8

Conclusions

In this final chapter, we want to draw some conclusions from the work presented in our dissertation. For this purpose, we first summarize the main points of the dissertation, followed by a discussion of its contribution. Finally, we conclude with some statements about how we expect the field of smart thing systems might develop in the future.

8.1 Summary

Based on Weiser's vision of ubiquitous computing, Gershenfeld's Things That Think, the understanding of pervasive computing by the industry, and the technological drivers of ubiquitous computing that Mattern proposes, we formulated our view on smart things. A smart thing is an everyday item with a representation that provides additional functions: we require that a smart thing has an identity, knows its own location, all the other smart things in its proximity, is able to communicate with other smart things, and is able to monitor its environment and its own state.

We showed that the realization of our view of smart things contributes to bridging the media break. Data that refers to processes in the real world needs to be transferred into computer systems in order to manage these processes. Since the transmission is mostly done manually, e.g. with a keyboard or semi-automatically, e.g. scanning a bar code, the data is transmitted via other media, which we call a media break. Two consequences of the media break are faulty data and slow data entry. Smart things that bridge the gap between the real world and the virtual world help to prevent these consequences. One generic application that was used throughout this work is a smart supply chain where smart things can automatically check in at the warehouse management systems and are able to monitor their temperature. Building on this generic example we derived the requirements of systems that enable such smart thing applications. Identification and localization of smart things or the support of sensors and actuators are examples of these requirements.

In order to support smart thing systems, one can make use of several existing software and hardware technologies. Jini and Web Services are two service discovery platforms that can be used as an underlying middleware system to distribute the components of our system in a network. We also described Auto-ID systems such as bar codes or RFID, as well as wireless networks such as Cellulars, WLANs and PANs as means to enable identification and to transmit further information from a thing to a background infrastructure. Besides the identification systems, we introduced localization systems with location models and localization methods that are needed to determine the location

of a smart thing. Since smart things might be able to perceive their environment and to manipulate it, we briefly introduced sensors and actuators.

After this preparatory work, we presented a model of collaborating everyday items. This model as the key contribution of this dissertation points out the relevant concepts in smart thing systems and shows how they depend on each other. First, the high level concepts that comprise a smart thing, namely, a tag detection system and managing services were introduced. Then, the more specific concepts were introduced that refer to the basic abilities of a smart thing: an identifier, locations, a location model, and sensor and actuator data. These basic abilities were used to explain in more detail the concepts of a smart thing that mainly refer to representation. Complementary to the smart thing itself, the concepts of the infrastructure were introduced next: a tag, a tag reader, a tag detection service, a home service, a hosting service, and location manager services. These concepts provide the actual means to enable smart things. Since all of these concepts are related to each other, we described the procedure for their interaction. For this purpose, we introduced communication channels for the basic abilities, the identification and the localization process, the update of the location managers, and the method of sensor and actuator communication between a tag and its representations. Next, we introduced three extensions to this procedure, whose handling is more complex than the standard procedure. The first extension refers to containedness, i.e. a smart thing that contains other smart things. The second extension considers the situation if a tag is detected simultaneously by several tag readers, and the third extension looks at smart things that have multiple tags, e.g. a bar code and an RFID tag. Besides these main concepts, we also had to consider how the application logic is distributed over smart things and separate applications. Finally, we modeled the lifecycle of a smart thing. Chapter 4.9 provides a short overview of how these components actually interact.

The model has been iteratively developed in conjunction with the development of three smart thing systems that implement the concepts as a proof of concept. We showed how we had implemented the concepts in our three systems: first, we started with the Voxi system that builds on Jini and that enables the migration of representations as well as a simple notion of location. Then, we described the Wsst system that makes use of Web Services and supports location hierarchies as well as the composition relation. Finally, we presented the Iceo system that again builds on Jini and explicitly supports the tag detection system as well as sensors and actuators. These systems also permit the testing of different implementation strategies that are independent of the actual concepts, such as the utilization of different middleware systems or the use of asynchronous event communication vs. synchronous method invocations.

The overall goal we want to achieve is to enable and to facilitate the development of smart thing applications. Therefore, we implemented the smart supply chain application with all three smart thing systems in order to evaluate the concepts in terms of their usability and effectiveness. This qualitative evaluation showed that the concepts of the model that has been implemented by the three systems can really be used to support such applications. We noticed that the use of a thread for every representation might consume too many resources and that the use of Web Services as the underlying middleware platform might be too slow for our real-time requirements. Thus, we conducted a performance test between Jini and Web Services that confirmed our concerns regarding the performance of Web Services, so that we came up with the recommendation to model the services of our model as a Jini service and a representation as a simple Java object that can be migrated from one hosting service to another. The requirements that we

stated in Chapter 2, derived from the smart supply chain demo, have been implemented by the systems and proven by the example implementation of the smart supply chain with our three systems.

