Diss. ETH No. 15833

Cooperating Smart Everyday Objects – Exploiting Heterogeneity and Pervasiveness in Smart Environments

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

for the degree of
Doctor of Technical Sciences

presented by
Frank Siegemund
Diplom-Informatiker, University of Rostock, Germany
born May 23, 1976
citizen of Germany

accepted on the recommendation of
Prof. Dr. Friedemann Mattern, examiner
Prof. Dr. Clemens H. Cap, co-examiner

2004

Prototype tabs, pads and boards are just the beginning of ubiquitous computing. The real power of the concept comes not from any one of these devices; it emerges from the interaction of all of them.

Mark Weiser

Acknowledgments

First of all, I would like to thank my supervisor, Prof. Friedemann Mattern, for giving me the opportunity to work in his research group at the ETH Zurich. He provided me with the right amount of freedom to choose my research topic, and I am also extremely grateful for his constant support and his remarks on preliminary versions of this thesis.

I am also deeply indebted to my co-adviser, Prof. Clemens H. Cap. He had already supervised my diploma thesis, and it is he from whom I learned a great deal about how to approach a complex research problem and how to publish scientific papers; I would also like to thank him for the extremely valuable remarks regarding the style and scientific contributions of this thesis.

During my first three years in Zurich, I had the opportunity to work in the EU-funded project *Smart-Its*. Most importantly, the scope of this project – interconnected embedded technology for smart artifacts with collective awareness – largely influenced my research and helped me to find an appropriate research topic. The exchange of experiences with the project partners in Finland, Sweden, England, Germany and Switzerland turned out to be a fruitful soil for interesting research ideas. I would like to thank all the participants in this project for their help, especially the coordinator of the Smart-Its project, Prof. Hans Gellersen, Dr. Michael Beigl for all the extremely valuable input regarding the Smart-Its hardware platforms, Prof. Bernt Schiele, Stavros Antifakos, Christian Decker, Oliver Kasten, Florian Michahelles, Dr. Albrecht Schmidt, Martin Strohbach, and Tobias Zimmer.

During my time at ETH I had the opportunity to work with extremely intelligent and kind people in the Distributed Systems research group, which is led by Prof. Friedemann Mattern. I would like to thank all of them: Jürgen Bohn, Vlad Coroamă, Svetlana Domnitcheva, Christian Flörkemeier, Christian Frank, Oliver Kasten, Matthias Lampe, Marc Langheinrich, Marie-Luise Moschgath, Matthias Ringwald, Christof Roduner, Michael Rohs, Kay Römer, Silvia Santini, Thomas Schoch, and Harald Vogt. I am also grateful to my colleague Michael Rohs for reading extracts of my dissertation.

I am also indebted to Jakub Ratajsky, Loïc Pfister, and Jörg Himmelreich for their support. Jörg Himmelreich read parts of my dissertation and provided valuable insights from the perspective of an architect (I considered all of his comments except his complaints about the small amount of footnotes). Nick Bell was so kind as to read my thesis, and provided input regarding orthography, grammar, and style. I would like to thank him for his efforts.

My family has always been a constant source of understanding and never-ending moral support. I know I can never repay them, but I would like to seize the opportunity to thank my family for all it has given to me.

I thank God.

Contents

Abstract

Pervasive Computing envisions the seamless integration of computation into everyday environments in order to support people in their daily activities. Due to recent technological advances, the computing devices that can be used to enrich our environments with computation have become increasingly smaller and less obtrusive, so that it is possible to embed them into mundane everyday things. Such smart everyday objects have the ability to communicate with peers, perceive their environment through sensors, and provide context-aware services to nearby people. However, the severe resource restrictions and limited user interfaces of smart objects make it difficult for them to realize services on their own. Instead, in order to implement context-aware services and because of their limited user interfaces, smart objects need to interact with other computing devices to make use of these devices' sensors and input/output capabilities. In consequence, there is an urgent need to facilitate cooperation between computational entities in smart environments.

This dissertation addresses the problem of cooperation from the perspective of smart everyday objects. Its main contribution is to illustrate how augmented items can make up for their limited resources by cooperating with nearby computing devices. For this purpose, we concentrate on two aspects of cooperation: interaction between smart objects and cooperation between smart objects and mobile user devices.

The first part of this dissertation focuses on how smart objects can provide services in cooperation with other augmented artifacts. In the proposed solution, a programming abstraction facilitates the design of collaborative context-aware services for smart objects. This abstraction groups nodes into sets of cooperating objects that bundle their resources and appear to an application as a single entity. Based on this approach, we present a software framework for realizing cooperative services on smart objects. It consists of (1) a description language for context-aware services, (2) a context recognition layer for smart objects, (3) an infrastructure layer for distributing data among cooperating artifacts, and (4) a communication layer that adapts networking structures to correspond to the real-world environment of smart objects. To evaluate the proposed concepts, we present an implementation on an embedded sensor node platform, the BTnodes.

The second part of this dissertation focuses on cooperation between smart objects and mobile user devices as an example of how augmented artifacts can benefit from the heterogeneity of smart environments. In this respect, our contribution is to identify recurring usage patterns that describe how smart objects can make use of handheld devices: (1) as mobile infrastructure access points, (2) as user interfaces, (3) as remote sensors, (4) as mobile storage media, (5) as remote resource providers, and (6) as user identifiers. We describe these usage patterns, report on experiences with prototypical implementations, and present several application scenarios illustrating the applicability of the proposed concepts.

Zusammenfassung

Aufgrund des bereits seit einigen Jahrzehnten anhaltenden rasanten technologischen Fortschrittes in der Informations- und Kommunikationstechnik werden Computer immer kleiner, schneller und bei gleicher Leistung auch zunehmend billiger. Vor dem Hintergrund dieser Entwicklung untersucht das Forschungsgebiet des „Pervasive Computing“, wie Kleinstcomputer in unsere alltägliche Umwelt integriert werden können, um Anwendern in Alltagssituationen neuartige Dienste zur Verfügung zu stellen. Mittlerweile haben diese Kleinstcomputer sogar Formgrössen erreicht, die es erlauben, sie unaufdringlich – vom Standpunkt eines Anwenders scheinbar unsichtbar – in beinahe beliebige Dinge zu integrieren. Solche mit Informationstechnologie ausgestattete so genannte smarte Alltagsgegenstände haben die Fähigkeit, mit anderen smarten Dingen zu kommunizieren, sie können ihre Umgebung mittels Sensoren wahrnehmen und Anwendern kontextbewusste Dienste zur Verfügung stellen.

Allerdings können smarte Alltagsgegenstände aufgrund ihrer sehr ausgeprägten Ressourcenbeschränkungen sowie begrenzten Benutzerschnittstellen Anwendungen im Allgemeinen nicht autonom und ohne externe Unterstützung realisieren. Stattdessen ergibt sich für sie oft die Notwendigkeit, mit in ihre Umgebung integrierter Informationstechnik interagieren zu müssen, um auf entfernte Sensoren und externe Nutzerschnittstellen zugreifen zu können. Als Konsequenz kommt der Kooperation in smarten Umgebungen eine grosse Bedeutung zu, weil die Zusammenarbeit mit anderen Geräten smarten Dingen helfen kann, die ihnen inhärenten Beschränkungen zumindest teilweise zu kompensieren.

Die vorliegende Dissertation widmet sich dem Thema Kooperation in smarten Umgebungen unter besonderer Berücksichtigung ressourcenbeschränkter smarter Alltagsgegenstände. Der Hauptbeitrag dieser Arbeit ist es aufzuzeigen, wie smarte Dinge trotz der ihnen eigenen limitierten Ressourcen komplexe Dienste realisieren können, indem sie mit anderer in ihrer Umgebung integrierter Informationstechnik kooperieren. Dabei konzentriert sich die Arbeit auf zwei Teilaspekte der Kooperation mit smarten Alltagsgegenständen: (1) die Interaktion von smarten Dingen untereinander und (2) die Kooperation von smarten Alltagsgegenständen mit mobilen Nutzergeräten.

Der erste Teil dieser Arbeit befasst sich damit, wie um Informationstechnologie erweiterte Alltagsgegenstände Dienste in Zusammenarbeit mit anderen smarten Dingen realisieren können. Im vorgestellten Lösungsansatz erleichtert ein Programmiermodell die Erstellung kollaborativer kontextbewusster Dienste. Dabei werden miteinander interagierende smarte Dinge zu einer Gruppe zusammengefasst, deren Teilnehmer ihre Ressourcen bündeln und daher aus der Sicht eines Anwendungsprogrammierers als ein einziger, hinsichtlich Ressourcen mächtigerer Knoten erscheinen. Basierend auf diesem Grundansatz wird eine Softwarearchitektur zur Realisierung kooperativer Dienste auf smarten Alltagsgegenständen vorgestellt. Sie besteht aus (1) einer Beschreibungssprache für kontextbewusste Dienste, (2) einer Softwareschicht für die kollaborative Kontexterkennung,

(3) einer Infrastrukturschicht, die für die Verteilung von Daten zwischen kooperierenden Objekten verantwortlich ist und (4) einer Kommunikationsschicht, die Netzwerkparameter in Abhängigkeit von der momentanen Situation eines smarten Objektes in der realen Welt anpasst. Die vorgestellten Konzepte werden auf der Basis einer prototypischen Implementierung auf eingebetteten Sensorknoten, den BTnodes, evaluiert.

Der zweite Teil dieser Arbeit widmet sich der Kooperation von smarten Dingen mit mobilen Nutzergeräten im Hinblick darauf, wie um Informationstechnologie erweiterte Dinge von der Heterogenität in smarten Umgebungen – d.h. von der Präsenz verschiedenartigster Geräte mit unterschiedlichen Eigenschaften und Fähigkeiten – profitieren können. In diesem Zusammenhang besteht der wesentliche Beitrag dieser Arbeit darin, wiederkehrende Benutzungsmuster zu identifizieren, die aufzeigen, wie smarte Dinge die Fähigkeiten von Handhelds für ihre Zwecke nutzen können. Mobile Nutzergeräte können dabei unter verschiedenen Aspekten von smarten Alltagsgegenständen benutzt werden: (1) als mobiler Zugriffspunkt zu einer Hintergrundinfrastruktur, (2) als externes Benutzerinterface, (3) als externer Sensor, (4) als mobiler Datenspeicher, (5) als Bereitsteller von Rechenressourcen und (6) zur Identifikation von Anwendern. Diese Benutzungsmuster werden detailliert beschrieben, es wird auf Erfahrungen mit prototypischen Implementierungen eingegangen, und es werden verschiedene Anwendungsszenarien einschliesslich deren exemplarischer Realisierung vorgestellt, um die Eignung der vorgestellten Konzepte in smarten Umgebungen zu illustrieren.

Chapter 1

Introduction

1.1 Motivation

As pointed out by Weiser and Brown [WB96], Pervasive Computing[1] "is fundamentally characterized by the connection of things in the world with computation." Smart everyday objects exemplify this vision of Pervasive Computing because they link mundane everyday things with information technology by augmenting ordinary objects with small sensor-based computing platforms (cf. Fig. 1.1). A smart object can perceive its environment through sensors and communicates wirelessly with other objects in its vicinity. Given these capabilities, smart objects can collaboratively determine the situational context of nearby users and adapt application behavior accordingly. By providing such context-aware services, smart objects have the potential to revolutionize the way in which people deal with objects in their everyday environment.

However, the resource restrictions of smart objects and their limited user interfaces severely hinder their ability to realize useful applications. In fact, because of their inherent resource restrictions, which result from the necessity to keep the computing platforms of smart objects small and unobtrusive, it is difficult to tap the full potential of augmented items to the extent that would be desirable. As a result, in today's applications involving augmented items, services associated with them are often entirely realized by background infrastructure services. The only functionality provided by smart objects themselves is thereby to transmit an identity or raw sensory data to a nearby stationary server that realizes services on their behalf. With the proliferation of active sensor nodes that can be integrated into everyday things, however, smart objects can take more responsibility for realizing their own services.

In this dissertation, we argue that cooperation with nearby computing devices is a key concept for enabling smart objects to overcome their resource restrictions and hence to realize more sophisticated services. To support this argumentation, we deal with two types of cooperation that are of special importance in smart environments: (1) cooperation among different smart objects and (2) cooperation between smart objects and mobile user devices. We chose to investigate these two aspects of cooperation because they show how augmented items can exploit two key properties of Pervasive Computing environments – ubiquity of computation and heterogeneity. Both, heterogeneity and the abundance of computing devices present in smart environments allow smart objects

[1] Weiser and Brown used the term Ubiquitous Computing in the paper cited. The term Pervasive Computing was introduced later and made popular by companies such as IBM. Today, the two expressions are usually used interchangeably.

to rely on other nearby computing devices when carrying out computations or realizing interactions with users. Regarding cooperation between smart objects, ubiquity of computation implies that in smart environments many – albeit computationally weak – smart objects are likely to be in range of each other. Cooperation among these objects is therefore a suitable means to bundle their resources in order to realize more complex services. In contrast, the cooperation with handheld devices illustrates how smart objects can benefit from the heterogeneity of smart environments. Heterogeneity implies that smart environments are populated with computing devices with many different capabilities and various means to support interactions with users. Cooperation is therefore an important tool for smart objects to make use of other devices' capabilities and to integrate their features into their own applications.

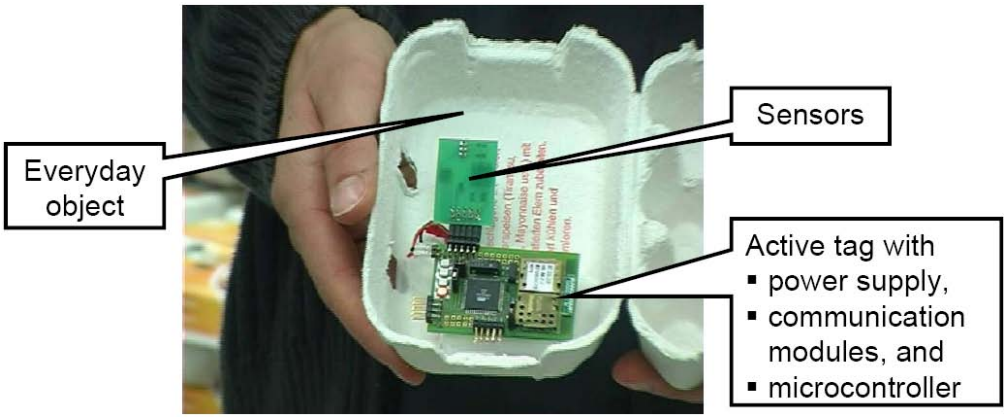

Figure 1.1: A smart object: an everyday item augmented with a sensor-based computing platform.

Cooperation with other computing devices in smart environments helps smart objects to implement their services for several reasons. The most prominent one is context-awareness. In smart environments, people expect applications to automatically adapt to their current situation, which requires smart objects to obtain information about their real-world environment and that of nearby users. The problem is that a single object can only perceive a small subset of its environment with its own local sensors, and therefore often needs to cooperate with other objects to more accurately derive context information. Another reason for cooperation is that the resources of smart objects are restricted as regards memory capacity and processing power. In order to carry out computationally expensive algorithms or to store large amounts of data, smart objects need to cooperate with other entities. Last but not least, the computing platforms attached to objects need to be as unobtrusive as possible when augmenting everyday things with computation. In fact, prospective users should not even be aware of the integrated technology, and it must not hinder the way in which people normally use their items. Consequently, smart objects often have no sophisticated input/output interfaces such as displays or keys. This means that when a smart object wants to provide services that rely on such interfaces, it needs to cooperate with other computing devices – handhelds or mobile phones for example – that provide such user interfaces.

1.2 Contributions

The main thesis of this dissertation is that cooperation with nearby computing devices helps smart objects to realize services for people in smart environments. We support this thesis in two ways. First, it is shown how nearby smart objects can bundle their resources and cooperatively implement context-aware services. Second, we focus on the cooperation between smart objects and nearby handheld devices as an example of how augmented items can interact with different kinds of computing devices and access their resources. As a result, the main contribution of this dissertation is to illustrate how smart objects can make up for their limited resources and restricted user interfaces by cooperating with other smart objects and mobile user devices; and to show how they can make use of the capabilities provided by other computing devices and integrate them into their own applications.

In the following, we list the core contributions of this dissertation in more detail:

- **Context-aware collaboration among smart everyday objects.** We have designed and built a software platform for realizing and executing context-aware services in collections of cooperating smart everyday objects. The platform consists of (1) a programming language for describing context-aware services, (2) a compiler for this language that maps a description of a context recognition process onto instructions for the context-layer of smart objects, and (3) a tuplespace-based infrastructure layer for distributing data among collaborating entities. Please refer to Chap. 3 for a detailed description of these concepts.

- **Context-aware communication services.** We call communication services context-aware if they adapt networking parameters according to the situational context of smart everyday objects. Our argumentation for context-aware communication services is as follows: Smart objects usually provide context-aware applications, which adapt themselves according to the real-world situation of nearby users and that of smart objects. Consequently, there is a narrowing gap between virtual and real worlds, i.e., what happens to a smart object in the real world (e.g., where it is, whether it is used, etc.) has an increasing effect on its communication in the virtual world. Therefore, we suggest considering the real-world environment of smart objects in networking services in order to construct networking structures that are more suitable for context-aware applications. This idea and corresponding examples are presented in more detail in Sect. 3.6.

- **The value of handhelds for smart everyday objects.** In this dissertation, we present several usage patterns that describe how smart objects can make use of nearby handheld devices. These usage patterns are: (1) mobile infrastructure access point, (2) user interface, (3) remote sensor, (4) mobile storage medium, (5) remote resource provider, and (6) weak user identifier. In this respect, our main contributions are that having identified these usage patterns, we give directions for their realization, report on our experiences with a concrete implementation on an embedded device platform, and show their applicability by means of several example scenarios (cf. Chap. 4).

- **Smoblets – integrating handheld devices into environments of cooperating smart objects.** The purpose of the Smoblet system is to enable smart

objects to dynamically access the computational resources and user interfaces of handheld devices such as PDAs (Personal Digital Assistants) and mobile phones. The handhelds thereby participate in a distributed data structure shared by cooperating objects, which makes the location where code is executed transparent for collaborating entities. This enables smart objects to outsource code to handhelds that has previously been stored on smart objects for this purpose. This code is then executed on the computationally more powerful handheld, can utilize its user interfaces, and has access to the data of smart objects by means of the shared data space. Besides introducing and evaluating the underlying concepts, our contribution includes a prototypical implementation on an embedded device platform. Sect. 4.5 contains more details about this.

- **Embedding smart everyday objects into the everyday communication infrastructure.** Smart objects not only need to cooperate with each other and nearby handheld devices, but also with remote services that must be accessed over a background infrastructure. For example, it is desirable for smart objects to access data on a distant backend server, or to make services available to remote users. Interaction with remote objects, remote users, and backend services requires smart objects to be embedded into existing communication infrastructures such as the Internet or mobile phone networks. In this dissertation, we present ways in which augmented artifacts can communicate with remote servers and how they can provide services to distant users by means of such everyday communication infrastructures as cellular phone networks and the Internet. For more information about this see Sect. 4.4.1 and 4.4.2.

- **Translating established interaction patterns to smart everyday objects.** Well-established interaction patterns such as making phone calls, writing SMS messages, or handling calendar entries on a PDA have the advantage of being widely accepted by a large number of users. Translating these interaction patterns to smart everyday objects might therefore be an opportunity to make it easier for people to deal with this new type of technology. This dissertation presents ideas that illustrate how such an approach can be realized. Our approach was to assign phone numbers to smart everyday objects in order to enable users to "call" their personal items or to communicate with them by means of SMS messages. We have also investigated forms of implicit user interactions that aim at making the handling of smart objects easier. More information on this topic can be found in Sect. 4.4.2, Sect. 4.6.1, and Sect. 4.6.2.

- **Applications.** We have prototypically implemented several application scenarios to illustrate the applicability as well as the limitations of our concepts. The applications include several "smart" things: a smart egg carton, a smart medicine cabinet, an object tracking and inventory monitoring application, and a smart training assistant (cf. Sect. 4.6).

1.3 Organization

The rest of this dissertation is organized as follows: Chap. 2 provides background information. First, it discusses the foundations of Pervasive Computing, and considers

the trends that speak for and against the vision of omnipresent information technology embedded into our everyday environments. Second, we show how objects can be made "smart" and present a survey of state-of-the-art tagging technologies. Third, the introductory chapter deals with the notion of *context* and gives an overview of approaches and systems for deriving context in smart environments. Context is an important concept in the scope of this dissertation because context recognition is one of the core reasons for smart objects to cooperate with each other.

Chap. 3 focuses on the cooperation between smart objects. Its main purpose is to show how smart objects can come to terms with their resource restrictions by realizing services in cooperation with other augmented items in the vicinity. The chapter starts with a short rationale and an overview of our proposed solution. Then, a description language is presented that helps programmers in realizing collaborative context-aware services. The chapter then introduces a software architecture for augmented artifacts that allows them to execute context-aware services involving multiple objects. This software architecture consists of a context layer, a layer facilitating the data exchange among cooperating objects, and a communication layer. The remainder of the chapter describes these in greater detail.

Chap. 4 deals with the cooperation between smart objects and mobile user devices. Its main purpose it to show how smart objects can dynamically access the resources of nearby handheld devices. After discussing the underlying problem and presenting an overview of the proposed solution, the chapter identifies usage patterns which illustrate how smart objects can actually make use of handheld devices. The chapter then describes these usage patterns in more detail and reports on our experiences with implementations on an embedded sensor node platform. Afterwards, six application scenarios and corresponding prototypical implementations are presented that illustrate the suitability of the proposed concepts. The application scenarios also show how different usage patterns can be combined in a single application.

Chap. 5 concludes this dissertation by summarizing its main contributions. It discusses the limitations of our work as well as the main lessons learnt, and gives directions for future work.

Chapter 2

Background and Related Work

This chapter presents background information on Pervasive Computing and smart everyday objects, and reviews related work carried out by other researchers. We start with a look at the research field of Pervasive Computing, its motivation and underlying vision, and try to assess whether it is reasonable to believe that it will ever fulfil the expectations associated with it. Second, we deal with the term *smart everyday object* and give an overview of today's tagging technology, which can be used to integrate computation into everyday things. Third, the chapter reviews work in the field of collaborative context recognition. The term *context* is of central importance for this dissertation, because context recognition is one of the main reasons for smart objects to interact with each other. We review how other researchers deal with the topic of deriving context in smart environments, and show that their approaches are mainly based on centralized systems or powerful computers in a backend infrastructure; smart objects often merely provide raw sensory data without actually being involved in the process of combining sensory data from multiple sources.

2.1 The Vision of Pervasive Computing

Pervasive Computing envisions a world of omnipresent information technology in which computation is seamlessly integrated into the environment in order to support people in their everyday activities. Enriching everyday surroundings with information technology is thereby done in such a way that computers become omnipresent but at the same time also invisible for prospective users. Invisible computers are meant to disappear into the background, and to become such integral parts of the environment that people might not even be able to identify the abundant information technology surrounding them.

A paramount motivation behind this vision of Pervasive Computing is the observation that general-purpose computers such as laptops and PCs are not well suited to naturally supporting people in their daily activities. Instead, working with laptops and PCs is very technology-centered: a substantial initial overhead involved in learning to use computers, their high complexity, and unintuitive usage patterns make people adapt to what a computer program expects of them and not the other way around. What makes Pervasive Computing different is that it is inherently human-centered. Computers embedded into the everyday environment are supposed to sense the context of people and adapt their application behavior automatically based on a user's current surroundings. This means that information technology becomes unobtrusive and interaction with smart environments implicit. Consequently, Pervasive Computing aims at making it easier for people

to benefit from information technology, and requires significantly less previous knowledge and fewer computer skills.

First steps towards realizing the vision of Pervasive Computing have been the development of information appliances such as tabs, pads, and board-sized writing and display surfaces [Gel04, Wei91, Wei93]. In contrast to general-purpose computers, information appliances are tailored towards the needs of a specific application domain, such as telephony or data communication. Consequently, they are better suited to supporting people in a restricted set of activities, easier to use than general-purpose computers, and are often also smaller and lighter because they only have to support a restricted set of functions. Prominent examples of information appliances are today's cellular phones, which already have a much greater market penetration than PCs. Paramount reasons for this development are that mobile phones as an information appliance are easy to use, small, designed for a specific purpose, and handy, and are therefore better accepted by users.