Finally, we looked at the related work, which we split into three categories: adjacent domains, smart thing systems, and ubiquitous computing systems. Under adjacent domains, we considered naming and addressing, location models, Cellular IP, and artificial intelligence. By adjacent we mean that these issues are not primarily related to smart thing systems but have a significant influence on these systems. Next, we considered four existing smart thing projects: the Auto-ID Center, the Smart Items Infrastructure, VisuM, and RAUM. The main difference to our model is that three of them provide no support for a dedicated representation, but rather manage the data coming from a tag reader. RAUM, on the other hand, provides a representation, but it only supports implicit coupling and a limited location model. After this, we looked at eleven ubiquitous computing middleware systems that do not primarily focus on smart things, and extracted the common tasks they support. We took a deeper look at five of these systems, since they have an overlap with our smart thing systems. Generally, we can state that either the related work is complementary to our work, or it provides a subset of our concepts that in fact appropriately models the challenges of its domain, but the concepts are not sufficient to fulfil the requirements of Chapter 2.

8.2 Contributions

We showed that there is a business need for smart things and that currently no existing project adequately models this domain and provides systems that support smart thing applications, except for emerging systems that only allow the passive data management of tag readers. Thus, the main contribution of this dissertation is to provide a complete and consistent picture of the modeling of smart thing systems, as well as recommendations for the implementation and structuring of a smart thing system and the actual applications. In particular, the subtle details of the location model, which supports containedness of smart things, can be denoted as one of the major contributions. In the following, we discuss the individual contributions of the most important concepts of our model.

High level concepts Although terms such as Things That Think, Smart Artefacts, Smart Items, etc. are used with some projects, these concepts are introduced in an intuitive and descriptive manner with the effect that an exact definition is missing. Thus, it is not really clear what makes up a smart thing, so that different people might have different notions of the term, which leads to misunderstandings. These complicate the focus of the research in this domain and the implementation of systems and applications. Our high level concepts that comprise things, representations, tag detection systems, and managing services provide a clear partitioning that serves as a starting point for modeling each high level concept in more detail. In particular, the distinction between implicit coupling and explicit coupling shows two ways of bridging the gap between the two worlds.

Identifier Identifiers in general are a commonly used concept in computer science. Other projects, such as Cooltown, which introduces a tag or the Auto-ID center, which introduces an EPC, propose identifiers. In our opinion these kind of identifiers are not

well suited for our purposes since they rely on additional name services that map the identifier to the actual resource. An additional name service means overheads in order to manage the name service itself and to issue identifiers that are compatible with the name service. In our model the identifier consists of a name and a home address. The home address refers to the host that builds the access point for a representation. It is resolved by the underlying network, so that no additional name services are needed. In most cases the home address will be a DNS name or an IP address. The name can be arbitrary chosen by the organization that manages the representation in order to allow internal optimizations. Although our identifier is considerably longer than an EPC, it does not need an ONS and the allocation of EPCs, so that everyone can run the system without any external dependencies.

Localization One of the major contributions is the location model that supports the containedness relation of smart things, so that we are able to describe a location within a smart thing, e.g. freighters, trucks or just simple containers. Our location model consists of a hybrid location model that describes symbolic locations and physical positions in the static real world, and it consists of a hybrid location model that describes symbolic locations and physical positions in mobile smart things. We require three functions that allow translation between symbolic locations and physical positions and between the smart thing and the world location models. Although a hybrid location model and the translation between location models have already been mentioned in the literature, the combination of a static model with mobile nodes is unique.

We also differentiate between four localization methods that our model supports. The four methods result from the combination of symbolic locations vs. physical positions and localization on the tag vs. localization at the tag reader. This explicit consideration is also unique.

Representation We already mentioned that other projects consider the representation only as a database entry. In our model, we introduce a representation as consisting of static code and dynamic data. This allows for individual behavior of a smart thing that can, dependent on its programming, decide on its own actions and does not depend on other programs that process database entries. This dichotomy allows for efficient migration and data storage, since the static code only needs to be stored or migrated once, so that different instances of the same smart thing type can use the same static code.

Relations of a smart thing As with object oriented programming languages, where the root object class only provides the necessary means to manage objects in the runtime environment, and where the specific behavior of an object class is up to the programmer, we take the same approach. We only model aspects of a representation that are absolutely necessary to manage a smart thing. It is up to the developer of a representation to extend it with the smart thing specific behavior. These core aspects, besides its identifier and its location, comprise three relations: neighborhood, containedness, and composition. We noticed that these relations are common features that need to be supported by the infrastructure. These concepts also support the infrastructure, e.g. the containedness relation allows for the inheritance of sensor data or localization data, so that the infrastructure does not need to update the location information of a contained thing, since this

information can be retrieved from the containing thing. We divided the neighborhood concept into the absolute and the relative neighborhood concept: absolute neighborhood refers to the fact that two smart things are contained at the same smallest location in the location tree, whereas relative location explicitly mentions the location where all the smart things of that location are neighbors. The containedness relation has been split into simple and complex containedness. Complex and simple refer to whether a smart thing has its own tag detection system. In particular, the division of the two concepts is unique.