But information appliances are not the final stage of the broader vision of Pervasive Computing. As pointed out by Mark Weiser – an influential pioneer in the research field of Pervasive Computing – "prototype tabs, pads, and boards are just the beginning of Ubiquitous Computing" [Wei91]. Recent research therefore goes one step further and strives to enrich mundane everyday objects with information technology. Given recent advances in hardware and communication technologies, mainly in the area of miniaturization and wireless communication, such an approach seems to have become more and more feasible. Also, many application scenarios – for example in the medical or insurance sector – suggest that there are actually many sensible cases for the use of pervasive technology. On the other hand, there are still many obstacles to overcome and open issues that have not yet been solved. The question, for example, of whether the social implications of pervasive technology regarding personal privacy will be accepted, or technological issues such as how to provide a sustainable energy supply for smart objects, remain largely open. As a result, the question of whether the vision of Pervasive Computing can ever be truly realized remains a matter of speculation. In the following, we briefly discuss some of the most dominant issues which are currently influencing the development towards the vision of Pervasive Computing, in order to shed light on this question.

2.1.1 Technological Foundations

This section discusses the technological foundations of Pervasive Computing. In this respect, Mattern [Mat01, Mat02, Mat04] distinguishes between the following technological "reasons" for the development towards a world of computer-augmented everyday artifacts: (1) Moore's law [Moo65, RCN02], (2) new materials, (3) progress in communication technologies, and (4) better sensors. However, regarding smart everyday objects that are augmented with active sensor nodes, there are also considerable technological problems which make the deployment of augmented artifacts difficult. As an example, we describe the problem of finding a sustainable energy supply for smart objects, and discuss current advances in energy scavenging and harvesting.

Moore's Law

In 1965, Gordon E. Moore observed that the number of components per integrated circuit increases at an exponential rate, doubling approximately every year [Moo65]. Although

he predicted that this trend was likely to subsist over the next decade – i.e., until 1975 – his observation of exponential growth, admittedly at a slightly slower speed, has proven to be valid for almost 40 years now. This development has led to the formulation of Moore's law, which predicts that the effectiveness of semiconductor technology doubles roughly every 18 months. It is believed by experts in the semiconductor industry that Moore's law will continue to hold true at least until the end of this decade [Moo04]. Moore's law not only applies to the number of transistors per integrated circuit, but in addition the storage capacity of discs, the bandwidth of communication networks and computation speed are increasing at an exponential rate, doubling every one to three years [Mat02]. As a result, computer hardware is becoming smaller, faster and also cheaper, which is a precondition for integrating large amounts of computing devices into everyday environments.

New Materials

Embedding computation into everyday things not only requires computers that are small, fast, and cheap, but also calls for new mechanisms to integrate computation into our surroundings and new means to interact with information technology. Whereas Moore's law predicts that it will be possible to integrate standard processors and communication chips into things, new materials open up entirely new ways for integrating computation into everyday objects.

A prominent example of such new materials are semiconducting organic polymers [Moo02], which might one day enable companies to print electronic circuits, to produce electronic paper, or to develop novel forms of graphic displays. New materials also help to make computation less obtrusive and more pervasive. Electronic paper, for example, combines the acceptance of conventional paper with the advantages of information technologies: the same sheet of paper can be reused several times, text passages can be easily edited, and smart pens can store entire documents. Smart paper also epitomizes one important aspect of smart everyday objects: It is not the goal of smart things to change interaction patterns that have proven to be useful. Instead, smart everyday objects strive to improve the way in which users deal with everyday things by providing additional services that enrich existing interaction patterns.

Many of these new technologies such as smart paper and plastic displays are still in their infancy. However, several prototypes underline their potential in the Pervasive Computing domain [Xer04a, Xer04b].

Communication Technologies

Over the past years, the bandwidth of communication technologies has increased at an exponential rate. This is true for both wired and wireless networks. Whereas wired networks constitute the backbone of today's Internet with data rates of up to several Gb/s, wireless communication technologies are tailored towards the needs of mobile computing systems and offer typical data rates in the range of several kb/s up to a few Mb/s. Efficient wireless communication is therefore indispensable for integrating computation into mobile everyday objects, and a powerful backbone is necessary to enable smart objects to communicate with background infrastructure services. In fact, wireless communication is the basic foundation for any cooperation between augmented items and their interaction with surrounding background infrastructure services or mobile user devices. In the scope

of this dissertation, it is therefore important that the protocols for cooperation between smart everyday objects can be mapped onto efficient networking structures (cf. Sect. 3.6).

There is a wide range of wireless communication technologies that are of potential interest in Pervasive Computing settings, e.g., IrDA, Bluetooth, Zigbee, WLAN, GSM, UMTS, custom RF protocols, optical communication, RFID protocols, powerline communications, or body area networks. Because optical communication and infrared-based technologies require line-of-sight for communication, they are not always suitable for augmenting everyday objects. Especially if manual alignment is necessary in order to enable devices to communicate with each other, such technologies are not adequate for augmenting everyday things because they require continuous user attention. In contrast, radio technologies do not have this drawback as radio waves penetrate clothes and walls. Radio technologies, on the other hand, can require substantially more power, especially when communication needs to take place over long distances. As a result, the energy restrictions of smart objects often render long-range wireless radio technologies – such as GSM, or UMTS – unusable for augmenting everyday things.

In order to save energy, smart objects, mobile user devices, or low-power sensor nodes are sometimes equipped with two different communication technologies. Of these technologies, one is highly energy-efficient but relatively slow and rather unreliable, whereas the other consumes more energy but transmits at high data rates. In this case, the low-power communication chip listens for incoming requests while the high-quality communication module is switched off. The high-quality module is only turned on when large amounts of data have to be transmitted in a small time frame. This idea has been successfully applied for saving energy on PDAs [SBS02]. Similar approaches are used in low-power radio design [RAK+02] for wireless sensor nodes; the third revision of the BTnode sensor node platform is also equipped with two communication technologies to decrease the energy consumption of communications [Btn04].

Sensors

Sensors are the basic foundation for any context-adaptive behavior in smart environments. In order to provide context-aware services, smart everyday objects must be able to perceive their environment and to share their observations with other objects. Sensors therefore need to be so small that they can be embedded into everyday objects, they must be so cheap that they can be used in large quantities, and they must be energy-efficient. Fortunately, recent advances in sensing technologies and the overall trend described by Moore's law make it seem likely that all these demands can be met in the not so distant future.

The rapid development of sensing technologies can best be illustrated considering the research field of wireless sensor networks, where small-scale wireless sensor nodes are used in application areas such as ecosystem monitoring and contaminant transport. The main challenge in these applications is to deploy autonomous wireless sensor nodes to enable highly accurate, in-situ monitoring of environmental parameters. The use of small sensor nodes in the above mentioned application domains has several advantages regarding price and usability. Considering the monitoring of vibrations in buildings as an example, Pscovitz [Pes04] and Rabaey [Rab03] report that traditional sensor nodes used in this application domain cost about USD 8000, were relatively large, and connected by wires. As a result, such sensor nodes were only used sporadically, which reduced the

accuracy of measurements. Today, experiments are under way using sensor nodes that are around 100 times cheaper (about USD 80), communicate wirelessly, and can therefore be utilized in larger quantities to produce more accurate results. Although current prices of around USD 80 are still too expensive, if prices continue to fall at the present rate it could become feasible to integrate such sensor nodes into everyday things in a few years time. In summary, there is a trend towards smaller and cheaper sensing equipment and wirelessly communicating sensor nodes that makes it at least technically feasible to embed such technology into everyday things.

Advances in sensing technologies have also made sensors more energy efficient. There are, for example, temperature sensors that can harvest the entire energy needed for their operation from the surrounding environment. On the other hand, relatively complex sensors such as cameras and fingerprint sensors have been made so small that they can be embedded into mobile phones and PDAs. Smart everyday objects can then benefit from such sensors by cooperating with the devices carrying them (cf. Chap. 4).

Energy Supply for Smart Objects

A major concern in developing ubiquitous networks of wirelessly communicating smart everyday objects is energy consumption. This is because the energy consumption of processors, communication modules, and sensors – despite the previously mentioned technological advances in data processing, communication, and sensing technologies – has unfortunately remained relatively high. However, smart everyday objects must be supplied with energy from an energy source that lasts all their life, because it is not reasonable to expect users to continuously change the batteries of all the computing devices embedded into smart environments. Basically, there are four methods to approach the energy problem: (1) higher energy capacity of conventional batteries, (2) alternatives to conventional batteries such as fuel cells or micro heat engines, (3) energy scavenging and harvesting, and (4) external power supply through electromagnetic fields emitted by special reader devices.

The energy capacity of conventional batteries is unfortunately not increasing at the same rate as processing power and bandwidth. It is therefore not experiencing the kind of development predicted by Moore's law. According to [ECPS02], the energy capacity of AA nickel alkaline batteries increased from 0.4 Ah to a mere 1.2 Ah in 20 years. As a result, recent research has concentrated on more enduring alternative energy supplies such as small fuel cells or micro heat engines [MZA+00].

However, these technologies still require manual recharging, which contradicts Pervasive Computing's vision of invisible and unobtrusive computation embedded into our environments. A possible solution to the problem of manual recharging is to harvest energy from the surrounding environment or to scavenge ambient power resources [ECPS02, RWR03]. This would theoretically enable the lifetime of a smart object to become infinite. Potential sources of ambient power are daylight, vibrations, acoustic noise, and temperature gradients. There are also technologies that can scavenge power from airflows or body movements [KKPG98, MM71, Sta03].

Another approach to solving the energy problem is that of passive tagging technologies, such as passive RFID labels or visual codes, which can be used to augment everyday objects computationally. In this solution, passive tags usually provide a simple identification of the object they are attached to, which is associated with a service in the surrounding computing infrastructure. Passive RFID tags are externally powered, i.e.,

they receive the energy needed for their operation from an electromagnetic field issued by a special reader device. In contrast, visual codes such as standard barcodes, which are present on virtually every product in a supermarket, do not carry out any computation on their own, but provide identification through their visual structure. A barcode scanner or a camera is needed to transform the visual layout of the tag into identifying information. Barcode scanners and RFID reader devices are often stationary and can be powered by the standard power supply system. Smart objects with a passive tagging system do not need a local power supply.

2.1.2 Economic Developments

Emerging commercial applications are a major driver for Pervasive Computing. Today, these applications primarily deal with passive RFID labels for remote identification. The former Auto-ID center [Aut04], a consortium including both leading research institutions and companies, actively promoted the usage of RFID labels for tagging items such as the products in a supermarket. Its broader goal was to design "an open standard architecture for creating a seamless global network of physical objects" and an "Internet of things" [Aut04]. Although the Auto-ID center is now closed, it has been superseded by the Auto-ID labs [Aut04], a research oriented network of six universities at the time of writing, and EPCglobal [EPC04], which works towards globally accepted standards and the commercial adoption of the work initiated by the Auto-ID center.

Potential application domains for RFID technology are manifold. They range from healthcare [Bro02], control systems [McF02] and manufacturing [CGS+03] to the automotive industry [SF03b]. Regarding healthcare, RFID systems can improve patient compliance by monitoring the medicine a patient is taking, they can be used for tracking medical instruments in a hospital, or for the validation of drugs [McF02]. The best-researched application field for ubiquitous tagging, however, is supply chain management [ZWA+02]. This is because RFID labels can provide accurate and timely information about a tagged product. For example, RFID tags make it possible for a company to find out precisely where their products are at any given point in time: they can provide information about inventory levels, the order status of a product, or delivery times. They can even help retailers to detect when a product is being stolen. All this enables companies to adapt dynamically and respond in a timely way to unpredictable events affecting their supply chain. The potential cost reduction of such an approach is immense.

The above scenarios mainly focus on passive RFID labels because they are externally powered and relatively cheap (the current goal is to produce RFID tags at a price of 5 cents per piece). However, although the tags themselves are cheap, the external reader devices necessary to read out the tags are costly. It is also difficult to tag items that are made of metal or contain fluids, such as coke cans and water bottles. In such cases, active RFID tags, which have an autonomous power supply, can be used used to identify items. Furthermore, pallets and boxes, which contain larger quantities of objects, are often augmented with tags instead of individual items. Active tags can also be equipped with sensors to monitor the state of products continuously during transport [KH01, Par04]. Other potential application areas of active sensor tags, beside remote identification, include location-aware services, the networking of smart objects over the Internet, and new kinds of interaction patterns with smart things that make it easier for people to handle their personal belongings [VMKA03].

2.1.3 Privacy

Although Pervasive Computing technologies are extremely versatile, enabling a wide range of new applications, they are also a source of major concerns regarding personal privacy. Besides energy consumption, threats to privacy are probably the main impediment to realizing the vision of Pervasive Computing. The privacy problem stems from an intrinsic property of Pervasive Computing called *invisible* or *implicit* human-computer interaction. In this respect, Weiser and Brown [WB96] have also coined the term *calm technology*, meaning that computation disappears into the background, anticipates user needs, and interacts with people in a non-intrusive way. Some researchers also speak of *invisible computing* [EHAB99].

As mentioned above, the capacity to adapt automatically is a desirable property of Pervasive Computing environments. This is because as computation becomes pervasive, explicit interactions, as with today's PCs, become impractical. Imagine hundreds or thousands of computers embedded into our daily surroundings that would try to "share" one user and would continuously require explicit actions from nearby people to tell them what to do. Imagine the hundreds or thousands of different actions that would be necessary to interact with these computers explicitly. Calm technology, in contrast, tries to put computers in the background where they only disturb users when the current situation requires it.

Calm computing is a potential threat to privacy because computers must be aware of a user's situation in order to react automatically according to this situation. In other words, computers need to know something about us and our environment in order to adapt automatically to our current needs; but often this is something that people feel only they should know. The fact that computers can store such information for unpredictable time periods and make it accessible via global networks further aggravates privacy concerns.

A good example of the privacy problems caused by Pervasive Computing is context-aware profile selection for mobile phones. To allow this profile selection, mobile phones are equipped with sensors that can derive the current situation of a person in cooperation with other nearby computing devices. As a result, the mobile phone can then automatically choose an appropriate phone profile based on this context information. For example, when the phone derives the context *meeting*, no calls are forwarded to the mobile terminal, or at least no acoustical ring tone is issued. Another approach is to determine the current context of a user and to share this information with prospective callers by means of a backend infrastructure server [STM00]. Callers can then access information about the situation of the person they want to contact, and decide themselves whether they really want to make the call or not. Here, the problem with privacy stems from the fact that the user can lose control over personal data: as the mobile phone cooperates with other computing devices to derive context information and possibly exchanges this information with a background infrastructure, it becomes more difficult to ensure that no unauthorized party can access personal data. Furthermore, by allowing the mobile phone to act automatically on behalf of a user, he or she might not even be aware of what the phone is actually doing and which state it is in. In general, giving autonomy to computers also means that they could do things we do not want and which we cannot control.

Collaboration among autonomous smart objects, which is the focus of this dissertation, can considerably aggravate the problem of privacy. Considering the previous example of context-aware profile selection, if the mobile phone cooperates with its en-

vironment to determine the user's current context, this might imply that not only the mobile phone itself knows the current situation of its owner, but also other smart objects in the room. Hence, nearby smart objects could analyze that data and share the results with third parties and thereby distribute personal information about a user. In collaborative environments, therefore, all participating objects must accept, conform to, and act in accordance with a user's privacy settings. Ensuring privacy therefore becomes more complex, especially in dynamic environments where devices know only little about each other and do not have access to background infrastructure services to obtain information about other entities from a reliable source.

Recent suggestions to approaching the privacy problem could not be more diverse. Some people think that increased privacy concerns can be ignored. A famous example of this point of view is a statement by Scott McNealy, a co-founder of Sun Microsystems, who told reporters: "We have zero privacy anyway – get over it." In an opinion article for the Washington Post [McN03], McNealy also warns against exaggerated privacy concerns: "Privacy is not always desirable – and absolute privacy is a disaster waiting to happen." Other researchers take the problem more seriously [Lan01]. A possible approach to dealing with privacy issues is by means of anonymization [KS03] or personal profiles that specify what kind of data people are willing to make publicly available in certain situations [Lan02]. The latter approach has its roots in the Platform for Privacy Preferences Project (P3P) of the World Wide Web Consortium, where it was initially designed for applications such as online shopping. Whether the relatively complex XML-based protocols used in the Internet are suitable for resource-restricted smart everyday objects remains open to question. Personal profiles can also cause increased annoyances if people must manually allow smart objects to collect data. With many smart objects in the vicinity, this can contradict the idea of calm technology. However, the use of personal profiles restricting the exchange of personalized data with smart environments is a promising approach to tackling privacy issues. At the same time, efforts must be made to ensure that the amount of user input necessary to negotiate privacy settings is kept to a minimum; the settings specified in a user's personal profile should be sufficient to carry out negotiations automatically in most cases [Lan02].

Another approach to alleviating privacy concerns in connection with smart objects is to deploy sensors that make it difficult to derive a person's identity. For example, there are digital cameras that blur the faces of people in recordings. If a smart object shares only such unpersonalized data with its environment, it reduces the probability of privacy intrusions.

2.2 Smart Everyday Objects

As pointed out by Weiser and Brown [WB96], Pervasive Computing can be "fundamentally characterized by the connection of things in the real world with computation." Smart everyday objects epitomize this vision of Pervasive Computing in that they combine ordinary everyday things and information technology. They are therefore a central concept in the research field of Pervasive Computing. In the following, we try to contribute to a deeper understanding of the term *smart everyday object*, review means to link computation with everyday things, and give an overview of current tagging technologies.

2.2.1 What is a Smart Everyday Object?

A smart everyday object is an arbitrary item from our everyday environment – such as a chair, a hammer, a car, or an umbrella – augmented with information technology. Such augmentation must meet these requirements:

- **Unobtrusiveness.** The computation embedded into an object must not distract people from carrying out the task originally associated with it. The information technology should be either embedded in the design of an object or completely invisible – and not just a bulky add-on getting in the user's way.

- **Integrity.** People using a smart everyday object should perceive it as a single consistent unit. There should be a clear connection between the original purpose of the everyday object and the additional services provided by the embedded technology.

Information technology can either be directly integrated into the object itself, be available in a supporting background infrastructure, or both. If computation is provided by the environment rather than by the object itself, there must be a mechanism to link the object to a corresponding background infrastructure service that processes data and communicates with peers on behalf of the object.

The terms *smart object* and *smart everyday object* are used interchangeably throughout this thesis. However, considering definitions from the literature, there are significant differences between these two terms. Kintzig et al. [KPPF03], for example, defines a smart object as a "physical device equipped with a processor, memory, at least one network connection, and various sensors/actuators, some of which may correspond to human interfaces." According to their definition, electronic tags and wireless-enabled PDAs are all smart objects in their own right. This constitutes a major difference to our understanding of smart *everyday* objects: A smart everyday object always consists of both an everyday thing and information technology that augments it. PDAs and mobile phones themselves are therefore not smart everyday objects.

In the work presented in this thesis we concentrate on everyday things that are augmented by active sensor-based computing platforms. Such smart objects are able to perceive their environment through sensors, can carry out local computations, and can collaborate with other objects in their vicinity by means of wireless communication technologies. Furthermore, they possess an autonomous power supply and hence do not rely on external reader devices to supply energy.

Besides sensor-based computing platforms there are several alternative ways to link things with computation. Achieving this linkage usually requires some sort of electronic tag that is attached to ordinary objects. In the next section we present an overview of such tagging technologies, review how related projects augment everyday objects, and compare their work to the tagging approach underlying our work.

2.2.2 Augmenting Everyday Objects

Considering recent research, there are three main approaches to adding computation to everyday objects:

- **Referencing.** The information technology integrated into an everyday object provides a unique identification to external reader devices, which is used to address a

representative in the computing infrastructure. This representative processes data and implements interactions with users on behalf of the augmented object. Apart from the provision of a unique identification, the computing devices integrated into everyday things do not contribute to the smart behavior of an augmented item.

- **Physical integration.** Sensor-based computing platforms are directly integrated into everyday objects. They provide an object with the ability to sense its environment, to autonomously communicate with peers, and to process data locally. Smart objects do not rely on background infrastructure services, but can instead realize services on their own and in cooperation with other smart objects. They do not usually depend on external reader devices because they are often equipped with an autonomous power supply.

- **Hybrid approaches.** The technology embedded into an object can sense its environment and process inputs autonomously, but also collaborates with backend services when infrastructure access is available. A unique identification is used to address a background infrastructure service that helps an object to realize applications. Hence, both smart object and background service substantially contribute to the smart behavior of an augmented artifact.

Referencing

In the referencing approach, the only functionality of an augmented object is to provide a unique identification to a computing infrastructure. The identification of an augmented item is usually retrieved by external reader or scanner devices, and used to address a service in the backend infrastructure – the smart object's *data shadow* [Bei02] or *virtual counterpart* [RMDS02]. This infrastructure service carries out computations on behalf of augmented artifacts and communicates with virtual counterparts of other smart objects. Hence, the referencing approach merely links everyday objects with a representation in the virtual world; a smart object does not itself possess the computing resources necessary to interact directly with other objects or users.

The core advantage of the referencing approach is that the technology embedded into or attached to an object can be kept relatively simple because it only needs to provide an identification. The tags used for this purpose are therefore relatively cheap and small, which facilitates unobtrusive augmentation of everyday things. Furthermore, tags providing only identification often require no batteries, but are powered by external reader devices. The energy supply does not constitute a severe problem in these cases.

On the other hand, the referencing approach often depends on external reader devices that process IDs and establish the link between objects and their virtual representations. If there is no adequate infrastructure of such reader devices, the referencing approach becomes ineffective. Furthermore, as smart objects only emit an identification code, it is not so easy to infer other more complex information about an object, which in turn reduces the amount of potential applications. In fact, further information about an object can only be provided by external reader devices. This works well for location information because the location of a reader device approximates the location of a scanned tag. But deriving more sophisticated contextual information about an object, such as its moving pattern, how it is used, or its current status often requires sensors on the object itself. Another problem with respect to the referencing approach is obsolete and inaccurate information about tagged items. In the previous example of location estimation, an

object's position can be accurately determined when there is a reader device nearby. But what happens if the object moves to a place where no reader device is in range? Also, as the range of reader devices increases, it becomes more difficult to assure that physical parameters measured at the reader are also valid for a scanned object. With respect to collaboration, in the referencing approach smart objects do not directly cooperate with each other, but their virtual representations. This results in an additional layer of indirection, making it necessary for the reader devices to relay all the information related to an object to its virtual counterpart. A problem here is that counterparts can operate on old and obsolete data if the objects are not frequently scanned.

In summary, the applicability of the reference approach depends to a large extent on a dense infrastructure of tag readers. As readers are relatively expensive (the price of a mid-range RFID tag reader is in the range of several hundred US dollars), it remains uncertain whether such an infrastructure is feasible in everyday environments. The price of reader devices, however, largely depends on the concrete tagging technology used to identify objects. Using optical markers, for example, the price of reader devices can be much lower than in the case of RFID (cf. Sect. 2.2.3). As smart objects only provide an identification code in the referencing approach, more detailed information about an augmented item, such as its location or temperature, is inferred from the location of, or the temperature at, a tag reader. This introduces an additional level of indirection and can lead to inaccuracies. Furthermore, there is no direct collaboration between smart objects, but only between their virtual representations.

Physical integration

The approach of physical integration attempts to embed sensors, communication, and processing power directly into everyday objects. This allows smart objects to become more independent of backend infrastructure services and external reader devices. Furthermore, the augmented artifacts can directly collaborate with other objects in their vicinity, exchange information, and autonomously process data. Because sensors are directly integrated into everyday objects, they can also better monitor the state of an object, and can derive more accurate information about its environment. Derived information about smart objects reflects their current status because sensory data can be accessed continuously and not only in the presence of external reader devices.

Physical integration leaves the vast majority of tasks involved in implementing an application to the smart object itself, instead of transferring them to a background infrastructure representative. Smart objects can therefore also provide their services even when no infrastructure access is currently available. However, a disadvantage is that the sensor-based computing platforms integrated into everyday objects typically rely on some form of active power supply – e.g., a battery or a mechanism to harvest or scavenge energy from the environment. Furthermore, the tags attached to everyday objects or the type of technology integrated into them becomes more complex, which makes the tags bigger, more expensive, and more difficult to hide them from users.

Hybrid approaches

Smart objects that only provide identification to external reader devices leave all data processing and issues related to interaction with users to the backend infrastructure, whereas in the the physical integration approach a smart object processes data autonomously and

in cooperation with other augmented items. However, even if a sensor-based computing platform is directly integrated into an everyday thing, it can benefit substantially from background services when infrastructure access is possible. Smart objects can therefore use nearby access points to interact with more powerful infrastructure services that help them to process and store data. This makes it possible for an augmented item to collaborate with other smart objects that are not in its physical communication range but can be reached through the backend. Also, parts of applications realized by augmented items can be outsourced to services that represent smart objects in the infrastructure. It thus becomes possible for these backend services to take over responsibility for realizing applications for smart objects and for handling user interactions. This is, however, only necessary if a user cannot access an object directly; if direct access is possible people interact with smart objects and not their virtual representations (see Sect. 4.4.1).

2.2.3 Tagging Technology

In this section, we will give an overview of technologies available for augmenting everyday objects. We call this kind of technology, that aims at enriching everyday things with computation, "tagging technology," and refer to the corresponding computing devices as "tags." The use of the term *tag* stems from the fact that the augmentation of everyday objects today usually takes place in a post-hoc fashion: i.e., information processing capabilities are not incorporated into an everyday object during its design process, but instead attached to it later on. This has purely practical reasons. If smart everyday objects are to be used on a larger scale, it would of course be desirable to embed the technology into everyday things during the production process.

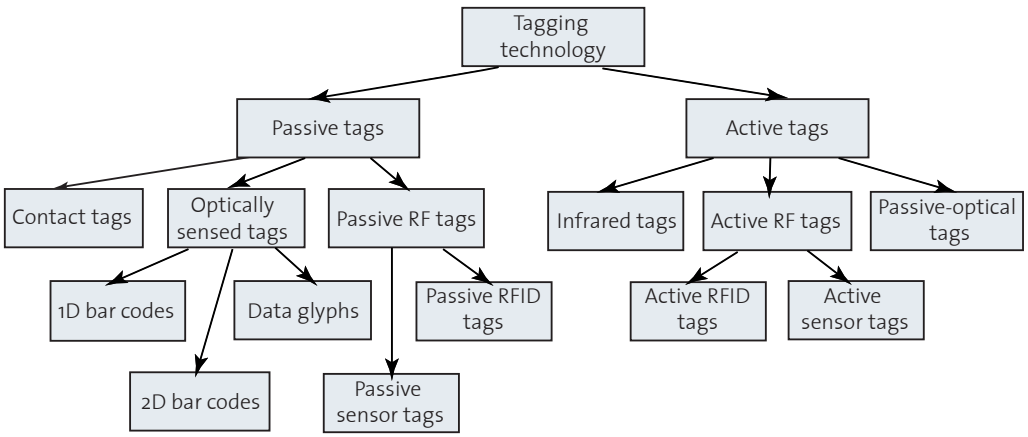

Figure 2.1: Overview of tagging technologies.

Fig. 2.1 depicts a classification of tagging technologies. At the highest level, we distinguish between active and passive tags. Active tags have an integrated power supply, which allows them to carry out computations autonomously without depending on external devices to supply the tag with energy. Active tags therefore have some kind of battery, can harvest energy, or scavenge ambient energy resources. As opposed to passive tags, they do not depend on external reader or scanner devices to access their data, but can broadcast data into their environment and collaborate with other augmented artifacts in the vicinity whenever they deem it appropriate. Furthermore, passive tags are usually much simpler than active tags. In the case of bar codes, for example, passive tags

do not even have a processor or other processing capabilities and therefore do not need energy for their own operation.

In the following, we present different passive and active tagging systems in more detail.

Passive Tags

There are three main groups of passive tagging systems: visible graphical tags (also called optically sensed tags [BK01]), passive RFID tags, and contact tags. The main application area of passive tags is to link an everyday object with its virtual representation in a background infrastructure – called the *virtual counterpart*, *data shadow*, *virtual entity*, or *digital counterpart* of a smart object. The identification stored on a tag is used to establish a connection to this representation. The purpose of the virtual representation is to provide additional functionality tailored towards the specific properties of an everyday object. A simple example is that the virtual representation of a book could open a Web page with customer comments about it when a passive tag attached to the book is scanned. Another application scenario is to attach digital content to physical objects. Here, the digital content is associated with an item and can be retrieved by a user in other physical environments. A user could, for example, attach information about other works by the same author to the tagged book from the previous example. Independently of the user's local environment (e.g., whether she is in a library or at home) she can then use the tagged book to access the associated information by scanning its tag. The corresponding information could then, for example, be displayed in a Web browser on her mobile phone or PDA. By using passive tags that simply store an identification code, it also becomes possible to translate interaction patterns from the virtual world – such as the predominant *copy and paste* – to physical environments. Barton and Kindberg [BK01] call this approach *copy and paste in the real world*. Here, digital content is associated with tagged items, which are carried to other places where the associated information can be "copied" to other objects. As passive tags can only store a simple identification, all data associated with an object is actually handled by its virtual representation. However, this is transparent for a user; from a user perspective it seems as if the tagged object itself were providing services on its own.

Visible Graphical Tags. Visible graphical tags can be classified into 1D bar codes, 2D bar codes, and glyphs, which are described in more detail in the following subsections.

1D Bar Codes. Optically sensed tags can be broadly classified into bar codes and glyphs (cf. Fig. 2.1). The simplest form of bar codes, the so called 1D bar codes, can be found on almost every product in today's supermarkets, where they are used for the mapping between products and prices at the checkout counter. Such bar codes contain a standardized Universal Product Code (UPC) identification number. Standard 1D bar codes usually require special scanner or reader devices that transform the bar code into a sequence of decoded bits. According to [Bar04] there are around 300 different types of bar codes, which differ mainly in the way spaces and bars are arranged on the tag. The main advantages of bar codes are that they can be easily produced by printing them on ordinary printers, and that they do not require a power supply on the tag.

1D bar codes have been extensively used in Pervasive Computing related research projects to link physical objects with virtual entities. WebStickers [LH99, LRH00], for example, are small bar code tags that can be attached to arbitrary objects, and thereby

serve as physical hyperlinks to documents on the World Wide Web (WWW). This allows Web pages to be associated with physical objects; when the physical object is scanned, the corresponding page is displayed in a web browser on a nearby computer screen.

Barrett and Maglio [BM98] propose using bar codes or other forms of tags to ascertain the identity of a physical object in order to manage information associated with it. In this solution, interactions in the real world affecting a smart object are mapped to corresponding instructions for its virtual representation. For example, by scanning the bar code on a diary, a graphical interface on a nearby computer screen could allow a user to add files or calendar entries to it. When in another physical environment – in the office, in a library, or at home – the object can be scanned again and the associated information is again made available on a nearby computer. In this way, real-world actions with the physical object (taking it from the office to another place) are mapped to corresponding actions affecting the attached information (copy data from the office to my home computer).

The Palette system [NIPA99] supports people in giving presentations by means of bar codes that are printed on index cards and refer to slides in a presentation. In order to switch to a specific slide, the corresponding index card, which contains the barcode as well as a thumbnail of the slide and additional comments, must be swept over a laser scanner. The broader goal of the Palette system is to simplify the process of handling technical equipment by using bar code augmented paper.

In the ETHOC system [RB03], bar codes are used as physical hyperlinks in a smart campus environment. Handouts for students, lecture announcements, posters, and even lecture halls are augmented with bar codes so that students and lecturers can access associated online content with their handheld devices or laptop computers. For example, students can scan the bar code printed on a sheet of paper containing the homework for the next seminar in order to get background information and help – such as access to research papers related to the given tasks. The goal is to provide students with easier access to information related to their university studies.

2D Bar Codes. 2D bar codes, which are also often called two-dimensional visual codes, can store data with a higher density than 1D bar codes. In other words, a 2D bar code usually contains more information than a 1D bar code of the same size. This is because 2D bar codes store information horizontally along a tag as well as vertically; as opposed to 1D bar codes, which can be cut off vertically without losing information.

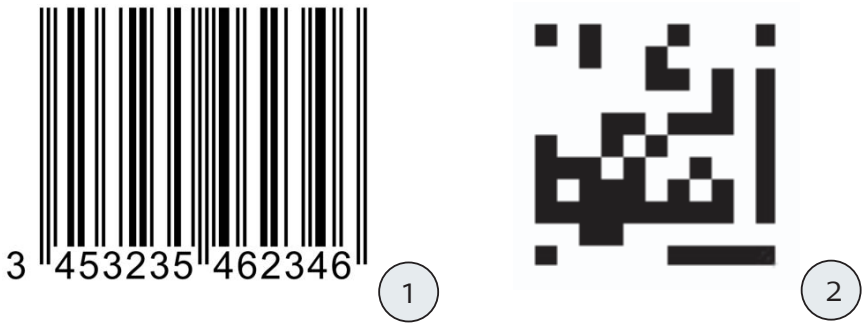

Figure 2.2: Examples of visual tagging systems: (1) a 1D bar code and (2) a 2D bar code.

Two-dimensional visual codes such as the CyberCode [RA00] were designed to be

recognized by low-cost CCD or CMOS cameras. This opens up new application domains in the field of Pervasive Computing because low-cost cameras can be deployed in large quantities as sensors throughout the environment in order to provide information about nearby tagged objects. The CyberCode system can also determine the position of an augmented item. TRIP [dIMH02] is another two-dimensional visual code system that is used for locating people in a smart environment. Rohs and Gfeller [RG04] focus on applications of 2D bar codes in connection with low-cost cameras integrated into mobile phones.

Glyphs. Like 1D and 2D bar codes, glyphs belong in the category of visual codes. However, the core characteristic of glyphs is that they are integrated into images in such a way that from the perspective of a user the embedded code becomes indistinguishable from the image itself. According to [Hec01], glyphs have approximately the same data density as 2D bar codes. However, they do not need blank space reserved on a picture but can be directly integrated into it. As people do not perceive images containing glyphs as tags, interactions with objects augmented by glyphs are usually implicit. Whereas users must often explicitly scan codes in other visual tagging systems, in the case of glyphs an image processing system in the background would read out glyphs and initiate interactions with users. The users do not necessarily need to know where the corresponding information comes from. Glyphs are an opportunity to tag objects without the sometimes disturbing and visually distracting 1D and 2D bar codes, and can hence make augmentation more unobtrusive.

Table 2.1: Properties of passive tagging systems.

	Visual tags	**Passive RF tags**	**Contact tags**
Battery	no	no	no
Manufacturing	simple (printable)	medium–hard	hard
Interaction	mainly explicit	mainly implicit	explicit
Unobtrusiveness	medium (high for glyphs)	high	low
Line-of-sight	required	not required	required
Sensors	no	possible	possible
Data density	low	low	low–high
Applications	tagging of consumer products, identification, positioning [dIMH02], physical hyperlinks [Kin02], price comparison [BG99], product recommendation [Kon02]	tagging of consumer products, object identification, physical hyperlinks [Kin02], supply chain management [ZWA+02], manufacturing [CGS+03]	product monitoring, authorisation, authentification

Passive RF tags. Passive RF (Radio Frequency) tags consist in their basic form of a small silicon chip and an antenna. Remote reader devices supply the tag with energy through an electromagnetic field and access the tag's data. Passive RF tags do not therefore require an on-board power supply, but are powered by external reader or scanner devices. We distinguish between passive RFID and passive sensor tags (see Fig. 2.1). In the case of RFID tags, the electromagnetic field issued by the reader is used to power the silicon chip on the tag and to access an internally stored identification code. The same mechanism can also be used to power a sensor integrated into the tag, which is read out and the result transmitted to the reader. In the latter case, we speak of passive sensor tags instead of passive RFID tags.

The size of RFID labels is largely influenced by the size of the antenna integrated into the tag. The antenna determines the maximum reading distance, i.e., the maximum distance a reader can be away from a tag and still access the data stored on it. The μ-chip [Hit04], for example, has the size of a dust grain without an extra antenna on the tag, being approximately 0.4 mm high and 0.4 mm wide. However, with antennas RFID tags are usually the size of bar codes or playing cards.

Besides 1D bar codes, RFID tags are probably the most widespread tagging technology. Compared to visual graphical tags, the advantages of RFID labels are (1) that they do not require line-of-sight to be read out and (2) that they are rewritable. The last property, rewriteability, is not supported by all types of RFID tags; some of them are read-only and are programmed only once during the production process. Disadvantages of RFID labels are that they are not so easy to produce as bar codes and are therefore more expensive. However, RFID labels open up entirely new application domains for tagging technologies. In Sect. 2.1.2 we have already described application areas of RFID tags in the fields of manufacturing, production, and supply chain management. We have also pointed out some of the security issues and privacy concerns that emerge from a widespread use of RFID labels.

With respect to the augmentation of everyday objects, Want et al. [WFGH99] first reported on the advantages of RFID tags for bridging physical and virtual worlds. Among these advantages are their unobtrusiveness, robustness, and aesthetics. The fact that RFID labels do not have the line-of-sight restrictions of visual tagging systems largely contributes to their unobtrusiveness. As a result, the interaction between an augmented object and its surrounding environment can be implicit and does not require explicit user attention. An example of an application with smart objects that have been augmented with passive RFID tags is the RFID chef [LMRV00]. The RFID chef suggests recipes given a number of ingredients currently available. For example, when a user returns from shopping she can put the shopping bag with all the ingredients just bought on the kitchen table. The kitchen table, which has an RFID reader integrated, scans the items in the bag and makes suggestions for the next meal on a nearby computer screen, based on the ingredients currently available.

Contact Tags. Contact tags such as the iButtons [iBu04] do also not require an autonomous power supply, but are supplied with energy by means of direct electrical contact with a reader device. This usually implies that a contact tag must be explicitly put into a special casing to ensure contact between tag and reader. As opposed to previously mentioned tagging systems, which aim at making the interactions with augmented entities as implicit as possible, the explicit actions necessary to operate contact tags are required in security-related applications. Contact tags often have an embedded and unchangeable

unique address, which can be used for identification purposes. Some tags are equipped with small discs that can be used to store significantly more data than barcodes or RFID labels; some contact tags even have integrated sensors. The Context Toolkit [DSA01] uses contact tags for sensing the presence and identity of people. Based on such sensory data, it derives information about the situation of users, e.g., the start of a meeting.

Active Tags

The defining characteristic of active tagging systems is that they do not depend on external reader devices to supply tags with energy. Instead, they are equipped with an autonomous power supply, meaning that they have a much higher degree of autonomy than passive tags, and are able to interact with their environment proactively. In other words: whereas passive tags are only operational when external readers are nearby to query the tags, active tags can interact with their environment whenever they deem it appropriate. Active tags draw their energy from an on-board battery, harvest ambient energy sources, or scavenge it from their surrounding environment. They usually consist of a microcontroller, communication modules, a power supply, and – optionally – a set of sensors. This makes them similar to sensor nodes. In fact, most of the sensor node platforms developed within the research domain of wireless sensor networks can also be used as computing platforms to augment everyday objects. The term *active tag* suggests a specific application domain of such sensor nodes, namely the augmentation of everyday items with computation. In contrast, the main drivers for sensor network research are military applications (e.g., the tracking of troops) and monitoring applications like habitat monitoring, monitoring of the structural integrity of buildings, and contaminant transport monitoring.

Although active tags can be classified by a wide range of different criteria, the wireless communication technology incorporated into tags is an outstanding characteristic that has far-reaching implications for both the tag design and prospective application areas. In the following, we therefore distinguish between infrared, active RF, and passive-optical tags depending on whether infrared, radio frequency, or passive optical communication is used by the tag to interact with its environment (see Fig. 2.1).

Infrared Tags. A typical application domain for infrared tags is the use of electronic badges for tracking people in office buildings or hospitals. In this context, examples of infrared tags are the Active Badge location system [WHFG92] and the SmartBadge [BMS97]. Although the term *badge* implies that such systems are mainly used for tracking people, in theory, the same technology can be used to track objects as well. The advantage of using infrared communication for determining the indoor location of people or objects is that infrared signals do not penetrate walls. Hence, when an infrared receiver is mounted onto the ceiling of a room and receives a signal from an infrared badge, it can be concluded with high probability that the person wearing the badge – or an object that has been augmented with this badge – is in the same room as the infrared receiver. Other advantages of infrared transmitters, which explain their frequent use in the last decade, are that they are relatively small, cheap, and easy to obtain.

To determine the indoor location of people, an Active Badge transmits a short identification (a so-called *beacon*) at regular time intervals to a nearby receiver station in order to advertise a person's presence. The SmartBadge has a similar behavior, but is also equipped with sensors for measuring acceleration, temperature, humidity, and light

levels. These sensors are used to monitor the environment of a person wearing a badge and to adapt services according to that person's current situation. Although the main application of badges is to "tag" people, Maguire [MSB98] states that a badge is a "tag for each object whose location or environment is important." However, because of the line-of-sight restriction of infrared tags, unobtrusive augmentation of everyday objects with infrared tags might not always be possible; instead, if manual alignment of objects is necessary to ensure communication with a base station, it is more appropriate to use active RF than infrared tags.

A prominent example of an everyday object that has been augmented with an active sensor-based infrared tag is the MediaCup [BGS01]. The MediaCup is a computer-augmented everyday artifact consisting of an ordinary coffee cup with an integrated sensor-based infrared tag. Using the built-in sensors, the MediaCup knows when it is being used (by measuring its current activity status by means of acceleration sensors), whether it is hot or cold (through temperature sensors), and where it is currently located (based on infrared localization). Information about the status of a MediaCup is shared with its surrounding computing environment, which can use this information to provide situation-aware services. As an example, Gellersen et al. [GBK99] reports on "colleague awareness applications" that use MediaCups and other sensors in a user's environment to track everyday activities. In these applications, cooperation among MediaCups makes it possible to derive more accurate and more detailed information about the current situation of a user. For example, the presence of many hot cups in a room could imply that a meeting is currently taking place in this room. By sharing such information with the surrounding environment, nearby mobile phones could automatically select a phone profile most appropriate for the current situation. As a result, a cell phone would just vibrate to announce an incoming call instead of issuing a loud ring tone, which would disturb the meeting.

Active RF tags. Although the line-of-sight restriction of infrared-based communication has advantages in some localization applications, it is generally difficult to ensure that infrared tags embedded into everyday objects can communicate with each other. Often, this requires manual alignment of tags and therefore full user attention, which is not feasible when there are large numbers of tags in a room that are supposed to communicate with each other autonomously. Considering the presented application domains of infrared tags, reliability of communication seems to be of only limited importance. Instead, most of the mentioned applications can tolerate lost packets or periods when no communication is possible. In case of the Active Badge system, for example, lost packets are not critical because beacons are sent out periodically. Should a beacon not reach its destination because of obstacles or other objects blocking sight, the next beacon is likely to be transmitted successfully when the user or the tagged object moves. The same is true for the MediaCup, where information about the status of a cup is also transmitted periodically to a stationary access point.

Because of the above-mentioned restrictions of infrared-based tags, today's sensor-based computing platforms are primarily equipped with radio frequency (RF) communication modules. RF communication offers the advantage of more robust and reliable data transmission, does not have the line-of-sight restriction, and therefore does not require additional user attention. Consequently, RF tags can be more unobtrusively integrated into everyday objects.

In the following, we give a short overview of computing systems that can be used

as active RF tags. Early hardware platforms that integrated RF communication with computing capabilities on a small board include the prototypes developed within the Prototype Embedded Networking (PEN) and Piconet projects [BCE+97]. Girling et al. [GWOS00] reports on a custom low-power protocol stack for these devices that ensures energy-efficient communication. The Smart-Its hardware platforms [BG03, BZK+03] were explicitly designed to be attached to everyday objects. Besides an autonomous power supply, computing capabilities, and RF communication modules they also contain sensors to perceive a smart object's environment. In addition, a wide range of small RF-based sensor nodes have been developed in the research field of wireless sensor networks. They can be used not only in the typical application domains of wireless sensor networks such as military surveillance and habitat monitoring, but also for augmenting everyday objects with computation and sensing capabilities. To name just a few, examples of sensor node platforms include the Berkeley Motes [Mot04], the sensor nodes developed in the European EYES project [Eye04], the "MANTIS hardware nymph" [ABC+03], the Medusa nodes [SS02] and the WINS sensor nodes [MAC+99]. Within the PicoRadio project [Pic04] an RF transmit beacon was developed that completely relies on ambient energy resources, namely solar energy and ambient vibrations [ROC+03]. It illustrates a promising approach to the question of how smart objects could be supplied with energy over long time periods without requiring users to change batteries. The Bat wireless tag device [ACH+01] is part of a 3D ultrasonic localization system that uses radio links to synchronize tags with a surrounding communication infrastructure and ultrasonic signals for localization. Bats can be worn by persons or attached to everyday objects to obtain their position.

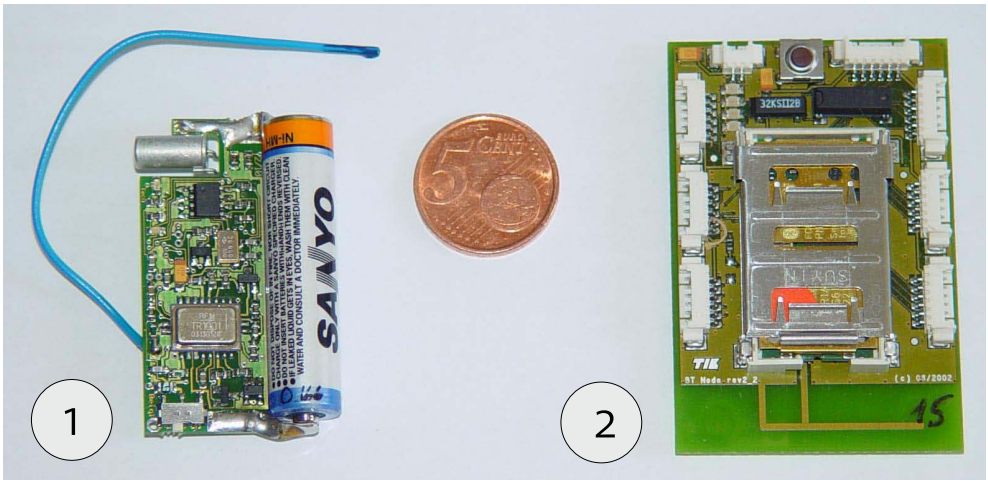

Figure 2.3: Examples of active RF tags: (1) a Smart-Its particle board and (2) a BTnode.

Considering the above-mentioned platforms, communication among tags is usually based upon custom radio protocols that strive to reduce the energy consumption of nodes. However, although custom communication protocols often do result in increased energy savings, they can limit the ability of smart objects to cooperate with other computing devices. This is because customized communication protocols are often supported only by a small subset of devices. Hence, problems with interoperability can arise if environments are populated by many different types of computing devices – as is often the case in Pervasive Computing settings. In order to ensure interoperability in smart environments, we therefore focus on active tags that are built around a standardized radio technology. In

this respect, the short-range communication technology Bluetooth has recently received considerable attention in the building of resource-restricted device platforms designed for smart environments and wireless sensor networks. Examples include the BTnodes [BKM+04], the sensor node platform built at the University of Karlsruhe [BHHZ03], the iBadge [PLS+02], the Intel Mote [Int04], and Bluetooth-enabled wireless microservers [HSR02]. Regarding cooperation, the main advantage of Bluetooth is that it is supported by a wide range of today's computing devices, including mobile phones, personal digital assistants (PDAs), laptops, digital cameras, and stationary PCs. Bluetooth-enabled smart objects can therefore be integrated into smart environments more easily, and it is becoming easier for them to cooperate with other computing devices.

Passive-Optical Tags. Integrated hardware platforms that are built around infrared or radio frequency communication modules are by far the most common active tagging systems. However, future sensor node generations such as the one envisioned by the Smart Dust project [KKP00, KKP99, Sma04a, WLLP01] could also be used to augment everyday objects. As such sensor nodes are supposed to be a mere 1 mm^3 in size, they could be distributed in large quantities over everyday objects without influencing their general appearance. Smart Dust nodes are envisioned to interact with base stations using passive optical communication: the base-station transmitter sends an optical beam to the Smart Dust node, which reflects a modulated beam back to the base station. The node does not need an active transmitter, which saves energy.