Management of representations We introduced the home, hosting, and location manager services that are responsible for managing a representation. This partitioning allows for a scalable infrastructure that enables migration of representations. A home service acts as the access point for representations, a hosting service executes a representation, and the location manager services (including hub, base and super location managers) provide the means for efficient administration, since local migrations within the same location domain only generate local network traffic. This approach is also unique: although it is oriented to GSM, it includes the recursive support of contained smart things. The management of representations also considers the aspects of multiple detection of the same tag and the detection of a thing with multiple tags. The working out of these last two concepts was subtle in its details.

Tag detection systems The general tag detection system concept does not depend on the deployed Auto-ID and wireless network technologies. A tag detection system is responsible for the identification and localization of tags. This abstraction, consisting of tags, tag readers, and tag detection services, is also unique.

Communication channel Communication channels allow the concrete communication aspects between a tag and its representation, or the location manager lattice, to be disregarded. A tag has two communication channels to one location manager, which are used to retrieve its identifier and its location. Two further communication channels exist between the tag and the representation: these are used to query the sensors and to control the actuators of a tag. This is necessary, since the concrete implementation can strongly vary. On the one hand, a tag with a GPS module and a WLAN module might directly contact the location manager hierarchy to update its location. On the other hand, an RFID reader might detect RFID tags and the tag detection service has to retrieve the information to contact the location manager. Every communication channel has its own protocol that defines what kind of information can be exchanged.

Application logic Our evaluation has shown that distributing the application logic over smart things and location-specific applications is a suitable approach in order to support smart thing applications. Location-specific applications register themselves at location managers, so that they are able to communicate with the smart things that have been detected at this location. Which functionality should be part of a smart thing and which functionality should be part of a location-specific application is not predetermined, but in most cases obvious.

Recommendations The evaluation of the various smart thing systems has shown that all the concepts of our model can be efficiently implemented and that the concepts effectively support the development of smart thing applications. On the basis of the performance analysis, we recommend implementing the services as Jini services and the representations as Java objects, in a similar way to the procedure in the Iceo system. As stated above, the application logic should be distributed over smart thing and location-specific applications.

8.3 Prospects

This dissertation has shown that the concepts of our model can be implemented in systems that support the development of smart thing applications, so that we currently see no reason to adapt or to extend the core model.

Although we evaluated all of our concepts with our three systems, none of these three systems implements all the concepts, so that a system that implements them all would be desirable. Such a smart thing system could take over most of the existing code.

During the development of the warehouse management systems, we noticed that all location-specific warehouse management systems have some common features that can be provided by a warehouse management framework that builds on our systems. Analogously to a framework that supports warehouse management applications, we assume that frameworks for other scenarios would also be helpful, e.g. frameworks that can be used in production, facility management, retail, etc.

We connected three different tag detection technologies with our smart thing system: RFID, bar code, and BTNodes with sensors and actuators. In addition, it would be desirable to connect other tag detection technologies that we mentioned, such as WLAN or GSM, to our system.

Other aspects that have to be considered are security and privacy. We explicitly did not consider these aspects in order to focus on the inherent aspects of smart thing systems. Security typically refers to issues such as authentication, authorization, integrity, confidentiality, non-repudiation, and delegation, which are normally covered by the underlying middleware platform, such as Jini or Web Services, that provide corresponding security packages that can be used to implement these security aspects. Analogously to the middleware systems, some tag detection hardware also provides security measures such as challenge/response procedures in order to authenticate a tag. We mentioned that a home service has a data storage facility where it stores its dynamic state, which is accessible by every hosting service. If an organization wants to keep the data captured by a representation at one of the organization's hosting services private, the representation can use the local data storage facility of the organization to store the private data that should only be accessible to this hosting service, so that other organizations have no access to this data. Privacy in general comprises more aspects than using this local data storage, so that results from current research into privacy, such as [71, 72, 77] could also be integrated.

Our smart thing system is most advantageous if many things are tagged and the infrastructure is installed at many locations, so that deployment on a larger scale would be another aspect of future work.