Comparison. Tab. 2.2 lists properties of selected active tagging systems and shows typical application domains. This information underlines the fact that some active tagging systems closely resemble each other with respect to their choice of a microcontroller unit and their communication subsystem.

Sensor Boards

In order to provide context-aware services, smart objects are usually equipped with local sensors that allow them to perceive their environment and to derive information about nearby users. A major design decision when augmenting everyday objects with sensor-based tags is the location of these sensors. In early hardware prototypes, sensors were usually integrated into the same board as the microprocessor and communication module. Several of today's sensor node platforms, however, aim for greater flexibility by deploying external sensor boards. In these solutions the core board – sometimes also referred to as *communication board* – contains a microprocessor, a communication module, a battery carrier, and connectors for one or more sensor boards; the sensor boards contain either a customized set of selected sensors required by a specific application, or a large number of various sensor types in order to ease the prototyping of new services. The main advantage of external sensor boards is that the same type of core board can be reused in different applications by connecting different sensor boards to it.

Integrated and Non-Integrated Sensor Boards. Generally, there are two main types of sensor boards: (1) integrated sensor boards with an extra microcontroller to provide a unified sensor interface and (2) sensor boards that only contain sensors and must therefore be accessed directly from the core board. The aim of integrated sensor boards is to provide a unified sensor interface for core boards to address sensors. In order

Table 2.2: Properties of selected active tagging systems.

Active Tag	Size (in mm)	Sensors	Processor	Communication	Applications
SmartBadge I [MSB98]	85 × 55	yes, on-board	PIC16C74, 5 MIPS	infrared, 4m range	University education, location dependent services
Smart-Its (particle core board) [Sma04c]	33 × 17 (without battery)	no, external sensor boards	PIC18F6720, 20 MHz (25 MHz and up to 10 MIPS according to data sheet)	custom RF, software adaptable transmission range	Tagging of everyday objects, positioning, context computing
Berkeley motes (MICA2DOT) [Mot04]	25 (diameter) × 6	temperature sensor on-board, external sensor boards	ATmega 128L up to 8 MIPS at 8 MHz	custom RF, range up to 300 m outdoors	Sensor network applications, environmental monitoring
Intel Mote [Int04, WKK03]	30 × 30 (without battery)	no, external sensor boards	ARM processor integrated into Bluetooth module, no extra CPU	Bluetooth, 10m range	sensor network applications, environmental monitoring
iBadge [CMY+02, PLS+02]	70 × 55 × 18	yes, on-board	ATmega 103L up to 4 MIPS at 4 MHz	Bluetooth, 10m range	Childhood education, positioning, speech processing
BTnodes (rev 2) [BKM+04]	60 × 40 × 5 (without battery)	no, external sensor boards	ATmega 128L up to 8 MIPS at 8 MHz	Bluetooth, 10m range	Tagging of everyday objects, prototyping sensor networks, university education

to achieve this goal, they are usually equipped with an additional microcontroller that maps proprietary sensor protocols to a simpler sensor or perception API. In contrast, non-integrated sensor boards merely provide a set of sensors that must be interfaced directly by the microcontroller on the communication board.

To make the underlying design decisions of both approaches clearer, we consider the sensor node platforms developed in the Smart-Its project [Sma04b] as an example. Within the scope of this project, several communication and sensor boards were designed and built. The most elaborate of the sensor boards, which was developed at the TecO in Karlsruhe, contains a digital temperature sensor, a pressure sensor, a microphone, an accelerometer, a light sensor and a loudspeaker serving as an actuator. From the perspective of an application programmer, the problem is that most of these sensors implement different proprietary protocols: the digital temperature sensor, for example, is addressed through an I2C bus interface [I2C00], whereas the acceleration sensor is read out by sampling one of its pins and transforming the corresponding samples into an acceleration value according to a proprietary equation. In order to ease the communication with sensor boards, a small extra microcontroller on the sensor board encapsulates all these different sensor protocols and provides a unified interface to access sensors via the I2C bus from the core board. This approach offers great flexibility for deploying sensor boards together with different core boards. In fact, the above-mentioned Smart-Its sensor board has been used together with different platforms. Our experiences with the BTnode embedded device platform [BKM+04] and several sensor boards has shown that integrated sensor boards make it much easier to access sensors. Furthermore, the programming statements for sampling ports and for accessing analog to digital converters differ considerably between different microcontroller families, even if the same language – such as C – is used. As a result, when a non-integrated sensor board is used together with different core boards, considerable parts of the sensor protocols must be re-implemented.

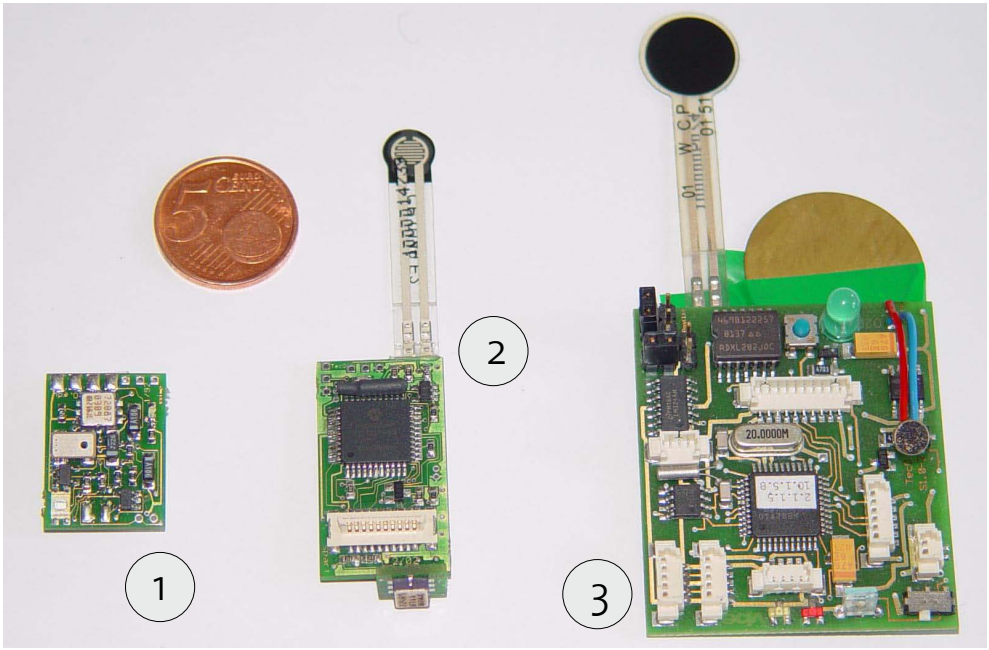

Figure 2.4: Examples of sensor boards: non-integrated sensor board (1) and integrated sensor boards with separate microcontroller (2 and 3); all sensor boards have been developed by the TecO research group in Karlsruhe.

Asynchronous Sensor Access. Besides greater flexibility and simpler deployment, integrated core boards offer another – maybe even more important – advantage: they make it easier to decouple sensor processing, such as sensor sampling and feature extraction, from communication. Using non-integrated sensor boards, a smart everyday object must carefully schedule sensor access and communication in such a way that the two tasks do not interfere with each other. Although sensor processing is not a significant problem as long as sensor access windows are short and sensor access sporadic, it can become increasingly difficult in case of streaming sensory data and if complex feature extractions are necessary to obtain information about the environment and nearby users. When local feature extractions are blocking other program executions, this would imply that no communication and hence no cooperation with other objects is possible during that time.

The advantage of integrated sensor boards is that they make it easier to implement asynchronous sensor access. A request for a sensor sample – or, more commonly, for a certain feature extracted from a stream of sensory data – is issued by the core board and transmitted to the sensor board. As the core board does not directly accesses sensors, it can continue with program execution while waiting for the result from the sensor board. During that time it can therefore still process incoming data requests. After the sensor board has finished the sensor-data processing, it triggers an interrupt on the core board, which can then conveniently access the result of the previously issued request. The implied advantages of such an approach are twofold: (1) the environment can be monitored more reliably because sensor access is not interrupted by other tasks of the microcontroller such as communication, and (2) collaboration among different nodes is faster because they are not blocked while accessing sensors. The latter aspect becomes especially important when there are many cooperating objects because a delay at a single node adds to the delay of an operation on many smart objects.

1-Processor, 2-Processor, and 3-Processor Tagging Systems

Active tags generally aim at having as few components as possible because every component will eventually consume energy. For the same reason, application programmers often have the opportunity to switch off components dynamically during program executions. Regarding the number of components, we distinguish between 1-processor, 2-processor, and 3-processor tagging systems. Besides multiple processor solutions, tags have also been built which contain more than one communication module – such as the third revision of the BTnodes, which are equipped with both a Bluetooth and a simpler RF communication chip [Btn04].

The Berkeley motes [Mot04] have only one microcontroller unit (MCU), which handles everything from communication and general processing to sensor access. Sensor boards used in connection with Berkeley motes are usually non-integrated, i.e., they are not equipped with a separate processor [Cro04b]. Berkeley motes are therefore a 1-processor tagging system.

In contrast, the BTnodes [BKM+04] possess two MCUs. One processor – an Atmel ATmega128L (cf. Tab. 2.2) – carries out the actual applications, deals with the higher layers of the Bluetooth protocol stack, and executes the basic system software. The other MCU – an ARM processor – is integrated into the Bluetooth communication module and handles lower-layer communication issues. A tagging system that incorporates a BTnode and a non-integrated sensor board is therefore a 2-processor tagging system. Deploying

BTnodes together with an integrated sensor board constitutes a 3-processor system.

A slightly different approach is taken by the Intel motes [Int04]. The Intel motes, whose communication is also based on Bluetooth, reuse the microcontroller integrated in the Bluetooth modules for executing applications. As a result, only one instead of two processors is needed on the core board, which can save considerable amounts of energy. However, the downside of this approach is that obtaining the right to change code directly on the processor of the Bluetooth module is difficult and expensive. Furthermore, it hinders the exchange of program code in the research community because of accompanying legal issues.

The decision whether to use a 1-processor, 2-processor, or 3-processor tagging system is important in any project that deals with active tags. It is also largely influenced by economic factors. As already pointed out, more processors usually result in increased energy consumption. Integrating all tasks on a single microcontroller is therefore desirable, and it is likely that future active tagging systems will strive in this direction. Especially in the prototyping phase of a research project, however, it can be beneficial to combine different sensor and communication boards that contain their own processors. As integrated sensor boards, for example, hide the complexity of accessing sensors by exporting a unified software interface, this can result in higher flexibility and lead to a simpler deployment of different sensor boards.

Energy Considerations

In Sect. 2.1.1 we have already discussed why low energy consumption is a precondition for successfully deploying active tagging systems. However, it is not always clear what energy efficiency really means in connection with the envisioned applications. Is it measured based on the energy dissipated per transmitted bit, the overall energy consumed over a certain time period of low or high activity, the energy consumption of a single node, or that of the whole network? The overall lifetime of the network is typically the most important criterion in wireless sensor networks. However, in this application domain, networks are usually more homogeneous than in Pervasive Computing settings, where the lifetime of some single nodes – for example those that cannot be easily recharged – might be more significant than that of others. In the following, we give an overview of the energy consumption of selected active tagging systems.

When evaluating the power consumption of an active tag, communication is usually the determining factor, above computation. For example, Park et al. [PLS+02] reports that for the Bluetooth-based iBadge system, the energy dissipation per transmitted bit amounts to $1.3 \times 10^{-5} J$ compared to $3.9 \times 10^{-8} J$ per instruction carried out on the integrated microcontroller. However, according to their equation the power consumption of the main processor running at 4 MHz is 47.8 $mA \times 3.3\ V$ compared with 26 $mA \times 3.3\ V$ for the Bluetooth modules, which puts the above mentioned figures in a slightly different perspective. With respect to the Berkeley motes, these figures differ even more significantly. According to the data sheet of the MICA2DOT motes [Cro04a], the current draw of the microcontroller on this platform is 8 mA in active mode, compared to 25 mA and 8 mA for the communication module during transmission with maximum power and reception.

Several papers deal with the energy performance of active tagging systems. Leopold et al. [LDB03] states that the power consumption of the MCU on the BTnodes (running at a rate lower than the maximum clock frequency) is about 46 mW. When switching on

the Bluetooth modules and setting them into a state where they are ready for connection establishment, a BTnode consumes about 89 mW, whereas 136 mW are necessary to maintain connections, and an additional 65 mW are needed when transferring data with 6 KiB/s. Hence, Leopold et al. [LDB03] reports that a BTnode consumes about 50 mW in idle mode and 285 mW when communicating. It is also argued that the energy per communicated bit on a BTnode is about 5.5 $\mu J/bit$ at a data rate of 6 KiB/s. With respect to the energy per transmitted bit, this is comparable to the Berkeley motes, where the energy per bit amounts to 4 $\mu J/bit$ or 15 $\mu J/bit$, depending on the researchers who measured it [HSW+00, LDB03].

Table 2.3: Energy characteristics of selected active tagging systems.

Berkeley mote (MICA2DOT) [Cro04a]	
Processor	
Active mode	8 mA
Sleep mode	$< 15\ \mu A$
Radio	
Transmit with maximum power	25 mA
Receive	8 mA
Sleep	$< 1\ \mu A$
Berkeley mote – early version [HSW+00]	
Overall power consumption in active mode (without LEDs)	60 mW
Overall power consumption in active mode (with LEDs)	100 mW
Overall current draw in sleep mode	$10\ \mu A$
Energy per bit during transmission	$\approx 4\ \mu J$
Energy per bit during reception	$\approx 2\ \mu J$
BTnodes – revision 2 [LDB03]	
Overall power consumption in idle mode	50 mW
Overall power consumption while communicating	285 mW
Energy consumption per bit	$5.5\ \mu J/bit$
Smart-Its sensor board [BZK+03]	
Capacitive microphone and amplifier	8 mW
Pressure sensor	$16\ \mu W$
Temperature sensor	2.3 mW

However, such comparisons between BTnodes and Berkeley motes should be considered carefully. As Min and Chandrakasan point out in their paper titled "Top Five Myths about the Energy Consumption of Wireless Communication" [MC03], latency is often traded against reduced energy consumption. Furthermore, the dissipated energy per bit often decreases as the data rate increases. For example, according to Min and Chandrakasan [MC03] the energy per bit for an RFM TR 1000 module (a communication module similar to those used on the Berkeley motes) sending at 2.4 Kbps is 14 μJ, whereas it is only 372 nJ when the module is sending at 115.2 Kbps. Also, the energy per bit for a Cisco Aironet 350 WLAN card is 236 nJ at 11 Mbps, which is even smaller. However, because of the high static energy consumption and the nature of the envisioned applications, WLAN cards might not be a good choice for active tagging systems. With

respect to Bluetooth, it should be kept in mind that although the application data rate might be only 6 KiB/s – as in the experiments of Leopold et al. [LDB03] – the data rate for transmitting data over the air is still the standard Bluetooth rate of 1 Mb/s. The required energy to transmit a single bit seems to be a good criterion only if the underlying applications really need the high data rate provided by the communication modules. Hence, supported data rate and desired application throughput should match in order to achieve good energy performance.

With respect to the energy performance of infrared tagging systems, an early version of the SmartBadge, the SmartBadge I, has an overall average current draw of 98 mA [MSB98]. Regarding passive-optical tags, Kahn et al. [KKP00] points out that optical data transmission requires significantly less energy that RF-based communication. Besides communication and data processing, the energy required to sample sensors is another important issue when choosing an appropriate tagging technology for smart environments. Beigl et al. [BZK+03] gives a thorough analysis of the power consumption of several sensor types. The authors show that power consumption can vary significantly between different sensor types, and they conclude that it is sometimes not negligible, compared to that of the communication and processing subsystems.

Tab. 2.3 compares the energy characteristics of selected active tagging systems found in the literature; the energy consumption of the processing, the communication, and the sensing subsystem are all considered.

Besides the general energy characteristics of major hardware components, the overall energy consumption of an active tag also depends on communication protocols and strategies for accessing sensors. MAC (medium access control) protocols for wireless sensor nodes, for example, strive to save energy by reducing the amount of time a communication module has to listen for incoming data. By reducing the time for idle listening, the number of packet collisions, overhearing, and the number of control packets, the energy consumption of a sensor node can be decreased considerably [YHE02]. The energy consumption of a microcontroller can be reduced by active power management. Here, the clock frequency of a processor can be reduced by an application in times of low activity in order to save energy. Other strategies for power management are discussed in [RSPS02].

Other Tagging Systems

Besides the active and passive tagging systems presented here, it is also possible to use chemical or biological methods to tag everyday objects [Cap04]. Certain chemical compounds and radioactive substances can be mixed into other materials or attached to objects and thereby serve as chemical tags. In the case of biological tags, it has recently become possible to synthetically encode digital information into biological molecules. Chemical and biological tags are often used to uniquely identify goods such as medicine or high-quality products. As a result, protection against counterfeiting, i.e. brand protection, is the core application area for these tagging systems. However, there are also applications in which chemical tags are used to track animals, and DNA-based tags for identifying cars involved in road accidents.

2.3 Context-Awareness

2.3.1 Context

In the scope of this dissertation, context is an important concept because context recognition is one of the core reasons for smart objects to cooperate with each other. This is because the "smart" behavior of smart objects usually stems from the fact that they can perceive their environment, derive information about the situation of nearby people, and adapt application behavior accordingly. As a single object alone can only perceive a small subset of its environment, it is often forced to interact with other devices within range of it, in order to obtain more detailed information about the situation of nearby users. Regarding the nature of *context*, there are several approaches in the literature to define the term *context*, which alone indicates that the notion of *context* or *context-awareness* is not easy to grasp and formalize. This section tries to identify the core aspects behind previous definitions put forward by other researchers, and aims to explain why the term context is so important for the emerging field of Pervasive Computing.

General Usage of the Word Context

A reason for the different approaches to defining context in Pervasive Computing might be that many people have a – slightly different, slightly more or less complex – individual understanding of context. This in turn might be due to the fact that the term context plays a role in several disciplines of science, and specifically in other sub-areas of computing such as compiler construction (e.g., context conditions, context-free grammars), artificial intelligence (e.g., contextual reasoning), or operating systems (e.g., context switch).

The Oxford Advanced Learners Dictionary [Hor00] provides two definitions of context: (1) *the situation in which something happens and that helps you to understand it* and (2) *the words that come just before and after a word, phrase or statement and help you to understand its meaning*. When comparing these two definitions, the core message of context seems to be that it provides a more detailed understanding of something by considering it as integrated into and influenced by its surroundings; not as isolated, self-explaining, or self-contained.

Linguistics

This is also how the notion of context is used in linguistics, where context information helps to dissolve the ambiguities of words or phrases. The meaning of a word often becomes clear only in the *context* of a sentence or paragraph. In this connection, linguists distinguish between the syntactical, semantic, and pragmatic dimensions of language. Whereas the syntactical dimension of a word is self-explanatory, determining the meaning (semantic dimension) or even the implicit requests for actions embedded into words or phrases (pragmatic dimension) requires an increased level of knowledge that cannot be inferred from a word or text alone. The English word "board," for example, has several different meanings, such as piece of wood (as in floorboard or hardboard), tool used in sports (surfboard, snowboard), and group of people (the board of directors, academic board). Hence, the semantics of the word "board" only become clear at the phrase, sentence, or paragraph level. Deriving the pragmatic dimension of a phrase requires even more knowledge that is not restricted to the text itself, for example knowledge about the situation of people participating in a conversation. The pragmatic dimension of "Frank,

you don't need to get up; I'll get you a beer." might imply just what it is saying if Frank has a broken leg. But it could also be pure sarcasm, criticizing the subject's laziness. For a computer, it is easy to understand the syntax of words, but extremely difficult to interpret the semantic and pragmatic dimension of language because this requires additional information about the surrounding context or the situation of people.

Context in Pervasive Computing

To understand the use of the term context in Pervasive Computing, it is important to realize that the interactions between humans take place at different levels that roughly correspond to the dimensions of language. If smart objects want to provide "smart" behavior that automatically adapts to the needs of nearby users, it is necessary for augmented items to obtain information about the situation of people, instead of processing only explicit and self-explanatory user input.

In the age of mainframes and PCs, it was difficult to recognize the situation of people, which made human-computer interaction more technology- than human-centered. Computer applications merely considered explicit input from users that had to be fed to information systems by means of conventional input devices such as keyboards, mice, and buttons. Explicit input can be compared to the syntactic dimension of language in that it must be completely self-explanatory; it does not require computer systems to monitor the current situation of users because any instructions the computer can carry out are controlled by explicit actions. However, as computing technology becomes pervasive, interaction with computer systems based purely on explicit actions becomes inappropriate. This is because there are so many computing devices, and because much of the technology will be invisible from a user perspective. Consider, for example, a tour guide in a museum – a favorite application domain for experimenting with context-aware applications. Usually, a visitor has to input a number on a mobile device to get information about an exhibit. In a simple version of such a system a user would always get the same amount of information independently of her situation and independently of how she has behaved previously. In contrast, a context-aware application would also consider her surroundings, how long she had been standing in front of other exhibits and whether she is visiting the museum together with other people, in order to adapt the amount of information and the way in which it is presented. Of course, the information about a user's situation could also be embedded into explicit actions, e.g., by requiring additional input specifying the desired amount of information about an exhibit. However, this does not suit the way in which humans interact with each other, becomes increasingly complex with an increasing number of devices and options, and is therefore not a human-centered approach to interacting with computers or smart environments. Context-aware applications aim to be more receptive to users' needs by considering their current situation and by automatically adapting their application behavior accordingly.

In the early days of computer science, context-aware applications were difficult to implement because information about the situation of users was difficult to obtain. With the vision of Pervasive Computing, however, this has changed dramatically: omnipresent information technology embedded into everyday environments allows us to retrieve information about the situation of people with unprecedented accuracy. This information can be used to provide more natural user interfaces to smart objects and smart environments in general. Omnipresent computation and the need for a more user-centered approach to interactions with computers are among to the core reasons why *context* and *context-*

awareness have become of paramount interest for the field of Pervasive Computing.

Context in Other Areas of Computer Science

The term *context* plays also an important role in the field of compiler construction, which deals with the concepts of context-free grammars, context-sensitive grammars, and context conditions. Crowley et al. [CCRR02] reports on the usage of *context* in artificial intelligence and computer vision.

What all these different usages of context have in common is that they try to get a more detailed understanding of something by considering how it is affected by its surroundings. The paramount message of context is that it aims to provide a more detailed understanding of something by considering it as integrated into and influenced by its environment. This is exactly the way context is used in Pervasive Computing: by observing the current surroundings of people, which becomes possible through information technology embedded into everyday environments, we get a more detailed understanding about their intentions and can offer human-centered services that take their real-world environment into account.

Definitions of Context

A detailed review of how researchers have used and defined the term context can be found in several papers by Dey [DA00, Dey00, Dey01, DSA01]. Schmidt [Sch02] also provides an extensive discussion of the evolution of the term and its usage in the field of mobile and ubiquitous computing. Dey and Schmidt point out that initial definitions of context merely listed examples to explain its meaning. Schilit and Theimer [ST94] for example, who originally introduced the term "context-aware" refer to context as location, collections of people and objects in an application's environment, and changes to those objects over time. Several researchers have provided synonyms for context to explain its meaning. Here, context is regarded as either the user's or the application's environment or situation [DA00, Dey00, Dey01, DSA01, Sch02]. In Schilit et al. [SAW94], context is considered more generally by identifying three core aspects of it: where you are, who you are with, and what resources are nearby. This approach also tries to define context more broadly and independently of the concept of location. Context is thereby not only restricted to properties derived from the location of objects and people, but instead also includes network connectivity, communication costs, and the social situation of people.

In connection with the Context Toolkit [Dey04], Dey refers to context as "*environmental information that is part of an application's operating environment and that can be sensed by the application.*" The most widely accepted definition of context today has been put forward by Dey and Abowd [DA00]: "*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*"

Whereas in this definition context is any information relevant for characterizing the situation of an entity, Chen and Kotz [CK00] distinguish between the characteristics that determine the behavior of an application and those that are relevant but not critical. In their definition "*context is the set of environmental states and settings that either determines an application's behavior or in which an application event occurs and is interesting to the user*" [CK00].

Schilit [Sch95] defines context-aware computing as "*the ability of a mobile user's applications to discover and react to changes in the context in which they are situated.*" Context-aware applications adapt to a constantly changing execution environment, where the environment can be classified into three subcategories: (1) the computing environment, (2) the user environment, and (3) the physical environment [DA00, Dey00, Dey01, DSA01]. The computing environment can be characterized by virtual-world parameters such as the current bandwidth and throughput of communication technologies, the amount of available memory, the processor load, the identity of connected devices, or the number of available input/output devices. Virtual-world parameters describe the inherent properties of a computer system and the aspects that have a direct impact on data processing and data communications, but do not consider the real-world surroundings of a computer system. In other words, virtual-world parameters describe the classical self-contained properties of computing systems. In contrast to the computing environment, the user environment identifies the situation of people participating in a particular context-aware application. It can be characterized by parameters such as their current location or social situation (e.g., whether a user is currently together with friends or family members, or whether she is having a meeting with her boss). The physical environment describes the real-world surroundings that influence an application. Light or noise levels, the current temperature and humidity are parameters that define an application's or a user's physical environment.

With respect to this classification, this thesis distinguishes between virtual-world contexts and real-world contexts (sometimes also referred to as situational contexts). Virtual-world context refers to the current computing environment, e.g., the amount of computing resources available, whereas real-world context describes the situation of an entity in the real world. Real-world context therefore corresponds to the above-mentioned user environment and physical settings. Furthermore, in the scope of this thesis, it makes sense not only to regard the real-world surroundings of a user as relevant but also the real world situation of smart objects. We therefore suggest an artifact-based approach to context, where not only the user environment but also the real-world situation of computer-augmented artifacts becomes important (cf. Sect. 3.6).

2.3.2 Infrastructural Support for Context-Awareness

The definitions presented in the previous section do not introduce a formal model of context. The questions of how context is represented in a computer system, and how concrete applications can derive context information in typical Pervasive Computing settings are also left open. In this section, we give an overview of selected frameworks and infrastructural components for deriving context, which all have a specific approach to representing context and different formalized or semi-formalized models of it.

The Context Toolkit

The Context Toolkit by Anind K. Dey is a framework that facilitates the design and the rapid prototyping of context-aware applications [DSA01, SDA99]. The framework provides support for persistently storing context data, facilitates resource discovery, and introduces three core abstractions for handling context information: context widgets, interpreters, and aggregators.

Context widgets hide the low-level details of accessing sensors, and deal with the

general low-level handling of sensory data. The underlying concept stems from GUI (Graphical User Interface) widgets, which are abstractions that make it easier for application programmers to process user input in graphical applications. In such situations, GUI widgets encapsulate implementation details of GUI elements such as buttons and menu bars, are able to capture mouse clicks or key strokes destined for these elements, and automatically translate user input into corresponding application events. Using GUI widgets, it becomes irrelevant whether a keyboard or a mouse is being used to press a button, for example. Instead, a widget automatically captures input and communicates with an application by means of *button pressed* or *button released* events, which frees the application programmer to continuously monitor the status of a GUI element. In this respect, context widgets pursue similar goals:

1. Context widgets hide the complexities of sensors. From the perspective of an application programmer it thereby becomes transparent what specific sensors are used to measure a certain physical property as long as the widget adheres to a well-defined interface. Hence, it is possible to substitute sensors without changing application code.

2. Widgets provide context data in a way that makes it easier for an application to handle context. For example, context widgets can automatically notify an application about certain context changes, can be polled for sensory data, or queried.

3. Context widgets can be reused in other applications.

Interpreters transform context data from one form into another. For example, an interpreter that is associated with a widget providing the identity of a person in form of a simple ID number could map the ID onto the real name of the identified person. Interpreters can also transform physical coordinates into symbolic location information, e.g., GPS (Global Positioning System) coordinates into the name of a town or a street. Aggregators collect low-level context information from multiple distributed sources and transform it into a high-level form more meaningful to applications.

With respect to smart everyday objects, the major drawback of the Context Toolkit is that it is designed for relatively powerful computers. The current implementation of the toolkit is written in Java, which makes it less suitable for smart everyday objects that are usually faced with severe resource restrictions and programmed in C. Furthermore, communication in the context toolkit is based on XML (eXtensible Markup Language), which inflates the amount of data that needs to be transmitted. As we will argue in Sect. 3.5, however, compact communication protocols are of paramount importance in environments of wirelessly communicating smart everyday objects. Conceptually, the framework aggregates sensory data from different sources, but this aggregation is done in a background infrastructure on powerful computers with powerful networking interfaces and practically unlimited energy resources. In contrast, our approach to deriving context focuses on environments where many resource-restricted devices need to derive high-level context in dynamic groups of cooperating entities without constant background infrastructure access (see Chap. 3).

XML-Based Representations

XML-based representations of context have the benefit that they present information in a human-readable format that can be automatically processed by computer systems.

Furthermore, there are various tools that can process, display, and transform XML documents, which makes it comfortable to work with this data format. XML is also one of the most influential standards for representing data on the World Wide Web (WWW). Considering all these advantages, XML seems to be a natural choice for representing context information. However, when evaluating XML for the purpose of exchanging context information among resource-restricted smart everyday objects, it must be kept in mind that because XML aims to be a human-readable format, it can significantly inflate the size of context descriptions. This is because XML is based on simple text, i.e., a seven digit number in XML needs 7 bytes if ASCII or UTF-8 is used as an encoding standard. In contrast, a simple binary encoding of such a number would require only 3 bytes. Also, the structure of XML files with nested human-readable tags increases the size of data representations. Although XML-based context descriptions could be compressed before they are exchanged between smart objects, which would require additional processing, a more compact description format for context would be likely to lead to less communication overhead. In Sect. 3.5, we will argue that wireless ad-hoc communication is a bottleneck for cooperation between smart everyday objects. A compact representation for context is therefore important in these settings; hence, XML might introduce too significant an overhead for exchanging context information. However, when context data needs to be transmitted in the background infrastructure, XML is a good choice for representing context information because of the above mentioned advantages.

In the following, some approaches to embedding context information into XML documents are presented. Brown et al. [BBC97, Bro96] introduces an SGML-based language for specifying context. SGML, the Standard Generalized Markup Language, is a superset of XML, which at the time Brown published his results was not yet an official W3C (World Wide Web Consortium) standard. The goal of using an SGML-based syntax was to facilitate the development of context-aware applications by making the process of specifying context-awareness as easy as designing Web pages. As Web documents are written in HTML (Hypertext Markup Language), which is widely accepted by programmers and is also a subset of SGML, it is argued that an SGML-based representation of context simplifies the creation of context-aware applications. The specification of context-aware behavior takes place in documents called *stick-e notes*, which consist of a context description and a body. Regarding the context description, Brown introduces an extensible mechanism for specifying context conditions and syntactic constructs for describing time, place, user orientation, and user presence. The body of a stick-e note is executed whenever the corresponding context condition is matched. Schmidt [Sch00] extends Brown's notation by adding more sophisticated mechanisms for grouping contexts and for triggering actions associated with them. Although Schmidt's notation conforms to the XML standard, it is explicitly stated that for resource-restricted devices a different implementation language is used to actually realize a context-aware application.

PML Core [FAOH03], the Physical Markup Language Core, is a recommendation of the Auto-ID center that specifies an XML format for the exchange of sensory data between components of an Auto-ID infrastructure. It focuses on the representation of such sensory data, which could theoretically be regarded as low-level context information, but leaves open how such data are aggregated and forwarded or used by applications. As a data exchange format, PML is more suitable for powerful backend infrastructures because of the considerable size of data representations. There are also other standardization attempts that specify data exchange formats for business applications based on XML [Ros04, XML04].

The Semantic Web initiative [Sem04] designed a framework for sharing machine-readable data on the Web. In this framework, RDF XML, the XML-based format of the Resource Description Framework (RDF), is the standard format for data exchange. By using the ontology and interference language DAML+OIL (DARPA Agent Markup Language + Ontology Interface Layer), which is based on RDF, it becomes possible to derive complex data [Hor02]. The tools provided by the Semantic Web can be used to specify context information as well as adaptive application behavior. In fact, the W3C provides its own definition of context [BL03]. Wang et al. [WGZP04] and Gu et al. [GWPZ04] present a context ontology in OWL (Web Ontology Language). It is shown how logic reasoning can be used to derive higher-level context data, and it is possible to check the consistency of context models in their approach. As already mentioned, the Context Toolkit also deploys a communication model that is based on XML.

Context Representations based on First-Order Predicate Calculus

First order predicate calculus is a powerful tool for modeling context because it can be used to describe contextual states and facilitates automatic reasoning about contextual information. The expressive power of the predicate calculus in this respect has been illustrated in the Gaia project [Gai04, RHC+02a, RHC+02b]. The Gaia context infrastructure introduces a notation for context information and a mechanism for deriving higher-level context data based on first-order predicate calculus [RC03, RCRM02].

In the GAIA context model, low-level context information is represented as first-order predicates. The name of a predicate determines the type of context it represents. A predicate usually has three attributes that reflect the subject-verb-object structure of simple English sentences. Regarding the context "location," for example, the predicate *Location(Frank, in, Room D41.2)* would express that the subject Frank is currently in room D41.2. It is argued that most other types of simple contexts can be expressed in a similar three-attribute predicate. The naming and overall structure of these predicates is specified in an ontology that was defined using the Web ontology language DAML+OIL [Hor02]. More complex context types are generated by connecting context expressions with Boolean operators such as negation and conjunction, or by parameterizing context expressions with quantifiers like $\forall$ and $\exists$. Using the predicate calculus it also becomes possible to deduce new contexts or to fuse sensory data. For example, $Location(Frank, in, Room\ D41.2) \wedge Sound(Room\ D41.2, <, 40\ dB) \Rightarrow EmployeeStatus(Frank, hardworking)$ would mean that Frank is hardworking when he is in room D41.2 and is not talking or listening to music. In Gaia's context infrastructure several infrastructural components such as context providers, context synthesizers and context history services are responsible for collecting, processing, and storing context information. Applications can query context providers that evaluate these queries using a Prolog reasoning engine. The rules that describe how high-level contexts are derived from basic sensor percepts are coded in Prolog-like logic programs.

First-order predicate calculus is an interesting tool for handling context. However, on resource-restricted smart everyday objects it is questionable whether Prolog-like logic programs can be executed with adequate efficiency and carried out in an acceptable time. The presented approach again assumes that there is a powerful background infrastructure available that carries out all reasoning-related tasks. However, a core idea of smart everyday objects is that they are autonomous and able to provide context-aware applications themselves. This results in new requirements for building context-aware applications in

smart environments.

Context Frameworks for Resource-Restricted Smart Devices

As cooperation among resource-restricted smart everyday objects is a main focus of this dissertation, we now give an overview of two context frameworks that were explicitly designed for computing devices with limited amounts of resources. These frameworks are (1) the context architecture for building smart appliances developed in the TEA project [GSB02, SL01, TEA04], and (2) the context architecture proposed by Malm et al. [MKVA03].

With respect to the context infrastructure proposed in the TEA project, Schmidt and Laerhoven [SL01] introduces a four-layered architecture for implementing context-aware applications consisting of sensors, cues, contexts, and an application layer. The cue layer hides implementation-specific details of sensors from upper layers and can also carry out limited preprocessing of sensory data. In their terminology, cues are characteristic features extracted from a set of sensory data. For example, the average of a list of sensor samples and the first derivative of sensor values obtained over a certain amount of time are called cues. The context layer then derives information about the current situation of people or devices based on these cues and makes context information available to applications. What is especially notable about the TEA project is that it resulted in the design of a smart mobile phone augmented with an active sensor tag. The active tag attached to this phone was the major platform involved in the context-recognition process. However, the authors do not report on experiences about collaborative context recognition among sets of cooperating devices.

Malm et al. [MKVA03] suggests adjusting the main concepts proposed in the Context Toolkit (cf. Sec. 2.3.2) to the requirements of resource-restricted active tagging systems. The main components of their architecture are context widgets, aggregators, and discoverers. As in the Context Toolkit, context widgets hide the details of accessing sensors, and aggregators can refine low-level context information provided by widgets or other sources. Discoverers search for services on remote devices and advertise local sensor sources. So far, there is unfortunately no report available on the experiences with such a system. It is also hard to predict whether the concepts of the Context Toolkit suit the needs of smart everyday objects. For example, the Context Toolkit is based on Java, which, due to its object-oriented nature, provides comfortable ways to deal with high-level programming concepts such as widgets and aggregators, and makes it easier to reuse code. Furthermore, the Context Toolkit does not address the complicated communication-related issues involved in deriving context among cooperating smart objects, but instead relies on high-level communication abstractions that are not available in environments of resource-restricted augmented artifacts.

A Semi-Formal Model of Context for Perceptive Computing

Crowley et al. [CCRR02] introduces a semi-formal model of context and presents an operational theory of context awareness. Context is thereby modeled as a set of roles and relations. A role describes the specific functions of entities, i.e., the specific roles entities can assume in a certain environment. An entity can have different roles associated with it; e.g., a pen can serve as a writing tool or as a pointer. Here, *pen* is an entity, and *writing tool* and *pointer* are roles. Relations are predicates describing the properties of

entities. For example, "looking at" is a relation between entities, which could contain the tuple *(Alice, Alice's pen)* when Alice looks at her pen. In the proposed model, context determines which roles and relations need to be observed by a context-aware application. Only in a certain *situation* can tuples of entities be assigned to relations and roles to specific entities (e.g., in one situation a pen can be used as a writing tool and in another as a pointer). This model of *context* and *situation* was prototypically implemented based on federations of observational processes that visually track entities in an environment and provide derived context information to applications.

Graphical notations for representing context

Henricksen et al. [HIR02] proposes a graphical notation for modeling context that is related to UML (Unified Modeling Language) and Entity-Relationship diagrams, known from the fields of software engineering and database research. The notation is claimed to be especially suited for modeling specific aspects of context such as context imperfections, temporal characteristics of context, and dynamic context changes. The basic components of the proposed context model are entities, associations between them, and attributes that describe certain properties of entities. The special needs of context-aware applications are addressed by introducing a range of specific associations (such as static or temporal associations) for the representation of context and the interrelations between entities.

The concepts behind the Semantic Web [Sem04] are also suitable for modeling context and dependencies between objects in context-aware applications (the Semantic Web introduces yet another definition of the term *context*). The Resource Description Framework (RDF), which plays an important role in this regard, also has a graphical notation. Müller [Mül97] uses conceptual graphs previously introduced by Sowa [Sow04] to model context in a particular application involving electronic patient records.

Shortcomings of Previous Approaches for Handling Context

Tab. 2.4 compares some of the previously presented approaches for building context-aware applications. It can be seen that they primarily focus on environments where contextual information is processed and aggregated in a powerful backend infrastructure. These approaches assume that although computational resources are distributed throughout the environment, resource-restricted low-end devices merely provide sensory data or object identifications to infrastructure components, which handle computations and human-computer interaction related tasks on behalf of the entire environment. The reason for this prevalent assumption is that until recently smart objects have been regarded as just powerful enough to accumulate and forward data to backend servers. Furthermore, many smart objects were augmented with passive tagging systems (cf. Sec. 2.2.3), which implies that such augmented items cannot provide applications autonomously. Instead, another kind of computing system, e.g. a backend infrastructure server or a nearby handheld device, had to implement the actual services on behalf of smart objects. In other words, augmented items were merely the fingertips of larger computing systems, supplied them with sensory data, and completely relied on backend infrastructure services to realize applications.

This changed with the development of more sophisticated active tagging technologies, which provide smart objects with a considerably greater amount of autonomy. Although more complex functions still need to be outsourced to a backend infrastructure, it is

becoming possible for smart objects to offer services to nearby users directly. This has several advantages:

- **Autonomy.** Smart objects do not depend on a permanently available communication link to a backend infrastructure. As active tagging technologies are based on short-range communication technologies because of energy restrictions, this allows for significantly higher *flexibility*: smart objects can be used and can provide services almost everywhere as they do not rely on nearby access points.
- **Convenient interaction.** More powerful everyday items can provide more sophisticated services to their users. A critical issue in this connection is implicit human-computer interaction. Objects augmented with active tags are much better suited to triggering interactions without requiring explicit user input because they can autonomously observe their environment and act according to the situation of nearby people.

Taking advantage of these properties requires that smart objects can derive context information more autonomously – i.e., without constant access to backend infrastructure services. Furthermore, the severe resource-restrictions of smart objects and their wireless ad hoc communication interfaces must be taken into account when realizing context-aware applications on smart objects. In Chap. 3 we approach some of these issues and show how smart objects can realize context-aware applications in cooperation with each other without requiring backend infrastructure access. The context infrastructure developed in the TEA project (cf. Tab. 2.4) is closest to the solution we desire, but it is vague regarding inter-object interaction and focuses on the context recognition process on single devices.

Table 2.4: Comparison of selected toolkits for building context-aware applications.

	Context Toolkit	GAIA Context Infrastructure	Semantic Web Context Infrastructure	TEA Context System
Basic concepts	Context widgets, aggregators, interpreters	First order predicate calculus and logic programs	OWL and first order logic	Cues, cue aggregation
Degree of formalization	Low	High	High	Low
Resource demands	Medium (Java code + XML)	High (Prolog like logic programs)	High (semantic web toolkits)	Low (C code)
Context computation	In backend infrastructure	In backend infrastructure	In backend infrastructure	In backend infrastructure and on smart objects

2.4 Summary

Briefly stated, Pervasive Computing envisions a world in which computation is seamlessly integrated into everyday environments in order to support people in their daily activities. In the first part of this chapter, we dealt with this vision of Pervasive Computing, reasoned about its technical feasibility, and discussed economic as well as social implications. In our argumentation, we started with the technological foundations of Pervasive Computing, such as Moore's law and recent advances in wireless communications, and reported on technological problems. We then discussed the economic relevance of Pervasive Computing and presented selected application scenarios. These application scenarios showed that Pervasive Computing is already having an impact on economic processes; for example in the field of supply chain management, where it facilitates the realtime tracking of products. Last but not least, we discussed concerns regarding personal privacy, which arise with an increasing adoption of Pervasive Computing technologies.

The second part of this chapter dealt with smart everyday objects. We provided a definition of the term and identified different approaches for adding computation to everyday things. We then gave a thorough overview of tagging technologies, and discussed how these tagging technologies affect cooperation among smart objects.

The third part of this chapter focused on *context* and *context awareness* because context recognition is one of the core reasons for smart objects to cooperate with each other. We reviewed definitions of context, gave an overview of frameworks for deriving context information, and discussed their suitability for environments of cooperating smart everyday objects.

Chapter 3

Cooperation among Smart Everyday Objects

In our effort to support the main thesis of this dissertation stating that cooperation helps smart objects in realizing services for people in smart environments, this chapter focuses on the cooperation between different augmented artifacts. The goal is to show how smart objects, which are all faced with the same kind of severe resource restrictions, can implement sophisticated services by bundling their resources and capabilities.

The solution of choice to achieve this goal is to form groups of cooperating objects according to their real-world context and application specific information. Nodes in a group exchange sensory data by means of a distributed data structure that hides the actual location of resources, and hence, appear to applications as a single node accumulating the resources of collaborating entities. Based on this infrastructure layer, we present a description language for the design of collaborative context-aware services. The language helps application designers to clearly distinguish between the core concerns of context-aware applications, and thus to write more structured and legible code than would be possible in the predominant C programming language. Our language also supports programmers in describing how high-level context information is derived from simple sensor samples, especially when this involves sensory data from multiple smart objects. We then present a compiler for this description language that transforms context-recognition statements into actual instructions for smart objects, and into corresponding instructions operating on the previously mentioned distributed data structure. In addition, it is shown how the operations of this distributed data structure can be efficiently mapped onto the primitives of the underlying wireless communication technology. The last core aspect covered in this chapter deals with wireless communication and how it affects the cooperation between smart objects. Ad hoc wireless communication between augmented artifacts is likely to be a bottleneck for realizing efficient inter-object collaboration because of the many objects potentially in range of each other and their severe resource restrictions. As a possible solution to alleviate this problem, we propose considering the real-world situation of smart objects in order to establish networking structures that are more suitable for context-aware applications.

The rest of this chapter is organized as follows. The next section gives a short overview of related work, and explains why it is beneficial for smart objects to cooperate with each other. In Sect. 3.2, we sketch out our approach for realizing collaborative services on smart objects; and present a software architecture allowing augmented items to provide collaborative context-aware services. In the remaining sections we then describe the

individual layers of this architecture: the application layer, the context layer, an infrastructure layer supporting inter-object cooperation, and the communication layer. Sect. 3.3 starts with the application layer and presents the above-mentioned description language for context-aware services. Sect. 3.4 focuses on the context layer and shows how a concrete description of a context-aware service is mapped onto instructions for retrieving and merging sensory data. Sect. 3.5 describes the layer for inter-object cooperation, which is responsible for hiding the exchange of sensory data and contextual information between cooperating smart objects. Sect. 3.6 deals with the communication layer and elaborates on our approach to adapt networking parameters according to context information. The main results of this chapter are summarized in Sect. 3.7.

3.1 Motivation and Background

3.1.1 Why Smart Objects should Cooperate with Each Other

The severe resource restrictions of computer-augmented everyday artifacts imply substantial problems for the design of applications in smart environments. In the following, we argue that these problems can in part be overcome if smart objects are able to cooperate with other augmented artifacts in their vicinity. Cooperation can make it easier for smart objects to realize their services chiefly for two reasons: (1) According to the vision of pervasive computation embedded into everyday environments, there are likely to be many augmented artifacts in range of each other. Nearby smart objects therefore constitute a reliable source of computing resources for any augmented item. (2) People in smart environments expect services offered by smart objects to be context-aware. In other words, augmented artifacts must be able to determine the real-world situation of users and adapt their services accordingly. Context recognition often requires cooperation because the sensors of a single object can perceive only a limited subset of peoples' real-world environment. Considering sensory data from other objects during the context recognition process can therefore considerably increase the accuracy of its outcome.

In summary, the limited amount of computational resources (e.g., a highly restricted memory capacity) and the necessity to provide context-aware services force smart objects to cooperate with each other. Cooperation between augmented items exploits the envisioned presence of ubiquitous computation in Pervasive Computing environments; in the next chapter it is shown how smart objects can make use of the heterogeneity inherent in such settings by cooperating with other kinds of computing devices.

3.1.2 Background

The work presented in this chapter is largely influenced by the ideas put forward in the MediaCup [Med04] and TEA [TEA04] projects, and was carried out while the author was working in the Smart-Its [Sma04b] project. All of these projects have in common that active sensor-based tags are attached to objects in order to realize context-aware services. This is also the scope of our work: we assume that smart environments are populated by many smart objects that have previously been augmented with active sensor-based computing platforms. Such smart objects can perceive their environment through sensors, act autonomously because they have an independent power supply, and cooperate with other smart objects by means of short-range wireless communication technologies. Based on these capabilities, smart objects can provide services to nearby people on their

own, without relying on a backend infrastructure. Consequently, augmented items (although they can – and, in fact, should – cooperate with backend services) have the main responsibility for realizing services associated with an augmented everyday thing. This is the major difference to other work on smart objects. In [Coo04, FFK+02, WFGH99], for example, smart objects merely provide an identification to background infrastructure services, which then implement the application logic and the interactions with users on behalf of augmented items.

In the following, we present background information on each of the separate layers of our architecture for providing collaborative context-aware services: the application layer, the context layer, the layer for inter-object cooperation, and the communication layer.