Bibliography

[1] Abowd G (1996) Software engineering and programming language considerations for ubiquitous computing. ACM Computing Surveys 18(4), Article 190

[2] Aiken B, Strassner J, Carpenter B, Foster I, Lynch C, Mambretti J, Moore R, Teilbaum B (2000) A Report of a Workshop on Middleware. RFC 2768

[3] AIM Homepage (2003) Auto-ID Manufactures. http://www.aimglobal.org/

[4] Akyildiz I, Su W, Sanakarasubramaniam Y, Cayirci E (2002) Wireless Sensor Networks: A Survey. Computer Networks Journal 38(4): 393-422

[5] Alexander K, Gilliam T, Gramling K, Kindy M, Moogimane D, Schultz M, Woods M (2002) Focus on the Supply Chain: Applying Auto-ID within the Distribution Center. Report IBM-AUTOID-BC-002, Auto-ID Center

[6] Arregui D, Fernström C, Pacull F, Rondeau G, Williamowski J (2003) Stitch: Middleware for Ubiquitous Applications. In: Proceedings of sOc'2003 (Smart Objects Conference), http://www.grenoble-soc.com/proceedings03/Pdf/55-Arregui.pdf

[7] Assistant Secretary of Defense for Command, Control, Communications, and Intelligence (2001) Global Positioning System Standard Positioning Service Performance Standard, http://www.navcen.uscg.gov/gps/geninfo/2001SPSPerformance StandardFINAL.pdf

[8] Bar Code 1 Homepage (2003) A Web of Information About Bar Code, http:// www.barcode-1.net/

[9] Barton J, Kindberg T (2001) The challenges and opportunities of integrating the physical world and networked systems. Technical Report TR HPL-2001-18, HP Labs

[10] Beigl M, Zimmer T, Decker C (2002) A location model for communicating and processing of context. Personal and Ubiquitous Computing 6(5-6): 341-357

[11] Beigl M. (1999) Using spatial Co-location for Coordination in Ubiquitous Computing Environments. In: Proceedings of First International Symposium on Handheld and Ubiquitous Computing, pp. 259-273

[12] Berners-Lee T, Fielding R, Masinter L (1998) Uniform Resource Identifier (URI): Generic Syntax. RFC 2396

[13] Berners-Lee T, Masinter L, McCahill (1994) Uniform Resource Locators (URL). RFC 1738

[14] Brumitt B, Meyers B, Krumm J, Kern A, Shafer S (2000) EasyLiving: Technologies for Intelligent Environments. In: Proceedings of the 2nd International Symposium on Handheld and Ubiquitous Computing, pp. 12-27

[15] Brumitt B, Shafer S (2001) Topological World Modeling Using Semantic Spaces. In: Proceedings of the Workshop on Location Modeling for Ubiquitous Computing, pp. 55-62, http://www.teco.edu/locationws/final.pdf (30.09.03)

[16] Burkhardt J, Henn H, Hepper S, Rindtorff K, Schäck, T (2001) Pervasive Computing. Addision-Wesley, ISBN 3-8273-1729-0

[17] Campbell A, Gomez J, Kim S, Valko A, Wan C, Turanyi Z (2000) Design, Implementation, and Evaluation of Cellular IP. IEEE Personal Communications 4: 42-49

[18] Campbell A, Gomez J, Valko A (1999) An Overview of Cellular IP. IEEE Wireless Communications and Networks Conferance (WCNC'99) 2: 606-611

[19] Carpa L, Emmerich W, Mascolo C (2002) Middleware for Mobile Computing. Advanced Lectures on Networking, Springer-Verlag, Networking 2002 Tutorials, pp. 20-58

[20] Carriero N, Gelernter D (1988) Applications experience with Linda. In Proceedings of the ACM SIGPLANPPEALS, ACM Symposium on Parallel Programming 23: 173-187

[21] Cauldwell P, Chawla R, Chopra V, Damschen G, Dix C, Hong T, Norton F, Ogbuji U, Richman M, Saunders K, Zaev Z (2001) Professional XML Web Services. Wrox Press, ISBN 1-861005-09-1

[22] Chen A, Muntz R, Yuen S, Locher I, Park S, Srivasta M (2002) A Support Infrastructure for the Smart Kindergarten. IEEE Pervasive Computing 1(2): 49-57

[23] Chen G, Kotz D (2000) A Survey of Context-Aware Mobile Computing Research. Dartmouth Computer Science Technical Report, TR2000-381

[24] Coulouris G, Dollimore J, Kindberg T (1998) Distributed Systems - Concepts and Design. Addison-Wesley, ISBN 0-201-62433-8

[25] Daniel R (1997) A Trivial Convention for using HTTP in URN Resolution. RFC 2169

[26] Domnitcheva S (2001) Location Modeling: State of the Art and Challenges. In: Proceedings of the Workshop on Location Modeling for Ubiquitous Computing, http://www.teco.edu/locationws/final.pdf, pp. 13-19

[27] Dübendorfer T (2001) An Extensible Infrastructure and a Representation Scheme for Distributed Smart Proxies of Real World Objects. Technical Report 359, ETH Zurich, Institute of Information Systems

[28] EAN.UCC White Paper on Radio Frequency Identification (1999) EAN International, http://www.autoid.org/SC31/clr/200305_3821_EANUCC

[29] Edwards W (1999) Core Jini. Prentice Hall, ISBN 0-13-014469-X

[30] Eicher T (2003) Smart Things - Modeling the Spirit. Diplomarbeit, ETH Zurich, Institute for Information Systems