Programming Models. Several context frameworks (see Sect. 2.3.2) provide programming abstractions for processing sensory data and context information stemming from different data sources. For example, the Context Toolkit [SDA99] provides aggregators, interpreters, and context widgets for dealing with context. In the Gaia context infrastructure [RC03, RCRM02] first order predicate calculus is used to describe the processing of sensory data. Other researchers, e.g. Brown [Bro96], describe context-aware applications in the form of XML files. These approaches, however, are difficult to translate to smart everyday objects, whose resource-restricted sensor node platforms are usually programmed in the low-level programming language C. Any form of program description must therefore eventually be transformed into a C file for a smart object's microcontroller. Using XML files, program descriptions would easily become complex and difficult for an application programmer to understand. Prolog interpreters, as they are used in the Gaia context infrastructure, generally have high resource demands. Programming abstractions proposed in the Context Toolkit, such as aggregators and interpreters, could be realized in C but would be more difficult to use compared to implementations in Java. Additionally, programs would be entirely written in C, which does not provide high-level programming constructs like object-oriented programming languages such as Java. This is a major difference to our work, where parts of an application – dealing with the context recognition process or access to sensors – are not specified in C, but in a higher-level description language. The goal is to help programmers in realizing the context recognition process and to automatically generate low-level C code for a smart object's microcontroller.

In a recent paper, Whitehouse et al. [WSBC04] propose a neighborhood programming abstraction for sensor networks that allows sensor nodes to share their state with a number of previously selected nodes. It is shown that based on this programming abstraction, distributed algorithms that operate on a set of tightly-coupled sensor nodes can be expressed in a more compact form requiring fewer lines of code. In our work, we also group cooperating entities and enable them to share and access each other's data. However, the programming concept presented as part of our work is tailored towards the needs of context-aware applications. Cooperating nodes appear to applications as a single entity, and the origin of data and resources can be entirely hidden from applications.

Context. As already mentioned, context recognition is one of the core reasons for smart objects to cooperate with each other. Section 2.3 gives a short survey of today's approaches to deriving and modeling context. Regarding the scope of our work, the shortcomings of these approaches are that they either rely on a constantly available backend infrastructure to perform the actual context recognition or derive context on

a single embedded device. In contrast, the work presented in this chapter focuses on context recognition that involves collections of resource-restricted smart objects. With respect to other work on context recognition with smart objects, it tries to be more general in that it covers all the major layers that are necessary to realize context-aware services, from low-level communication issues to a programming model.

Shared Data Spaces. In our approach, shared data spaces – such as tuplespaces – play an important role for cooperating objects because they provide a platform for the exchange of sensory data and context information. Tuplespaces were originally used in the field of parallel programming, where they serve as centralized data storage accessible by various parallel processes. Such processes write tuples to and read data from the space. The tuplespace helps to coordinate different processes on a single computer system and decouples inter-process communication in time and space [GC92]. Because of the latter properties, the tuplespace model is also of significance in distributed systems. Distributed tuplespaces distribute the originally centralized tuplespace structure among different nodes. There are distributed tuplespace implementations, such as L^2imbo, that operate on fast IP-based networks. In contrast, our implementation focuses on embedded devices with severe resource-restrictions and dynamic networks that run on top of short-range wireless networking protocols.

The Stanford Interactive Workspaces [JF01] project introduces a tuplespace based infrastructure layer for coordinating smart objects in a room. This tuplespace is centralized and runs on a server in range of smart objects, whereas we distribute the tuplespace among smart objects and do not assume that there is always a stationary server in wireless transmission range.

Context-Aware Communication. Katz [Kat94] points out that "mobile systems must be able to detect their transmission environment and exploit knowledge about their current situation to improve the quality of communications." The variety of sensors carried by smart objects makes it possible to determine the current situation in which communications take place with unprecedented accuracy. By considering such context information in the design of communication algorithms, they become more aware of their environment, and are therefore better suited for computing settings where the real-world surrounding of nodes affects their communication.

Information about the location of nodes improves the performance of routing protocols in mobile ad hoc networks [BCSW98]. Such location information is usually derived on each single unit autonomously (e.g., by means of a single GPS sensor attached to it). Consequently, determining the location of nodes requires no or only little data exchange, whereas we concentrate on cooperative environments where the process of deriving context information requires tight collaboration.

Beigl et. al [BKZ+03] reports on a networking protocol stack for context communication, and suggests considering situational context in order to improve its efficiency. In their approach, contexts such as the battery level, the current processor load, and link quality serve as indicators for adapting networking parameters. We do not focus on such virtual-world parameters, but are more interested in adaptiveness based on the real-world situation of objects. Furthermore, we concentrate on providing the distributed infrastructure and high-level services that allow for this kind of adaptiveness.

3.2 Solution Outline

The goal of this chapter is to show how smart objects can provide context-aware services in cooperation with other augmented items in their vicinity. The solution of choice to achieve this goal is to form groups of collaborating nodes that share each other's resources by means of an infrastructure layer for inter-object cooperation. This infrastructure layer hides the distribution of sensory data between collaborating objects so that they appear to applications as a single node accumulating the resources of all participating smart objects. Grouping is not only a logic concept for writing applications, but has also its physical representation on the networking layer. In other words, grouping serves as a logic programming abstraction to ease implementation, but cooperating nodes are also physically arranged into networking clusters. This considerably increases the efficiency of cooperation because ad hoc communication – regarding both energy consumption and time – is often the dominating cost factor in carrying out collaborative services. In our approach, we group collaborating smart objects in one networking cluster in which nodes communicate on a distinct broadcast channel. The main criteria for organizing smart objects into clusters are their real-world context and application specific information. By considering the context of smart objects in clustering nodes, the underlying networking structures can better meet the requirements of context-aware applications. Figure 3.1 illustrates our idea of grouping collaborating nodes into distinct networking clusters.

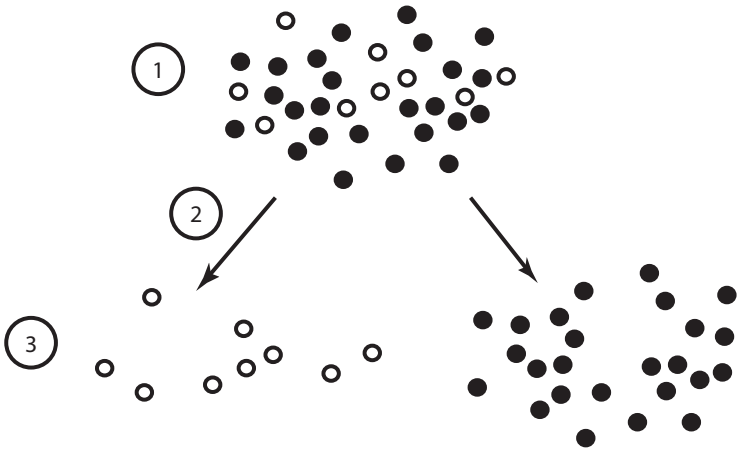

Figure 3.1: Grouping of cooperating nodes: (1) cooperating nodes in a network of loosely-coupled smart objects are (2) organized into distinct networking clusters. (3) Nodes in a cluster bundle their resources and make them available for each other's applications.

To realize collaborative context-aware services in the envisioned environments, we present a software architecture for smart everyday objects consisting of a communication layer, a layer for inter-object cooperation, a context layer, and an application layer. The main tasks of these layers are (1) to help programmers in implementing collaborative context-aware services, (2) to automatically generate code for smart objects from a high-level description of context-aware applications, (3) to facilitate the exchange of sensory data between smart objects, and (4) to provide efficient networking structures for context-aware applications. Fig. 3.2 shows how these tasks have been addressed in our software architecture. In the following, we give a more detailed overview of the different layers of our architecture, and show how they influence the design of collaborative context-aware services.

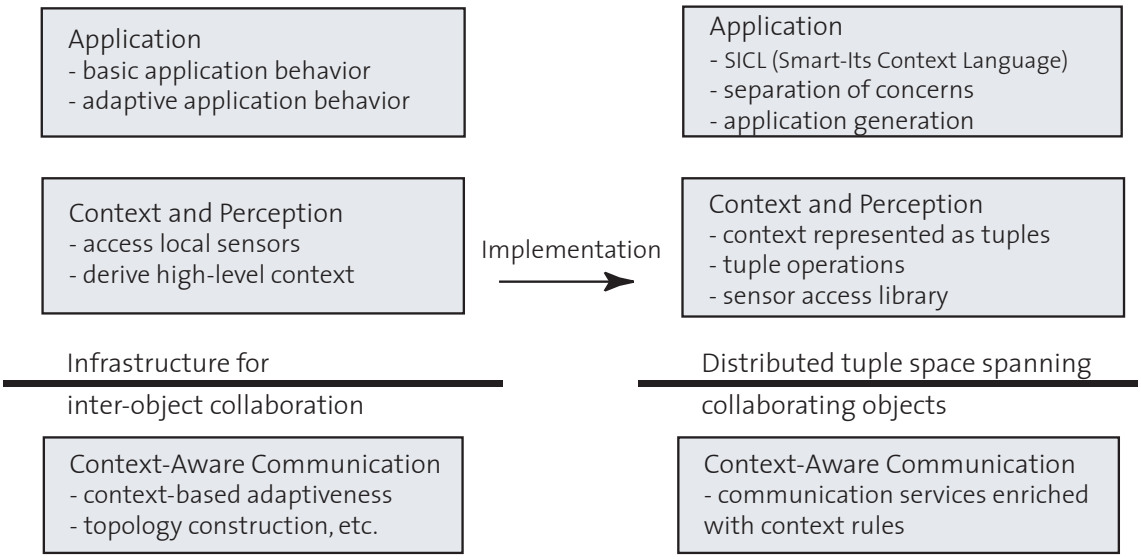

Figure 3.2: A software architecture for smart objects and its implementation.

Application Layer

The application layer supports programmers in implementing context-aware services. Generally, context-aware applications have four parts: (1) basic application behavior, i.e., functionality that does not depend on context changes, (2) adaptive application behavior, which describes how an application reacts to changing situational context, (3) a section that specifies how to derive context from local and remote sensor readings, and (4) a part that specifies how to access sensors. We have implemented the Smart-Its Context Language (SICL), which separates these concerns and encourages programmers to write more legible and structured code (cf. Fig. 3.2). Given a description of the four main parts of a context-aware service in SICL, a compiler automatically generates low-level C-code for an embedded platform. The context recognition part of an application specified in SICL usually operates on a set of cooperating objects, not on a single device. As a result, an application programmer does not have to worry about how to retrieve data from other objects or how to exchange context information. Instead, the programming model provided by SICL hides these issues.

Context and Perception Layer

The context and perception layer derives the current situational context of a smart object or that of a nearby person. How to access sensors and how to derive context information is specified in SICL as part of an application. Given an SICL description, the SICL-compiler then generates the actual code for the context and perception subsystem of smart objects. The context layer is also responsible for providing information about an object's context to the application layer, which is then able to adapt according to this information (cf. Fig. 3.2). In our current implementation, context information is represented as tuples and the derivation of higher-level context information as tuple transformations. Tuples representing sensory data or contextual information might originate from other smart objects, and the results of a tuple operation might be of potential interest to other smart objects nearby. The challenge is therefore to implement multi-sensor context awareness on sets of cooperating smart objects.

Support for Inter-Object Collaboration

We address this challenge of multi-sensor context awareness with an infrastructure layer providing cooperating entities with a distributed data space (cf. Fig. 3.2). This data space allows smart objects to share data as well as resources, and to fuse sensory data originating from different smart objects. The data space has been implemented as a distributed tuplespace. Its main purpose is to hide low-level communication issues from higher layers, and it also provides a means to group cooperating objects. As the usefulness of sensory data for deriving context usually decreases with increasing distance from the data source – e.g., a sensor that is only two meters away from a user is usually much better suited for deriving his/her context than a sensor that is 20 meters away – smart objects that collaborate to determine a certain context are often in close proximity of each other, for example in the same room or vicinity of a user. Cooperating objects can therefore be grouped into a tuplespace that operates on a dedicated wireless channel, which significantly increases the efficiency of tuplespace operations (cf. Sect. 3.5).

Communication Layer

The communication layer is responsible for low-level communication issues, such as device discovery, topology construction, connection establishment, and data transfer. As a Bluetooth-enabled device platform [BKM+04] is used in our prototypical implementation, the communication layer consists in its basic form of a Bluetooth protocol stack. Our platform for context-aware communication considerably extends this protocol stack with mechanisms that adapt networking parameters according to results from the context layer. Here, communication services contain rules for the context layer that describe how to derive contexts relevant in communication protocols, and how to change communication parameters accordingly. For example, when smart objects find out that they are in the same room (information that is derived by the context layer), the wireless network topology can be automatically changed so that communication between these devices is more efficient (cf. Sect. 3.6). We call communication services that take input from the context layer into account *context-aware*.

We have prototypically implemented the presented software architecture for cooperating smart objects on an embedded sensor node platform, the BTnodes, in order to evaluate its effectiveness on the basis of concrete experiments. In the next sections, we describe the individual components of our architecture and their interrelations in more detail.

3.3 SICL – A Description Language for Context-Aware Services

Developing context-aware services in a language that natively supports embedded platforms (usually C) often leads to unnecessarily complex and error-prone code. To address this issue, we have designed a high-level description language – the Smart-Its Context Language (SICL) – which facilitates the development of context-aware services and applications. SICL programs usually operate on a set of cooperating objects, but hide much of the complexity caused by such cooperations. In the following, we describe SICL and its underlying concepts in greater detail.

3.3.1 The Structure of an SICL Program

While developing a range of context-aware applications on embedded sensor node platforms [BKM+04, SF03a], we found that they all have a similar recurrent structure. This structure consists of four parts: (1) basic application behavior, (2) adaptive application behavior, (3) access to local sensors, and (4) context recognition. By basic application behavior, we understand the subset of a context-aware application that does not depend on context changes. For example, basic communication subroutines, code implementing sensor protocols, and code for storing data in internal data structures belong in the category of basic application behavior. In contrast, adaptive application behavior describes the context-aware adaptiveness of an application, i.e. how an application reacts to context changes. The third category deals with access to local sensors. It contains sensor access strategies describing at what intervals or in reaction to which external events what kinds of sensors are read out. Finally, context recognition makes up a substantial part of every context-aware application, especially when it involves sensory data from many different sources. The subset of an application responsible for recognizing context describes the steps that are necessary to derive high-level context information from low-level sensory data.

Unfortunately, the C programming language – which is the predominant language for programming embedded platforms – does not encourage application programmers to clearly separate the different concerns of context-aware applications identified above, which often leads to illegible, unstructured and error-prone code. Furthermore, C does not provide high-level programming constructs for describing how to derive context information from basic sensory data. As a result, C code often becomes unnecessarily complex, and tends to distract programmers from concentrating on the core aspects of recognizing context in cooperation with other nodes. On the other hand, C is a good choice for implementing the basic application behavior of a context-aware service. This is because C as a low-level programming language is well suited to manipulating internal data structures on embedded platforms. Application components encapsulating sensor protocols and access to communication primitives can also be more easily realized in a language like C.

For the above reasons, we decided to design the description language SICL, which aims at supporting programmers in the design of collaborative context-aware applications. Given a description of a context-aware service in SICL, a compiler automatically generates C code for smart objects. The description language SICL simplifies the implementation of a context-aware application in the following ways:

- It separates basic application behavior from adaptive application behavior and therefore leads to better structured code for context-aware applications.
- It allows an application programmer to specify the context-recognition process in a clear and legible way in the form of simple tuple transformations.
- It provides a concise syntax for fusing sensory data from multiple smart objects. In fact, the location of sensory data can be completely transparent for application programmers.
- The adaptive part of an application, that is, how the application reacts to results from the context recognition process, can be clearly formulated.

```
    smart object name;

    %{
    C declarations
    %}

    /* sensor access */
8:  define [search policy] [location specifier] sensor name {
        tuple type declaration;
        sensor access function;
        sensor access strategy;
    };

    /* context recognition */
15: func_1(arg_11, ..., arg_1m_1), ..., func_n(arg_n1, ..., arg_nm_n)
    ------------------------------------------------------------------
    tup_1 < field_11, ..., field_1p_1 > + ... + tup_q < field_q1, ..., field_qp_q >
                                  -> tup_q+1 < field_(q+1)1, ..., field_(q+1)p_q+1 >;

    /* adaptive application behavior */
20: tup < field_1, ..., field_p > -> func(arg_1, ..., argn);

    %%

    /* basic application behavior */
25: C code
```

Figure 3.3: The basic structure of an SICL program.

- SICL enables a programmer to specify how to access sensors.

Fig. 3.3 gives an overview of the main programming constructs of SICL, and shows how they correspond to the previously identified subparts of context-aware applications: (1) sensor access (cf. line 8 in Fig. 3.3), (2) context recognition (cf. line 15 in Fig. 3.3), (3) adaptive application behavior (cf. line 20 in Fig. 3.3), and basic application behavior (cf. line 25 in Fig. 3.3).

As mentioned above, SICL aims at hiding much of the complexity involved in realizing cooperative context-aware service from an application programmer. The SICL program depicted in Fig. 3.4, for example, consists in its original form – i.e., including the code blocks for the basic application behavior omitted in Fig. 3.4 – of 40 lines of code. The SICL compiler, however, generates 540 lines of C code from the given SICL program. This indicates that on the communication and context recognition levels, the evaluation of context recognition rules operating on a set of cooperating objects and the implementation of different sensor access strategies can get complicated. Of course, it would be possible to reduce some of the complexity involved by implementing C library functions that hide some of these lower level issues. However, an application programmer would still have to deal with the internal data structures used to represent context information and with many of the error conditions that can occur when smart objects cooperate with each other. We would therefore like to argue that SICL provides a concise description of the context recognition process operating on a collection of cooperating objects. This has also been the main rationale for designing the SICL language: based on our own experience in implementing context-aware applications [SF03a], handling the representations of context information and communication issues using C code can be highly complex. Introducing a description language such as SICL is therefore meant to make it easier for application programmers to realize collaborative context-aware services.

```
    smart object egg_box;

    %{
    #include "sensor.h"
    #include "gsm.h"
    %}

8:  define callback sensor acceleration {
        <type, subt, leng, accX, accY>;
        void get_and_write_accel();
        /* read out as often as possible */
        best effort;
    };

15: eggs_damaged(x,y)
    ============================================
    acceleration<t, s, l, x, y> => damaged<x, y>;

19: damaged<x, y>' -> send_damaged_message_to_phone(x, y);

    %%

23: void send_damaged_message_to_phone(s32* accel_x, s32* accel_y) {
        ...
    }
```

Figure 3.4: A very simple SICL program that monitors the state of a smart product by means of an acceleration sensor and sends a message to a mobile phone when the product has been damaged.

Example Application. Fig. 3.4 depicts a concrete example of an SICL program; this example is also used in the remainder of this chapter to illustrate further SICL concepts. The purpose of the underlying application scenario is to monitor the state of a smart product during transport and storage, and to notify its owner by sending a message to the owner's mobile phone when the product has been damaged (cf. Chap. 4 for a detailed description of how smart objects can access the capabilities of nearby mobile user devices). In order to better illustrate the basic idea of our application, we have chosen an egg carton as an example of a fragile product whose status needs to be monitored. Products are augmented with active sensor-based computing platforms that are equipped with accelerometers to detect when a product is damaged (cf. Fig. 1.1). In SICL, sensor access statements (cf. Fig. 3.4 line 8) specify when and how to read out sensors. In the example, accelerometers are accessed in a best effort strategy, i.e., they are sampled as often as possible. Because sensory data and context information are represented as tuples, the context recognition process is specified in the form of tuple transformations. Tuple transformations describe how context information is derived from sensory data and low-level context information. In Fig. 3.4 line 15, for example, the context *damaged* is derived from an acceleration sample if the measured acceleration values exceed a previously defined threshold. An adaptation rule (see Fig. 3.4 line 19) describes the adaptive part of our application and specifies an application's reaction to context changes. Here, a message is sent to a mobile phone when the context *damaged* has been derived. Functionality that does not depend on context changes – i.e., the basic application behavior – is given in ordinary C code. In the example application, the function for sending a message to a mobile phone is part of the basic application behavior (cf. Fig. 3.4 line 23).

3.3.2 A Programming Abstraction for Collaborative Context-Aware Services

Collaborative context recognition requires sensory data and contextual information to be retrieved from remote objects. For example, if a smart artifact is interested in a certain context but is not equipped with all the sensors necessary for detecting this context on its own, it must obtain corresponding sensory data from nearby objects. The description language SICL hides the distribution of sensory data, and hence allows programmers to abstract from the communication issues underlying a collaborative context recognition process. Using SICL, the locations at which sensory data are stored and retrieved can become completely transparent for application programmers.

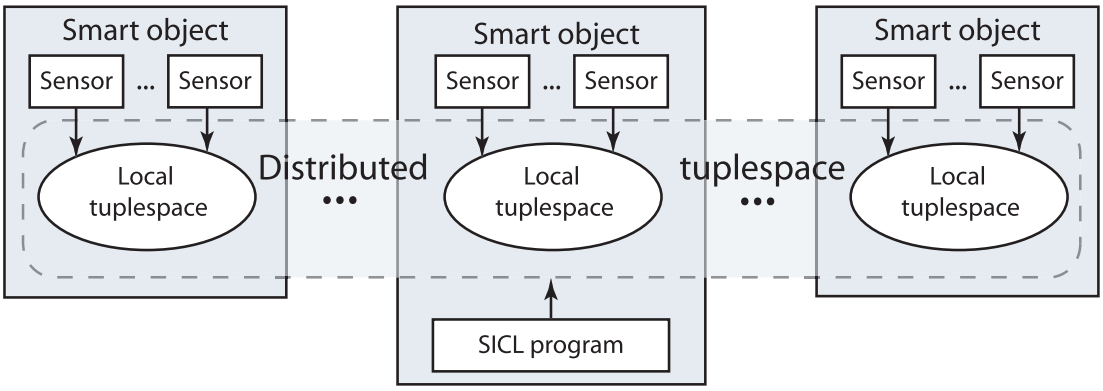

Figure 3.5: Cooperation in SICL: smart objects embed sensory data into tuples and put them into their local tuplespaces; an SICL program operates on a federation of such spaces – the distributed tuplespace established by cooperating smart objects.

This is achieved as follows. The rules of an SICL program describing the context recognition process (see Fig. 3.4 line 15) operate on tuples that represent sensory data and context information. When evaluating these rules, matching tuples are by default not only searched for on the local node carrying out the transformation, but are retrieved from a data structure shared by all the nodes it cooperates with. Sensor access statements not only describe when to read out sensors, but also specify how sensor samples are embedded into tuples and stored in the shared data space. This ensures that sensory data are accessible from all cooperating nodes (see Fig. 3.5). Considering again the example in Fig. 3.4, sensory data used for deriving the context *damaged* could also be retrieved from other nearby objects. The status of the product could, for example, also depend on the temperature of the room it is currently in. If the product itself is not equipped with a temperature sensor, but there is another smart object nearby that can provide the desired information, the following rule would automatically consider the temperature reading of the remote node:

```
eggs_temperature_damaged(temp, status)
=====================================
temperature<temp> => damaged<status>;
```

The shared data structure from which sensor tuples are retrieved has been implemented as a distributed tuplespace that is established by cooperating objects. Each node contributes a subset of its local memory to the shared data space and allows cooperating

objects to store and retrieve data from it. This local space is called the local tuplespace of that node. The shared data space is a confederation of these local spaces, and operations on the shared data space involve all local spaces.

3.3.3 Basic SICL Programming Constructs

Sensor Access

Raw sensory data constitute the basic input for every context recognition process. In SICL, sensor access statements describe how to deal with local and remote sensors. The basic syntax of a sensor access statement is as follows:

```
define [search policy] [location specifier] sensor name {
    tuple type declaration;
    sensor access function;
    sensor access strategy;
};
```

The *location specifier* in a sensor statement can be used to distinguish between local and remote sensors. In case of remote sensors, a sensor access statement only contains a tuple type declaration, and no access function or access strategy. However, the type of both local and remote sensors must be explicitly specified in order for the SICL compiler to ensure type safety. If the location specifier is omitted, a sensor is assumed to be local.

In our programming model, sensor readings are embedded into tuples and used in transformations to derive new tuples that eventually represent higher-level context information. The *search policy* specifies which sensor tuples are considered in tuple transformations. In the program in Fig. 3.4, for example, the keyword *callback* indicates that only the most recently generated acceleration tuple is used in tuple transformations.

As sensor tuples are not generated in tuple transformations, their type – i.e., the type of every field in a sensor tuple – must be explicitly specified. This allows the SICL compiler to check for type safety in transformation rules, and to automatically derive type information for newly generated tuples. The *tuple type declaration* in a sensor access statement lists the types of all the components in a sensor tuple.