[31] Elpas Ltd (2001) EIRIS with Web/WAP Capabilities. Case study, http://www.elpas.com/Downloads/Uploads/Pacific.pdf

[32] Esler M, Hightower J, Anderson T, Borriello G (1999) Next Century Challenges: Data-Centric Networking for Invisible Computing. In: Proceedings of MOBICOM'99, pp. 256-262

[33] EUROCONTROL, Institute of Geodesy and Navigation (1998) WGS 84 Implementation Manual, http://www.wgs84.com/files/wgsman24.pdf

[34] Finkenzeller K (2002) RFID-Handbuch. Carl Hanser Verlag, ISBN 3-446-22071-2

[35] Fleisch E, Mattern F, Billinger S (2003) Betriebswirtschaftliche Applikationen des Ubiquitous Computing. In: Heinz Sauerburger (Ed.): Ubiquitous Computing, HMD 229 - Praxis der Wirtschaftsinformatik, dpunkt.verlag, ISBN 3-89864-200-3, pp. 5-15

[36] Fleisch E, Mattern F, Österle H (2002) Betriebliche Anwendungen mobiler Technologien: Ubiquitous Commerce. Computerwoche (CW-Extra, Themenheft "Collaborative Commerce / Neue Geschäftsprozesse")

[37] Flörkemeier C, Lampe M, Schoch T (2003) The Smart Box Concept for Ubiquitous Computing Environments. In: Proceedings of sOc'2003 (Smart Objects Conference), pp. 118-121

[38] Franklin S, Graesser A (1996) Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, pp. 21-36

[39] Friday A, Davies N, Catterall E (2001) Supporting service discovery, querying and interaction in ubiquitous computing environments. In: Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access table of contents, pp. 7-13

[40] Geihs K (2001) Middleware Challenges Ahead. IEEE Computer 34(6): 24-31

[41] Gershenfeld N (1999) When Things Start To Think. Henry Holt, ISBN 0-8050-5874-5

[42] Gloss B (2002) Ortsbewusste Anwendungen mit Nexus. Beiträge zur ITG Fachtagung Neue Kommunikationsanwendungen in modernen Netzen, pp. 179-180

[43] Grimm R (2002) System support for pervasive applications. Ph.D. Thesis, University of Washington

[44] Grimm R, Anderson T, Bershad B, Wetherall D (2000) A system architecture for pervasive computing. In Proceedings of the 9th ACM SIGOPS European Workshop, pp. 177-182

[45] Grimm R, Davis J, Lemar E, MacBeth A, Swanson S, Gribble S, Anderson T, Bershad B, Boriello G, Wetherall D (2001) Programming for Pervasive Computing Environments. University of Washington, Technical Report, UW-CSE-01-06-01

[46] Hauzeur B (1986) A Model for Naming, Addressing, and Routing. ACM Transactions on Office Information Systems 4(4): 293-311

[47] Henricksen K, Indulska J, Rakotonirainy A (2001) Infrastructure for Pervasive Computing: Challenges. GI Jahrestagung (1), pp. 214-222

[48] Hess C (2003) The Design and Implementation of a Context-Aware File System for Ubiquitous Computing Applications. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Computer Science

[49] Hess C, Campbell R (2002) A Context File System for Ubiquitous Computing Environments. Technical Report UIUCDCS-R-2002-2285 UILU-ENG-2002-1729, University of Illinois at Urbana-Champaign, Urbana-Champaign, Computer Science

[50] Hightower J, Borriello G (2001) Location Sensing Techniques. Technical Report, University of Washington, Computer Science and Engineering

[51] Hightower J, Borriello G (2001) Location Systems for Ubiquitous Computing. IEEE Computer magazine 34(8): 57-66

[52] Hightower J, Brumitt B, Borriello G (2002) The location stack: A layered model for location in ubiquitous computing. In Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA 2002), pp. 22-28

[53] Hightower J, Want R, Borriello G (2002) SpotON: An Indoor 3D Location Sensing Technology Based on RF Signal Strength, Technical Report UW CSE 00-02-02, University of Washington, Department of Computer Science and Engineering

[54] Hrachovec H (2001) Computernamen im Internet. Techno-phänomenologische Aspekte. In: Journal Phänomenologie, Heft 15/2001, pp. 15-24

[55] Hupfeld F, Beigl M (2000) Spatially aware local communication in the RAUM system. In: Proceedings of the IDMS, pp. 285-296

[56] IEEE Robotics & Automation Magazine (1999) 6(1)

[57] Immer Anschluss unter dieser Nummer (2003) Frankfurter Allgemeine Zeitung, November 24th, http://fazarchiv.faz.net/

[58] JavaTM Remote Method Invocation (RMI) (2003) Sun Microsystems, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/index.html