Following the type declaration of a sensor tuple, a function must be named that is called in the program generated from an SICL description in order to actually access the sensor. As low-level drivers for sensors are written using the C programming language, this is a C function that has three main responsibilities: (1) it reads out the sensor, (2) it embeds the sensory data in a tuple conforming to the type specified, and (3) it writes the tuple into the data space in order to share the embedded sensory data with other nodes.

As the last component of a sensor access statement, *sensor access strategies* determine when to read out sensors. With respect to the time period between consecutive samplings and the time at which to access sensors, we distinguish between the following strategies:

1. **Best effort.** Some sensors, for example accelerometers and microphones, monitor environmental parameters that can change rapidly in relatively short time frames. In such cases, the *best effort* strategy allows an application to access sensors as often as technically feasible in order to monitor environmental changes accurately. The above-mentioned function for reading out a sensor is thereby called whenever possible.

2. **Fixed.** In this sensor access strategy, there is a fixed time period between consecutive sensor samplings. For example, the statement *every 2000 ms* means that the corresponding sensor access function is scheduled to be called every two seconds. There are two ways to implement the fixed strategy on a microcontroller. The first, and more precise one, is to use a timer and to access the sensor from the corresponding interrupt handler. This ensures that an application accurately conforms to the given time interval. In our prototypical implementation, however, we have implemented a scheduler that repeatedly checks for functions that need to be executed. The accuracy of such an approach is influenced by the execution time of other scheduled functions.

3. **Random interval.** In order to avoid unwanted synchronization effects between nodes, it is sometimes necessary to vary the time period between consecutive sensor accesses. Such a behavior is supported by the random interval strategy, where a time interval is specified out of which a value is randomly chosen. The statement *random [1000, 2000] ms* is an example of this where the time between consecutive readings is randomly distributed between one and two seconds.

4. **Event-driven.** In contrast to the need for regular sensor access, some sensors only need to be read out under special circumstances – i.e., in case of certain internal or external events. Event-driven sensor access can considerably reduce the number of times a sensor needs to be sampled. It can therefore contribute towards reduced energy consumption, as the process of sampling sensors often consumes significant amounts of energy (cf. Tab. 2.3). Hence, especially when designed for resource restricted smart objects, applications should aim at accessing sensors as seldom as possible. Event-driven sensor access is specified using adaptation rules, and not directly in sensor access statements.

The resource restrictions of smart objects play an important role in scheduling the access to sensors. It should be kept in mind that reading out sensors needs both energy and time. The energy consumption can be reduced using the previously mentioned event-driven strategy for sampling sensors. But the time constraints on embedded platforms can also make an accurate monitoring of environmental parameters difficult. Especially if sensors need to be read out frequently with high sampling rates, smart objects may not have sufficient time to communicate with other objects. This can impair the performance of collaboration protocols if they depend on responses from all devices cooperating with each other. A solution to this problem is to use two-processor tagging technologies (cf. Sect. 2.2.3), where communication and sensor related tasks are handled by two separate microcontrollers. From the perspective of an application, sensor sampling on such systems can be carried out asynchronously. Consequently, an application can issue a request for sensory data that is transmitted to and executed by the microcontroller responsible for sampling sensors. As requests for sensory data are carried out on a different microcontroller than the application, it can still react to communication requests while sensors are sampled. This is a significant advantage for collaborating smart objects because cooperation naturally requires nodes to communicate with each other. With respect to the performance of collaboration protocols it is therefore a considerable drawback if nodes must wait for responses from other objects because they are blocked while reading out sensors. However, the time needed to retrieve a single sensor sample is usually very short, so that it does not significantly affect the performance of communication protocols. The

problems increase if sensors have to be sampled at high rates of frequency over long time periods in order to extract characteristic features from a data stream. In Sect. 3.6.4, we discuss an example where microphones are read out at 40 kHz over time periods of one second in order to extract a sensor feature indicating the level of noise in a room. Here, it is advantageous that the application does not block while obtaining the sensor feature.

The BTnodes together with the TecO sensor boards [BZK+03], which were used in our experiments, are a two-processor tagging technology. Hence, both the BTnodes and the sensor boards are equipped with an individual microcontroller. The main part of the application logic resides on the microcontroller of a BTnode.

Context Recognition

Context recognition in SICL is expressed in the form of tuple transformations. Tuple transformations describe how tuples are transformed and new tuples are created. This reflects the process of deriving high-level context information by fusing sensory data from different nodes. In SICL, it does not matter at which specific smart object a tuple has been created because the infrastructure layer for inter-object collaboration makes tuples available to all collaborating nodes. That is, the context recognition process operates on a shared data structure established by cooperating objects; newly generated sensory data and derived context information are embedded into tuples and shared by means of this distributed data space. The basic structure of a transformation rule is as follows:

$$\frac{func_1(arg_{11}, \ldots, arg_{1m_1}), \ldots, func_n(arg_{n1}, \ldots, arg_{nm_n})}{tup_1\langle field_{11}, \ldots, field_{1p_1}\rangle + \ldots + tup_q\langle field_{q1}, \ldots, field_{qp_q}\rangle \rightarrow tup_{q+1}\langle field_{(q+1)1}, \ldots, field_{(q+1)p_{q+1}}\rangle};$$

Let us first consider the expression under the horizontal separator line. The left-hand side of this expression lists the tuples that must be available for the rule to be executed, and the right-hand side specifies the tuple that is generated when the rule fires. More specifically, the given tuples are tuple templates which are substituted by matching tuples during the evaluation of a rule. By means of this substitution, the fields of tuples on the left-hand side are given concrete values, which are used to generate the tuple on the right-hand side.

Above the horizontal separator, a set of *restricting functions* can be specified that operate on the fields of left-hand side tuples. Restricting functions serve two purposes: First, the tuple transformation is only carried out for a concrete set of left-hand side tuples if all of these functions return true. Second, the given functions, which must later be provided in C code, can be used to calculate new fields for the tuple on the right-hand side. If no functions are given, a transformation rule is executed whenever matching tuples for the templates on the left-hand side are available.

The following is an example of a transformation rule in SICL:

```
is_critical(x, y, c, status)
=============================================================
acceleration<x, y> + criticalAccel<c> => damaged<status>;
```

The example belongs to the application introduced at the beginning of this section, and is designed to calculate the status of a product based on accelerometer measurements. In the above rule, a tuple representing the context *damaged* is derived out of

a *criticalAccel* tuple and an *acceleration* tuple. How *acceleration* tuples are generated is specified by sensor access statements; the *criticalAccel* tuple contains only one field representing an upper bound for acceleration values and is written into the shared data space during initialization. If acceleration measurements are above the *criticalAccel* value, the tagged product has been damaged. The function *is_critical* is a C function returning a boolean value and checks if the measured acceleration is above the threshold. The tuple transformation rule only fires if *is_critical* returns true. The function also generates a new field, *status*, that is used in the newly generated result tuple. The variable *status* could for example indicate the level of damage inflicted on the augmented product.

The program generated from an SICL description continuously evaluates rules. As a result, the problem arises that if a rule fires once, it will also fire in all consecutive evaluations. To address this problem, SICL offers a syntax to mark tuples that are to be deleted after a rule has been successfully executed.

The types of all tuple templates in transformation rules must be given either implicitly or explicitly. The type of tuples can be explicitly declared either in sensor access statements or in tuple type declarations (see appendix A for a complete syntax of SICL). If the type of a tuple is not explicitly declared, it must be possible for the SICL compiler to derive type information from other tuples in a transformation rule. Otherwise, the SICL compiler will not accept a program because it does not conform to the context conditions of SICL.

Adaptive Application Behavior

Adaptation rules represent the adaptive part of a context-aware application in that they describe how an application reacts to context changes. An adaptation rule consists of a tuple template on the left hand side, which stands for a certain situational context, and a function on the right hand side that is executed when a corresponding context is derived:

$$\boxed{tup\langle field_1, \ldots, field_p\rangle \rightarrow func(arg_1, \ldots, argn);}$$

Basic Application Behavior

The basic application behavior of a smart object is provided in form of ordinary C code. SICL allows programmers to embed corresponding C declarations and definitions into an SICL program. The SICL compiler generates C code from the other parts of an SICL description, which is then compiled and linked with the C code representing the basic application behavior.

3.3.4 Improving the Effectiveness of Tuple Transformations

SICL provides programming concepts for accelerating the evaluation of transformation rules. In the following, we consider two mechanisms to improve the efficiency of such rule evaluations: (1) reducing the search space for tuples, and (2) push-based reception of tuples.

Reducing the search space

The search space for tuples can be reduced by restricting the scope of tuple declarations to the local tuplespace of the node carrying out an SICL program. As a result, when

a transformation rule is evaluated, tuples are only searched for on the local node. As we will see in Sect. 3.4, this has two core advantages: First, it reduces the amount of communication and therefore saves energy. Second, it accelerates rule transformations because the time needed for local tuplespace operations is significantly lower than the time needed for operations on a distributed tuplespace.

Restricting the search scope is particularly sensible with respect to certain types of sensory data. Accelerometer samples, for instance, are useful to characterize the situation of the object they are attached to, but are less meaningful for remote objects. The egg carton application introduced above is an example of this observation because only local accelerometer samples are useful to detect the current status of the augmented egg box. Whether it is reasonable to restrict the search space for tuples therefore largely depends on the specific application and the kind of sensors used. In the case of temperature sensors or microphones, for example, samples measured at remote nodes might be as valuable as local sensor readings.

The scope of tuples in SICL can be restricted in individual tuple type declarations and in sensor access statements. The keyword *private* indicates that the scope of a tuple is bounded to the local tuplespace:

```
define private sensor acceleration { <accX, accY>; ... };
define private tuple criticalAccel { <Trsh>; };
```

Push-based reception of tuples

By default, transformation rules are continuously evaluated by substituting tuple templates from the left-hand side of a rule with matching tuples from the distributed data structure. After substitution a check is made as to whether the rule conditions hold, i.e., whether all restricting functions return true for a given set of instantiated tuples. Repeated requests for tuples on the shared data space can thereby cause an unnecessary communication overhead if matching tuples change rarely. This is because consecutive requests would repeatedly return the same or no results.

To address this issue, we introduced the concept of *callback* tuples. If a tuple is declared as a callback tuple, the node carrying out an SICL program registers a callback for the given tuple template at all the nodes it cooperates with. If a tuple of the specified type is then remotely generated, it is automatically sent to the node that initially registered the callback. As a result, callback tuples are especially useful for sensory data or context information that is generated or changes rarely. Here, they can considerably reduce the amount of communication necessary to evaluate a transformation rule, because a node executing an SICL program does not need to query for such tuples actively. On the other hand, if callback tuples are generated frequently, remote nodes could send matching tuples with such high rates that it would be difficult for a smart object to process the corresponding data. If a tuple is declared as a callback tuple, only the most recently received instance of a matching tuple is considered in a transformation rule. This is another major difference to non-callback tuples. The following depicts an example of declaring *callback* tuples:

```
define callback tuple criticalAccel { <Trsh>; };
private callback accelAlarm;
```

In line 1, *criticalAccel* is declared as a callback tuple because the threshold for acceleration measurements above which an augmented product is damaged usually changes rarely. Consequently, in a tuple transformation involving a *criticalAccel* tuple, it would not make sense to waste energy by searching repeatedly for a matching tuple in the shared data structure. As can be seen in line 2 of the above example, *private* and *callback* declarations can also be combined with each other. As a result, the callback for a private tuple is only registered on the local tuplespace of the smart object carrying out the SICL program.

3.4 The Context Layer

The context recognition process described in an SICL program must be translated into code that is executed on smart objects. The context layer is then responsible for actually accessing sensors and for carrying out the tuple transformations previously specified in SICL. It manages the retrieval of sensory data and contextual information from remote nodes, derives new contexts, and schedules the access to local sensors. In this section, we will show how context descriptions in SICL are transformed into actual instructions for the context layer of smart objects.

3.4.1 Preliminaries

In our description of the context layer we use the following notations. $N = \{n_1, \ldots, n_m\}$ is a set of cooperating smart objects with $m \in \mathbb{N}$ and $m \geq 2$. We refer to the local tuplespace of a node $n \in N$ as $L^{(n)}$, and to the distributed tuplespace shared by all nodes in N as $D^{(N)}$. A tuplespace T is a multiset of tuples. A tuple $t = (f_1, \ldots, f_p)$, $p \in \mathbb{N}$, $p \geq 1$ is an array of typed fields. A typed field $f = (t, v) \in \mathbb{N} \times \mathbb{N}$ is a pair consisting of a type and a value. Furthermore, $type : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is a function that returns the type of a field, and $value : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ a function that returns the value of a field. If $f = (t, v)$ is a field, then $value(f) = v$ and $type(f) = t$. A field f is formal if and only if $value(f) = 0$. A tuple $t = (f_1, \ldots, f_p)$ is formal if and only if $\exists i \in \{1, \ldots, p\} : value(f_i) = 0$. Formal tuples are also referred to as tuple templates. A non-formal tuple $t_1 = (f_{11}, \ldots, f_{1p_1})$ matches a tuple template $t_2 = (f_{21}, \ldots, f_{2p_2})$ if and only if $p_1 = p_2 \wedge \forall i \in \{1, \ldots, p_1\} : (type(f_{1i}) = type(f_{2i}) \wedge (value(f_{2i}) = 0 \vee value(f_{1i}) = value(f_{2i})))$.

We distinguish between the following operations on a tuplespace: *scan*, *count*, *write*, *take*, *read*, and *consumingScan*. The function $scan_T(templ)$ returns all tuples matching the given tuple template *templ* on the tuplespace T. $scan_T(templ) = \{t \in T : t$ matches $templ \wedge t$ is not formal$\}$. $scan_T$ returns a multiset, i.e., elements can occur more than once. $count_T(templ)$ returns the number of tuples in T matching the template *templ*. $count_T(templ) = |scan_T(templ)|$, where again multiple occurrences are separately counted. $write_T(tuple)$ stores the tuple *tuple* into the tuplespace T: $T := T \cup \{tuple\}$. Remember that T is a multiset and that therefore multiple occurrences are relevant. $take_T(template)$ removes one tuple matching *template* from the shared data structure T and returns the matching tuple as result; if there is no matching tuple, $take_T(template)$ does not have any effect on the data structure T. $read_T(template)$ returns exactly one tuple matching *template* from the data space T if there is at least one matching tuple in T. $consumingScan_T(templ)$ has the same effect as $scan_T(templ)$ but also removes all tuples found from the data structure T.

In the following, we present the algorithms for evaluating tuple transformation and adaptation rules as they are carried out by the context layer. The input for the context layer consists of (1) a list of tuple transformation rules, (2) a list of adaptation rules, and (3) the sensor access strategies previously specified in an SICL program. The SICL compiler is responsible for translating the description of a context recognition process given in SICL into instructions for the context layer of a smart object (cf. Fig. 3.6). We refer to the list of tuple transformation rules as $(r_1, \ldots, r_p), p \in \mathbb{N}, p \geq 1$. A transformation rule $r_k, k \in \{1, \ldots, p\}$ has the form $r_k = t_{k1} + \ldots + t_{kq_k} \rightarrow t_{k,q_k+1}, q_k \in \mathbb{N}$ and a list of associated restricting functions $F_k = (f_{k1}, \ldots, f_{k,s_k}), s_k \in \mathbb{N}$. The list of restricting functions F_k for a rule r_k can be empty.

The context layer also possesses information about the type of tuples, and can therefore determine whether a tuple has been declared as a *private* or *callback* tuple. The SICL compiler ensures that the type information of all tuple fields are known. For any transformation rule $t_1, \ldots, t_q \rightarrow t_{q+1}$ the context layer therefore knows the types of all fields in $t_1, \ldots, t_{q+1}$. It can hence use these tuples as tuple templates to issue queries on the underlying shared data structure. The same is true for adaptation rules.

Besides type information, the context layer also knows about the field identifiers used in transformation rules to describe which fields of left-hand side elements are used to generate resulting tuples. Consider the following simple SICL example:

```
define tuple temperature { <temp>; };
define tuple acceleration { <accx, accy>; };
temperature<t> + acceleration<x,y> => temp_accel_pair<t, x>
```

Here, t, x, and y are field identifiers. They determine how the fields of a concrete *temperature* and *acceleration* tuple are used to generate a *temp_accel_pair* tuple. It is important to understand the difference between field identifiers and type information. Type information is either explicitly given in tuple declarations, or implicitly derived by the SICL compiler. In the above example, the type information of *temperature* and *acceleration* tuples is explicitly stated in lines 1 and 2. If there is no explicit declaration of the tuple *temp_accel_pair*, its type information would be generated from the types of *temperature* and *acceleration*. As a result, the deduced type for *temp_accel_pair* would be $\langle temp, accx \rangle$.

In the following, a node $n \in N$ is the node carrying out the SICL program that we consider; each of the cooperating nodes in N, however, can execute its own SICL program.

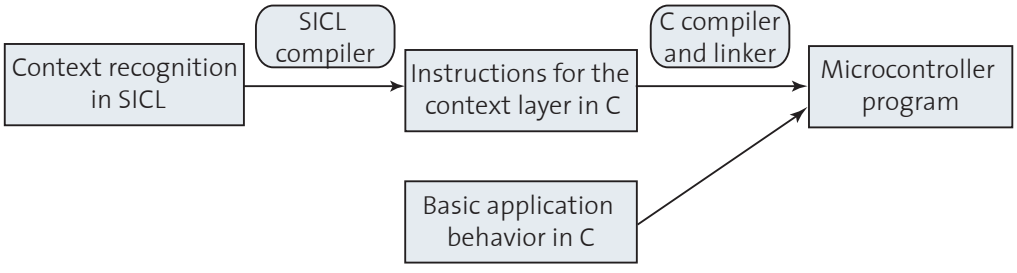

Figure 3.6: Given a description of a context-aware service in SICL, the SICL compiler generates C code that is translated into a program for a smart object's microcontroller.

3.4.2 Initialization

Before a smart object starts evaluating tuple transformation and adaptation rules, it initializes its context layer. A core responsibility of the initialization routine is to register callbacks on the shared data structure for all tuples that have been previously specified as *callback* tuples in SICL (see algorithm 1). As a result, whenever a matching tuple is generated at one of the cooperating smart objects, this tuple is transmitted to the augmented item that registered the callback. For every callback tuple there is an internal data structure that contains a copy of the last occurrence of a tuple matching the callback template. In other words, the callback function associated with a callback template merely copies matching tuples received from other objects into this internal data structure.

During initialization, the context layer also registers functions for accessing sensors at the scheduler of a smart object. The scheduler is responsible for invoking functions at the time specified (e.g., every 2000 milliseconds) or as often as possible. This functionality is used to implement the previously-introduced sensor access strategies.

Algorithm 1: Initialize callbacks

Input:
$(r_1, \ldots, r_p)$: list of tuple transformation rules
N: set of cooperating nodes
$n \in N$: node carrying out the SICL program

```
for k := 1 to p do
  for i := 1 to q_k do
    if t_ki is a callback tuple then
      if t_ki is a private tuple then
        register callback for t_ki at local tuplespace L^(n);
      else
        register callback for t_ki at tuplespace D^(N);
      end if
    end if
  end for
end for
```

3.4.3 Tuple Transformations

After initialization, the context layer continuously evaluates the tuple transformations and adaptation rules contained in an SICL program. Rules are evaluated in the same order as they are given in the SICL description.

The algorithm responsible for evaluating transformation rules (see algorithm 2) first classifies the tuples of a rule into three different categories: callback tuples, private tuples, and tuples that are neither private nor callback tuples. Afterwards, it first searches for matching callback tuples, second for private tuples, and only then does it consider the remaining tuples. The reason for this approach is that the effort required for searching matching tuples increases from category to category: When tuples associated with a registered callback are generated at any of the cooperating nodes, a copy of these tuples is transmitted to the object carrying out the SICL program and stored in its above-mentioned local data structure. Hence, during rule evaluation only this internal

data structure needs to be checked, which takes place very quickly and does not require additional communication.

Algorithm 2: Evaluate tuple transformation rule

Input:
$t_1 + \ldots + t_q \rightarrow t_{q+1}$: tuple transformation rule
$F = (f_1, \ldots, f_s)$: list of restricting functions
N: set of cooperating nodes
$n \in N$: node carrying out the SICL program

Output:
R: set of generated tuples

```
R := ∅; A := ∅; B := ∅; C := ∅;
for i := 1 to q do
  if t_i is callback tuple then
    A := A ∪ t_i;
  else if t_i is private tuple then
    B := B ∪ t_i;
  else
    C := C ∪ t_i;
  end if
end for
for all t ∈ A do
  if callback routine for t has not been called then
    return;
  else
    M_t := most recently received tuple matching t
  end if
end for
for all t ∈ B do
  if F is empty then
20:   M_t := read_{T^(n)}(t);
  else
22:   M_t := scan_{T^(n)}(t);
  end if
  if M_t = ∅ then
    return;
  end if
end for
for all t ∈ C do
  if F is empty then
30:   M_t := read_{T^(N)}(t);
  else
32:   M_t := scan_{T^(N)}(t);
  end if
  if M_t = ∅ then
    return;
  end if
end for
R := create_tuples(M_{t_1}, ..., M_{t_q});
```

Searching for data matching a given private tuple also causes little time overhaed, but requires the local tuplespace to be searched. The most costly operations for searching tuples are those executed on the shared data structure, i.e., operations that involve all cooperating objects. The algorithm for evaluating rule transformations therefore only carries out such operations if tuples matching all local and callback tuples have already been found. This helps to reduce the amount of communication necessary to check rules.

As a result, private tuples and callback tuples can also be exploited deliberately by application programmers to reduce the communication overhead and hence the energy consumption induced by rule evaluations. As long as there is a private tuple in a transformation rule with no matching tuple currently in the local tuplespace, no communication with other nodes is carried out to evaluate the rule. As callback tuples are transmitted by means of a callback mechanism, at least there is no necessity for communication during rule evaluations.

When matching tuples for all tuple templates on the left-hand side of a rule have been found, result tuples as specified by the right-hand side of a transformation rule can eventually be generated (see algorithm 3). Afterwards, result tuples are written into the local tuplespace $L^{(n)}$, and are therefore shared by means of the federation of local spaces $D^{(N)}$ with other objects. An SICL rule determines exactly how result tuples are created. This process is also more formally described in algorithms 3 and 4, which generate result tuples from a list of sets $M_1, \ldots, M_q$ containing the previously-retrieved matches for left-hand side tuples of a transformation rule.

Algorithm 3 generates all possible combinations of matching tuples, tests whether a combination satisfies all restricting functions, and generates a result tuple if the test was successful. Equal results are filtered out by the algorithm. However, every combination of tuples in $M_1, \ldots, M_q$ can potentially create a different result. An application programmer should keep in mind that this means that as many as $\prod_{i=1\ldots q} |M_i|$ results can potentially be generated in a single successful transformation. Due to the resource restrictions of smart objects, this can be a serious problem because it can cause nodes to run out of memory, which might lead to the silent deletion of tuples. To cope with this problem, the context layer provides the following concepts: (1) callback tuples, (2) restricting functions, and (3) mechanisms for reducing the number of matching tuples.

- **Callback tuples.** If there are many matches for a left-hand side tuple in the shared data space, such tuples can be declared as *callback*. This means that only the most recently generated tuple of that form is considered in a transformation. For a tuple t_i that is declared as *callback* the cardinality of M_i is therefore always either zero or one.

- **Restricting functions.** Alternatively, the amount of tuples generated can be reduced by restricting functions. If a transformation rule has conditional functions, the context layer always checks whether a combination of left-hand side tuples satisfies all of these functions. If this is not the case, no new tuple is generated for a given combination.

- **Reducing the number of matching tuples.** If there are no restricting functions associated with a rule (F is empty), the context layer retrieves only a single match for every left-hand side tuple of a transformation rule. This can be seen in algorithm 2 lines 20, 22, 30, and 32, where the *read* instead of the *scan* operation is called if the set of restricting functions is empty. Hence, for all tuples t_i from the left-hand

side of a transformation rule, M_i contains at most one element. Consequently, when the transformation rule fires, only one result tuple is generated. We have made this design decision because of the memory constraints of smart objects, and in order to reduce the amount of generated tuples. Another reason is the communication overhead involved in retrieving multiple tuples from remote nodes.