[59] Jiang C, Steenkiste P (2002) A Hybrid Location Model with a Computable Location Identifier for Ubiquitous Computing. In: Proceedings of the 4th international conference on Ubiquitous Computing, pp. 246-263

[60] Johanson B, Fox A (2002) The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In: Proceedings of 4th IEEE Workshop on Mobile Computing Systems and Applications, pp. 83-93

[61] Johanson B, Fox A, Winograd T (2002) The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. IEEE Pervasive Computing 1(2): 67-75

[62] Kambil A, Brooks J (2002) Auto-ID Across the Value Chain: From Dramatic Potential to Greater Efficiency & Profit. Report ACN-AUTOID-BC-001, Auto-ID Center

[63] Kindberg T (2001) Ubiquitous and contextual identifier resolution for the real-world wide web, Technical Report HPL-2001-95R1, HP Labs

[64] Kindberg T, Barton J (2001) A Web-Based Nomadic Computing System. Computer Networks 35(4): 443-456

[65] Kindberg T, Barton J, Morgan J, Becker G, Caswell D, Debaty P, Gopal G, Frid M, Krishnan V, Morris H, Schettino J, Serra B, Spasojevic M (2000) People, places, things: Web presence for the real world. In: Proceedings of Third IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'00), pp. 19-30

[66] Kindberg T, Fox A (2002) System Software for Ubiqutous Computing. IEEE Pervasive Computing 1(1): 70-81

[67] Kindberg, T, Barton J (2000) Towards a Real-World Wide Web, Technical Report HPL-2000-47

[68] Kirchner H, Schönfeld W, Steinmetz R (2002) Verfahren zur Diensterkennung und Dienstinformationsbereitstellung im Vergleich. ITG Fachtagung Neue Kommunikationsanwendungen in modernen Netzen, pp. 19-27

[69] Kubach U (2003) Betriebswirtschaftliche Applikationen des Ubiquitous Computing. In: Heinz Sauerburger (Ed.): Ubiquitous Computing, HMD 229 - Praxis der Wirtschaftsinformatik, dpunkt.verlag, ISBN 3-89864-200-3, pp. 56-67

[70] Langheinrich M (2000) The 5th Dimension: Building Blocks for Smart Infrastructures. Workshop on Infrastructure for Smart Devices - How to Make Ubiquity an Actuality, http://www.vs.inf.ethz.ch/events/HUK2kW/

[71] Langheinrich M (2001) Privacy by Design - Principles of Privacy-Aware Ubiquitous Systems. In: Proceedings of Ubicomp 2001, pp. 273-291

[72] Langheinrich M, Coroama V, Bohn J, Rohs M (2002) As we may live - Real-world implications of ubiquitous computing. Technical Report, ETH Zurich, Institute for Information Systems

[73] Langheinrich M, Mattern F, Römer K, Vogt H (2000) First Steps Towards an Event-Based Infrastructure for Smart Things. Ubiquitous Computing Workshop (PACT 2000), http://www.vs.inf.ethz.ch/publ/papers/firststeps.pdf

[74] Leonhardi A, Kubach U (1999) An Architecture for a Distributed Universal Location Service. In: Proceedings of the European Wireless '99 Conference, pp. 351-355

[75] Leonhardt U (1998) Supporting Location-Awareness in Open Distributed Systems. Ph.D. Thesis, Department of Computing, Imperial College London

[76] Mattern F (2003) Vom Verschwinden des Computers - Die Vision des Ubiquitous Computing. In: Friedemann Mattern (Ed.): Total vernetzt, Springer-Verlag, pp. 1-41

[77] Mattern F, Langheinrich M: Allgegenwärtigkeit des Computers? Datenschutz in einer Welt intelligenter Alltagsdinge (2001) In: Müller G, Reichenbach M (Hrsg.): Sicherheitskonzepte für das Internet, Springer-Verlag, pp. 7-26

[78] Mattern F, Sturm P (2003) From Distributed Systems to Ubiquitous Computing - The State of the Art, Trends, and Prospects of Future Networked Systems. In: Irmscher K, Fähnrich KP (eds) Conference Proceedings der Fachtagung Kommunikation in Verteilten Systemen. Springer-Verlag, pp. 3-25

[79] Meyers B, Kern A (2000) <Context-Aware> schema </Context-Aware>. CHI Workshop on The What, Who, When, Where, Why, and How of Context-Awareness, http://www.cc.gatech.edu/fce/contexttoolkit/chiws/Meyers.doc

[80] Minar N, Gray M, Roup O, Krikorian R, Maes P (1999) Hive: Distributed Agents for Networking Things. In: Proceedings of the First International Symposium on Mobile Agents, pp. 118-129

[81] Moats R (1997) URN Syntax. RFC 2141

[82] Muller N (2001) Bluetooth. MITP-Verlag, ISBN 3-8266-0738-4

[83] Nicklas D, Grossmann M, Schwarz T, Volz S, Mitschang B (2001) A model-based, open architecture for mobile, spatially aware applications. In: Proceedings of the 7th International Symposium on Spatial and Temporal Databases, pp. 117-135

[84] Nicklas D, Mitschang B (2001) The Nexus Augmented World Model: An Extensible Approach for Mobile, Spatially-Aware Applications. In: Proceedings of the 7th International Conference on Object-Oriented Information Systems, pp. 392-401

[85] Oat Systems (2002) The Savant - Version 0.1 (Alpha). Oat Systems & Massachusetts Institute of Technology (MIT) Auto-ID Center, Cambridge, http://www.autoidcenter.org/research/MIT-AUTOID-TM-003.pdf

[86] Orr R, Abowd G (2000) The Smart Floor: A Mechanism for Natural User Identification and Tracking. GVU Report GITGVU-00-02, Graphics, Visualization and Usability (GVU) Center, Georgia Institute of Technology

[87] Ponnekanti S, Lee B, Fox A, Hanrahan P, Winograd T (2001) ICrafter: A Service Framework for Ubiquitous Computing Environments. In: Proceedings of Ubicomp 2001: Ubiquitous Computing: Third International Conference, pp. 56-75

[88] Pradhan S (2000) Semantic Location. Personal and Ubiquitous Computing 4(4): 213-216

[89] Rhodes B, Minar N, Weaver J (1999) Wearable Computing Meets Ubiquitous Computing - Reaping the best of both worlds. In: Proceedings of The Third International Symposium on Wearable Computers (ISWC '99), pp. 141-149

[90] Roman M (2003) An Application Framework for Active Space Applications. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Computer Science

[91] Roman M, Hess C, Cerqueria R, Ranganat A, Campbell R, Nahrstedt K (2002) Gaia: A Middleware Infastructure to Enable Active Spaces. IEEE Pervasive Computing 1(4): 74-83

[92] Roth J (2002) Mobile Computing. dpunkt.verlag, ISBN 3-89864-165-1

[93] Russel S, Norvig P (1995) Artificial Intelligence - A Modern Approach. Prentice Hall, ISBN 0-13-360124-2

[94] Römer K, Schoch T (2002) Infrastructure Concepts for Tag-Based Ubiquitous Computing Applications. Workshop on Concepts and Models for Ubiquitous Computing at Ubicomp 2002, http://www.vs.inf.ethz.ch/publ/papers/infrastructure-concepts.pdf

[95] Römer K, Schoch T, Mattern F, Dübendorfer T (2003) Smart Identification Frameworks for Ubiquitous Computing Applications. In: Proceedings of PerCom 2003 (IEEE International Conference on Pervasive Computing and Communications), pp. 253-262

[96] Römer K, Schoch T, Mattern F, Dübendorfer T (2004) Smart Identification Frameworks for Ubiquitous Computing Applications. To appear in: Wireless Networks 10(6)

[97] Schnellere Hilfe durch Handy-Ortung (2003) Frankfurter Allgemeine Zeitung, December 3rd, http://fazarchiv.faz.net/

[98] Schoch T (2000) An Authentication and Authorization Architecture for Jini Services. Diplomarbeit, Darmstadt University of Technology, Deptartment of Computer Science

[99] Schoch T, Krone O, Federrath H (2001) Making Jini Secure. In: Proceedings of 4th International Conference on Electronic Commerce Research, pp. 276-286

[100] Schwägli T (2002) Jini vs. Web Services - Ein Leistungsvergleich. Semesterarbeit, ETH Zurich, Institute for Information Systems

[101] Schädler S (2002) Smart Things - Einsatz von Web Services zur Modellierung von intelligenten Alltagsgegenständen. Diplomarbeit, ETH Zurich, Institute for Information Systems

[102] Siegemund F (2004) A Context-Aware Communication Platform for Smart Objects. In: Proceeedings of PERVASIVE 2004 (Pervasive Computing: Second International Conference), pp. 69-86

[103] Siegemund F, Flörkemeier C, Vogt H (2004) The Value of Handhelds in Smart Environments. In: Proceedings of ARCS 2004 (17th International Conference on Architecture of Computing Systems - Organic and Pervasive Computing), pp. 291-308

[104] Siegemund F, Krauer T (2004) Integrating Handhelds into Environments of Cooperating Smart Everyday Objects. To appear in: Proceedings of EUSAI 2004 (2nd European Symposium on Ambient Intelligence)

[105] Sousa J, Garlan D (2000) Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In: Software Architecture: System Design, Development, and Maintenance (Conference Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture), pp. 29-43

[106] Sriram T, Vishwanatha R, Biswas S, Ahmed B (1996) Applications of barcode technology in automated storage and retrieval systems. In: Proceedings of the 1996 IEEE IECON 22nd International Conference, pp. 641-646

[107] Steiner M (2003) Smart Infrastructure - Dienste für eine Welt schlauer Gegenstände. Diplomarbeit, ETH Zurich, Institute for Information Systems

[108] Strassner M, Schoch T (2002) Today's Impact of Ubiquitous Computing on Business Processes. In: Short Paper Proc. International Conference on Pervasive Computing, pp. 62-74

[109] Strassner M, Schoch T (2003) Wie smarte Dinge Prozesse unterstützen. In: Heinz Sauerburger (Ed.): Ubiquitous Computing, HMD 229 - Praxis der Wirtschaftsinformatik, pp. 23-31

[110] Su Z (1983) Identification in computer networks. In: Proceedings of the eighth symposium on Data communications, pp. 51-55

[111] Sun Microsystems (2003) JXTA v2.0 Protocols Specification, http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html

[112] Thayer S, Steenkiste P (2003) An Architecture for the Integration of Physical and Informational Spaces. Personal and Ubiquitous Computing 7(2): 82-90

[113] The GCI Intelligent Tagging Model (2001) Report, Global Commerce Initiative, http://www.autoid.org/SC31/clr/200305_3827_GCI

[114] Vogt H (2002) Efficient Object Identification With Passive RFID Tags. In: Proceedings of International Conference on Pervasive Computing, pp. 98-113

[115] Vogt H (2002) Multiple Object Identification with Passive RFID Tags. AutoID invited session at SMC '02 conference, http://www.vs.inf.ethz.ch/publ/papers/smc02rfid.pdf

[116] Wang H, Raman B, Chuah C, Biswas R, Gummadi R, Hohlt B, Hong X, Kiciman E, Mao Z, Shih J, Subraimanian L, Zhno B, Joseph A, Katz R (2000) ICEBERG: an Internet core network architecture for integrated communications. IEEE Personal Communications 7(4): 10-19

[117] Wang Z, Garlan D (2000) Task-Driven Computing. Technical Report CMU-CS-00-154, Computer Science Department, School of Computer Science, Carnegie Mellon University

[118] Want R, Fishkin K, Gujar A, Harrison B (1999) Bridging Physical and Virtual Worlds with Electronic Tags. In: Proceedings of SIGCHI '99, pp. 370-377

[119] Want R, Schilit B, Adams N, Gold R, Peterson K, Goldberg D, Ellis J, Weiser M (1995) The ParcTab Ubiquitous Computing Experiment. Xerox Palo Alto Research Center Technical Report, CSL 95-1

[120] Weiser M (1991) The computer for the twenty-first century. Scientific American 44(9-20): 94-104

[121] Weiser M (1993) Hot topics - ubiquitous computing. IEEE Computer 26(10): 71-72

[122] Weiser M (1993) Some computer science issues in ubiquitous computing. Communications of the ACM 36(7): 75-85

[123] Weiser M (1994) The world is not a desktop. ACM Interactions 1(1): 7-8

[124] Westhoff A (2003) Bridging the Gap - Kommunikation von Gegenständen mit Repräsentationen. Diplomarbeit, ETH Zurich, Institute for Information Systems

Appendix A

Curriculum Vitae

	Personal data
	Thomas Marcus Schoch Born June 13, 1977 Unmarried Computer scientist Citizen of Germany
	School
1993-1996	Gymnasiale Oberstufe Claus-von-Stauffenberg-Schule, Abitur diploma, mark 1.2, best student of the year, collaboration in setting up the class schedule, library software
1983-1996	Elementary school and comprehensive school 3 years in a row class representative best student of the year
	University
2001-2004	Setting up and participation in the third-party funded M-Lab project, key account for Volkswagen
2001-2004	Assistant at ETH Zurich, Computer Science Department, distributed systems group, tutorship for seminars and classes, supervision of student projects and theses

1996-2000	Study of computer science at Darmstadt University of Technology, subsidiary subject business administration, computer science diploma with honors, mark 1.18, course completed in 8.5 semesters compared to the average 14.3 semesters required
1997-2000	Scholar funded by Studienstiftung des deutschen Volkes (German national foundation), participation in various programs
2000	Master's thesis at International Computer Science Institute, Berkeley, USA
1998	Exhibition at CeBIT, Hanover for Bosch Telekom GmbH and Darmstadt University of Technology

Professional

1996-2002	Software development, freelance, Rock 'n' Roll tournament management, timekeeping
1995-2002	Private lessons, freelance, individualized teaching private and foursome groups at institutes
1996-1999	DTP, freelance for VICTORIA AG
1999-2000	Tutorship, Darmstadt University of Technology
1997-1999	System administration, Darmstadt University of Technology
1997	Laboratory report (CeBIT), Darmstadt University of Technology

Additional items

1994-1996	Home care of critically ill father
Foreign languages	German (native language), English (USA stay), French, elementary Italian
Miscellaneous	Driver's license, class 3 (Germany)