Algorithm 3: create_tuples

Input:
$t_1 + \ldots + t_q \rightarrow t_{q+1}$: tuple transformation rule
$F = (f_1, \ldots, f_s)$: list of restricting functions
$M_1, \ldots, M_q$: lists of tuples matching $t_1, \ldots, t_q$

Output:
R: set of generated tuples

```
R := ∅;
for j := 1 to q do
  currentCombination[j] := 0;
end for
do forever
  changeNext := true;
  i := 1;
  while changeNext = true do
    currentCombination[i] := currentCombination[i] + 1;
    if (currentCombination[i] ≥ |M_i| then
      if i = q then
        return R;
      else
        currentCombination[i] := 0;
        changeNext := true;
      end if
    else
      changeNext = false;
    end if
    if satisfies_funcs(F, M_1[currentCombination[1]], ...,
                       M_q[currentCombination[q]]) do
      t = create_tuple(F, M_1[currentCombination[1]], ...,
                       M_q[currentCombination[q]]);
      R := R ∪ {t};
    end if
    i := i + 1;
  end while
end forever
```

Algorithm 4 shows how a concrete combination of left-hand side tuples is used to generate a result tuple. It starts by copying the template of the resulting tuple. How to set the values of fields in the result tuple is specified by the field identifiers used in transformation rules. If the field identifier of a left-hand side tuple is equal to a field identifier of the resulting tuple, the value of the field in the result tuple is assigned the value of the field of the corresponding tuple in the current tuple combination. For the sake

of conciseness, algorithm 3 neglects the role of restricting functions in tuple generations. In our prototypical implementation, the values of fields of a resulting tuple can also be generated by a restricting function. In this case, the arguments for calling restricting functions are determined in a similar fashion by substituting function arguments with field values of tuples in the current tuple combination. Which fields are used for which arguments is again determined by comparing field identifiers.

Algorithm 4: create_tuple

Input:
$t_1 + \ldots + t_q \rightarrow t_{q+1}$: tuple transformation rule
$m_1, \ldots, m_q$: current set of tuples matching $t_1, \ldots, t_q$
$\forall i \in \{1, \ldots, q+1\} : fid_{i1}, \ldots, fid_{ip_i}$:
field identifiers for tuple t_i as given in SICL program

Output:
t: newly generated tuple

```
t := copy(t_{q+1});
for i := 1 to p_{q+1} do
  for j := 1 to q do
    for k := 1 to p_k do
      if fid_jk = fid_{q+1,i} then
        set value of ith field in t to value of kth field in m_j;
      end if
    end for
  end for
end for
```

3.4.4 Adaptation Rules

Adaptation rules describe how a smart object reacts to context changes. They consist of a tuple template, typically representing a certain context, on the left-hand side of a rule and a function on the right-hand side. The context layer evaluates adaptation rules continuously in that it repeatedly searches for tuples matching the left-hand side tuple. If a matching tuple has been found, the field values of this tuple are used as arguments for calling the function on the right-hand side of the rule. The values of which fields are used as arguments in a function call are again determined by comparing field and argument identifiers. The adaptation rule below has been taken from the example introduced in Sect. 3.3.1. It states that when a *damaged* tuple is generated, the value of its first field is used as the first argument and the value of its second field as the second argument in a call to the *send_damaged_message_to_phone* function. This is because of the matching identifiers x and y on both sides of the adaptation rule. The SICL compiler ensures that all arguments necessary to call the function on the right-hand side can be obtained from left-hand side tuples. In order to prevent adaptation rules from being continuously executed, SICL provides a syntax to automatically delete tuples after a rule has been successfully evaluated. (This is the semantics of the apostrophe in the example below.)

```
damaged<x, y>' -> send_damaged_message_to_phone(x, y);
```

3.4.5 Improving Rule Evaluations

With respect to the context layer, a major concern is to keep the cost of rule evaluations as low as possible. In Sect. 3.4.3, we have already shown how an application programmer can decrease the communication overhead by using callback or private tuples, and transformation rules without restricting functions. To further decrease the amount of communication required to evaluate rules, our prototype implementation for the BTnode embedded device platform tries to reduce the number of *scan* operations carried out on the shared data structure by replacing them with *count* and *read* operations if possible. This is because – as shown by the evaluation in Sect. 3.5.4 – *scan* operations are on average more expensive than *count* and *read* operations.

For example, in algorithm 2 lines 22 and 32 the context layer searches for matching tuples on the whole shared data structure using the *scan* operation. However, the *scan* method does not have to be called immediately. Instead, we first check by means of the *count* method whether the tuples necessary to execute a transformation rule are currently in the distributed tuplespace. Only after it is clear that all required tuples are available are the tuples actually retrieved. The reason for such an approach is that in most cases rules are evaluated much more frequently than they are executed. Our approach decreases the communication overhead for checking whether a rule can be executed because the *count* operation does not actually retrieve matching tuples (such as the *scan* operation) but merely the number of matching tuples. However, the presented variation for evaluating rules can also be counterproductive if rules are frequently executed. This is because in the presented approach both *scan* and *count* operations must be carried out in a successful rule evaluation.

Another improvement of the initial algorithm is not to search for tuples repeatedly that have been already found in a previous unsuccessful rule evaluation. To illustrate this approach, consider a transformation rule $t_1 + t_2 \rightarrow t_3$ that depends on two non-private and non-callback tuples t_1 and t_2. Now, if a matching tuple for t_1 has been found during a previous evaluation of the rule but there was no match for t_2, the context layer only searches for tuples matching t_2 in consecutive evaluations. The goal is to prevent the context layer from searching successfully for the same tuple again and again. However, the downside of this approach is that if all tuples have been found after several evaluations, it must be ensured that tuples that have been previously found are still in the shared data structure. The approach is therefore only beneficial if rules are evaluated much more frequently than they are actually executed.

Other mechanisms to decrease the effort for rule evaluations are hidden in the low-level protocols for querying the distributed data structure. Sect. 3.5 contains an in-depth description of these protocols and their evaluation.

3.4.6 Performance Evaluation

In this section we present an experimental evaluation of the context recognition layer.

Experimental Setup and Restrictions

The experimental data presented in this section were obtained based on an implementation of the context layer for the BTnode embedded device platform [BKM+04]. BTnodes communicate with each other using the Bluetooth communication standard. Nearby Bluetooth units can form a piconet, which is a network with a star topology of at most 8

active nodes. Furthermore, Bluetooth distinguishes master and slave roles. The master of a piconet is the node in the center of the star topology; slaves cannot communicate directly with each other but only via the master. The master also determines the frequency hopping pattern of a piconet and implements a TDD (time division duplex) scheme for communicating with slaves. Consequently, a slave can only transmit data when it was previously addressed by the master, and only the master can broadcast data to all units in a piconet. The frequency hopping pattern of a piconet implements a dedicated channel. As a result, multiple piconets can coexist with each other in the same area with only little interference. Multiple piconets can form so-called scatternets. In a scatternet, bridge nodes participate in more than one piconet and switch between piconet channels in a time division multiplex (TDM) scheme. The Bluetooth standard specifies a host controller interface (HCI) that can be used to access the capabilities of Bluetooth communication modules from a microcontroller. Above this layer there is a suite of protocols that deal with the transmission of data between Bluetooth devices, for example the L2CAP (Logical Link Control and Adaptation Protocol) and RFCOMM protocols. The L2CAP protocol provides connectionless and connection-oriented data services, and is responsible for data segmentation and reassembly. The RFCOMM protocol emulates a serial connection between Bluetooth devices. In the scope of this dissertation, we have prototypically implemented a Bluetooth protocol stack for the BTnodes.

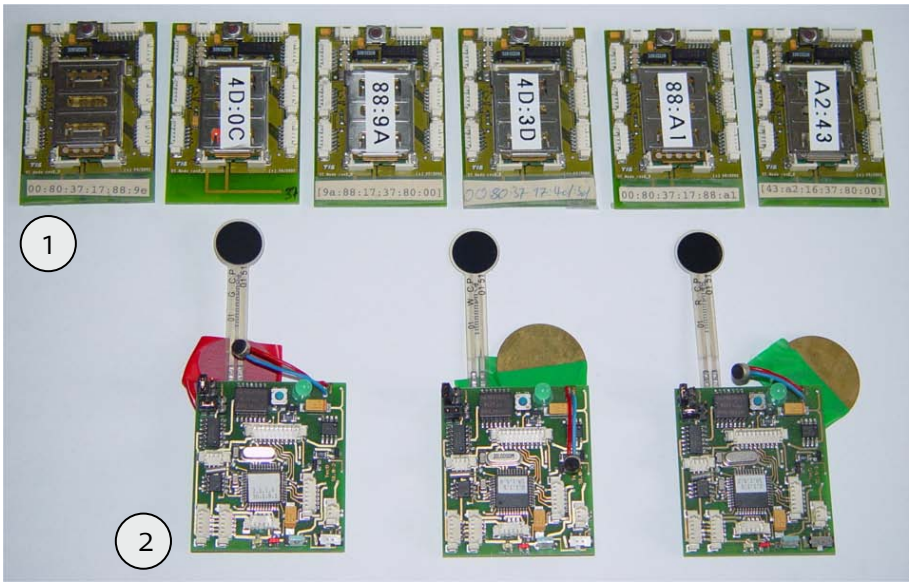

Figure 3.7: The experimental setup: a subset of the BTnodes (1) and sensor boards (2) used in our experiments.

In our experiments regarding the context layer, BTnodes that want to cooperate with each other are grouped into a single cluster. Sect. 3.5 and Sect. 3.6 explain this approach with respect to smart environments and present a corresponding algorithm for grouping nodes. As the implementation of scatternets on the BTnode revision we used is rather unreliable, producing many lost packets and unpredictable disconnections, we consider only piconet topologies in the evaluation of the context layer. As a result, cooperating nodes communicate with each other over at most two hops. Piconet topologies also have a restriction regarding the number of nodes that can participate in them. With respect to the BTnodes, only seven nodes can reliably communicate in such a topology – although

according to the Bluetooth standard a piconet can have up to eight active devices. The Bluetooth modules on the BTnodes also have other practical restrictions. For example, packets that are sent via broadcasts to other nodes in a piconet are transmitted five times; mechanisms to change that behavior are not supported by the Bluetooth modules used in our experiments although they are specified in the Bluetooth standard.

Another important restriction is that the speed of the serial line on a BTnode limits the throughput of the communication technology. Although Bluetooth modules transmit data over the air at 1 Mb/s, if the microcontroller does not send the data to the Bluetooth module at that speed, this limits throughput. On the BTnode platform, we have used a maximum speed for the serial interface of 230.4 kbit/s in our experiments. Furthermore – as also reported by other researchers [LDB03]– the general throughput of the BTnodes using small L2CAP packets is significantly below the theoretically achievable throughput. If data are streamed, we achieve data rates of about 37.4 kbit/s with our implementation. Considering all these issues, it should be kept in mind that the measurements in this section present results from a concrete practical system. (Please refer to Sect. 3.5 for a comparison of experimental results with results from simulation experiments.) Figure 3.7 depicts our experimental setup, i.e., BTnodes and sensor boards that are the foundation for the experiments described in this chapter.

Initialization

During initialization, every smart object participating in the context recognition process registers callbacks at the data space shared by cooperating objects. This is done for all the callback tuples in the SICL program the smart object is executing. In the case of dynamic groups, these callbacks must be removed from nodes leaving the shared data structure and registered at objects joining the group. Fig. 3.8 depicts the amount of time needed for registering a single callback tuple at the shared data space. As can be seen, the effort required for this operation depends on the number of objects currently cooperating with each other as well as the size of the tuple template for which a callback is registered. Because of our choice of the BTnodes as the device platform for our experiments, the

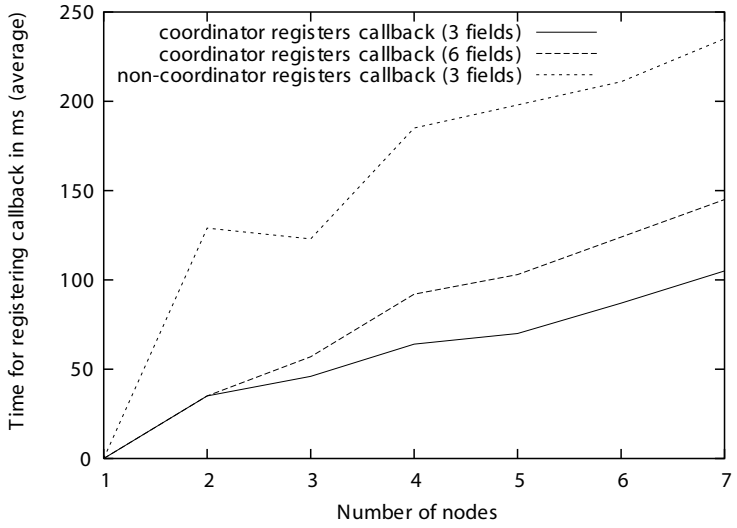

Figure 3.8: Average time for registering a callback tuple versus the number of nodes participating in the shared data structure.

amount of time needed for registering callbacks also depends on the type of node issuing an operation. This is because Bluetooth is a communication technology that distinguishes between two types of nodes: masters and slaves. The master node is the device in the center of a piconet topology; the other nodes are slaves. As the master can reach all of its slaves in a single hop but slaves need two hops to reach all nodes in a piconet, registering callbacks is more efficient for masters. The time difference in registering callback tuples from master and slave nodes is shown in Fig. 3.8. It can also be seen that local operations account for only a small fraction of the overall time required by the initialization procedure: if there is only one device present, registering a callback tuple is almost instantaneous. This is because operations that are executed on the local data structure are very efficient as a result of the severe memory constraints of smart objects. As only a relatively small number of tuples can be stored in the local tuplespace, search operations for tuples and the removal of tuples of a specified type can be realized very effectively.

In the above experiment, a single smart object issued the operation for registering a callback template and the other nodes served its requests. However, in practice all nodes access the underlying distributed data structure simultaneously in order to evaluate the context recognition rules specified in their own SICL programs. In order to take account of these practical constraints, we conducted an experiment where each of the cooperating nodes continuously registers and deletes callback tuples. Fig. 3.9 depicts the number of such operations that can be carried out on the underlying data structure in one minute.

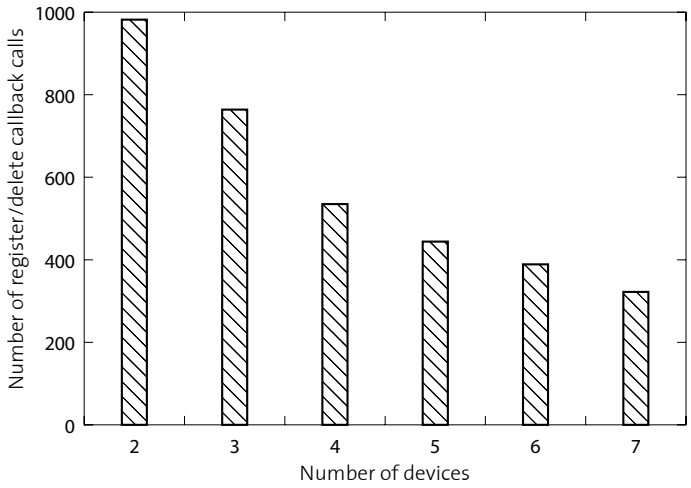

Figure 3.9: Number of register and delete callback calls carried out by all cooperating nodes in t = 1 min versus the number of nodes cooperating with each other.

With respect to cooperation, the master node coordinates the requests of other nodes and ensures fairness among cooperating entities. I.e., if all nodes continuously access the shared data space, every node can carry out an equal number of operations. In our prototypical implementation, the master executes the requests from all its slaves in a round-robin-like fashion. As the master node carries out an SICL program itself, it ensures that slaves can interrupt its own program by means of adaptive fairness slots. Fairness slots determine how many slots a master node waits for requests from remote nodes before starting a new operation itself. In Bluetooth, the master node is predestined as coordinator because all operations in a piconet topology involve the master. This is

because a piconet is a star topology with the master in the center, and slave nodes must therefore route packets via the master in order to reach other slaves.

The Cost of Rule Evaluations

In the following, we try to estimate the effort for evaluating tuple transformation rules of the form $t_1 + \ldots + t_q \rightarrow t_{q+1}$.

Private tuples. When estimating the cost of tuple transformations, the effort required for carrying out operations locally can be neglected compared to operations that involve communication. This is because the access to local tuples is relatively fast (usually in the range of a few milliseconds) as a result of the severe memory restrictions of smart objects. Hence, operations that involve private tuples contribute only a very small fraction to the overall costs of tuple transformations.

In our implementation, the time needed to search for matching private tuples depends on the amount of tuples stored in the local tuplespace and their similarity to the tuple template they are matched against (if tuples resemble each other but do not completely match, in general more tuple fields have to be compared with each other). In an experiment, a local data space with a capacity of 100 tuples and a maximum of 600 fields – amounting to a size for the local space of approximately 8.5 kB – was used to collect data about the performance of operations with private tuples. In order to estimate the worst-case scenario for searching the data space, the time needed for a *scan*(*templ*) operation was measured using the local tuplespace filled with 90 matching tuples. The template *templ* had 6 fields, the *scan*(*templ*) operation therefore returned an array with 90 tuples with 6 fields per tuple. Under these – worst-case – circumstances the average time for a *scan* operation is about 14 ms. A more realistic scenario with 10 matching tuples led to time consumption of 1.5 ms per *scan* operation. In our discussion about how to improve rule evaluations (cf. Sect. 3.4.5), we mentioned that other operations than *scan* can be used to check for tuples in the shared data space. Considering again the previous experimental setup (10 matching tuples), *read* and *take* operations for private tuples take only 0.1 ms on average. These figures confirm our assumption that local operations are insignificant compared to operations involving communication.

Callback tuples. When a smart object in a group of cooperating smart artifacts generates context information or sensory data matching a callback tuple, this data is transmitted to the object that registered the callback. In other words, if any of the tuples $t_1, \ldots, t_q$ is a callback tuple, corresponding context or sensory data are transferred in a push-based fashion. The time needed for transmitting callback tuples in such a way depends on the number of nodes cooperating with each other as well as on the number of hops over which a matching tuple has to be transmitted. As a piconet is a two hop topology, data is sent over at most two hops. In Fig. 3.10 it is shown how the number of collaborating nodes and the number of hops affect the time for delivering callback tuples from the smart object generating context data to the node that registered a corresponding callback. In our implementation, we have realized a protocol where the reception of callback tuples is acknowledged. While this implementation is useful to evaluate the effectiveness of callback tuples, the protocol could be streamlined for better performance. For instance, as the Bluetooth modules themselves should provide a reliable data delivery service, it should not be necessary to acknowledge the reception of

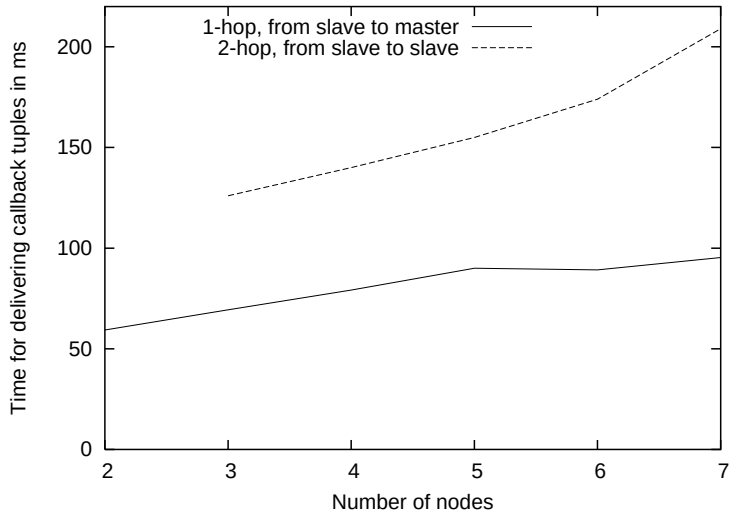

Figure 3.10: Average time for delivering callback tuples over one and two hops vs. the number of cooperating nodes.

application data packets. Such improvements could be realized in a system beyond the scope of our research prototype.

Non-private and non-callback tuples. If one of the templates $t_1, \ldots, t_q$ is neither a callback nor a private tuple, the context layer must search for matching data on the entire shared data structure. The evaluation of transformation rules containing such tuples therefore requires communication among cooperating nodes.

Fig. 3.11 depicts the amount of time required to search for non-private and non-callback tuples on the shared data space. In the underlying experiment, we embedded context data into a tuple with 3 fields and distributed tuples on remote nodes in such a way that there was always exactly one matching tuple on exactly one of the remote nodes. Furthermore, the operations were issued directly by the coordinator node. It can be seen

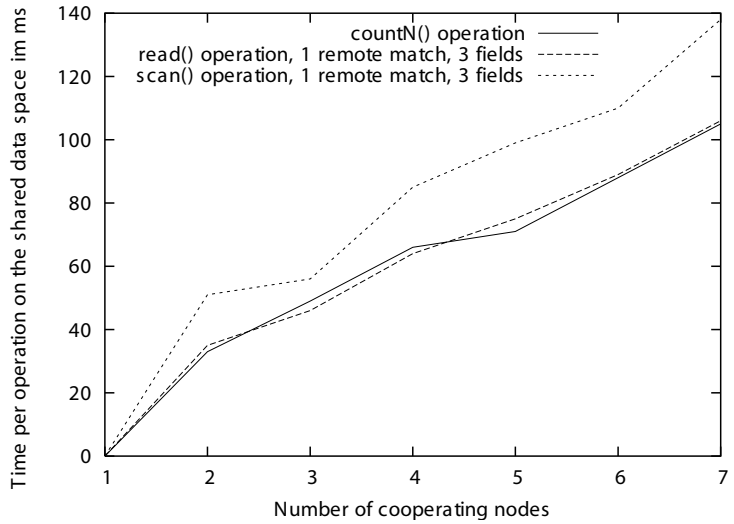

Figure 3.11: Cost of searching for matching tuples on the shared data space when the search operation is issued by the coordinator.

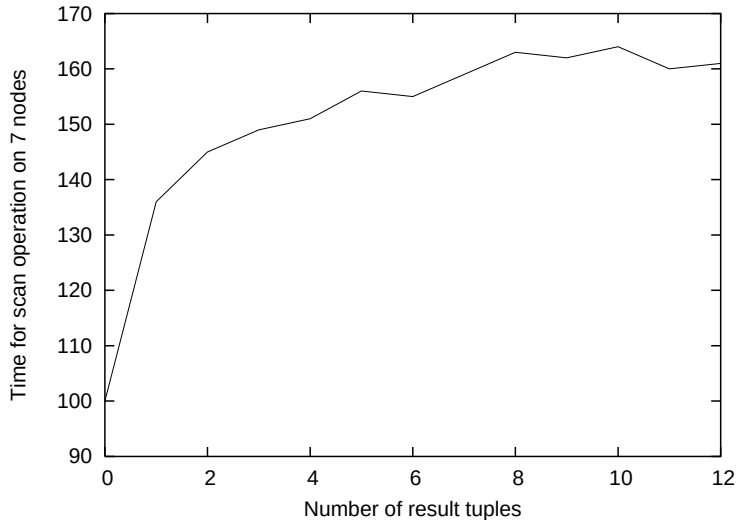

Figure 3.12: Time for a *scan* operation on 7 nodes vs. the number of result tuples.

that the *read* and *countN* functions outperform the *scan* operation under the described circumstances (i.e., even if there is only one matching tuple). Sect. 3.5 contains a thorough description end evaluation of the underlying communication protocols. However, with respect to our evaluation of tuple transformations, it is only important that the *scan* operation consumes more time than calls to *read* and *countN*. This performance difference also increases with an increasing number of matching tuples found on remote nodes because a *scan* operation always retrieves all matching tuples. In order to illustrate this effect, we have more closely investigated the performance of a *scan* operation by distributing up to 2 matching tuples on every one of 7 remote nodes. Fig. 3.12 shows how the time required by a single *scan* operation depends on the number of matching remote tuples. The effect of an increased effort with an increased number of matching tuples is considerably aggravated if matching tuples are not equally distributed on nodes (cf. Sect. 3.5).

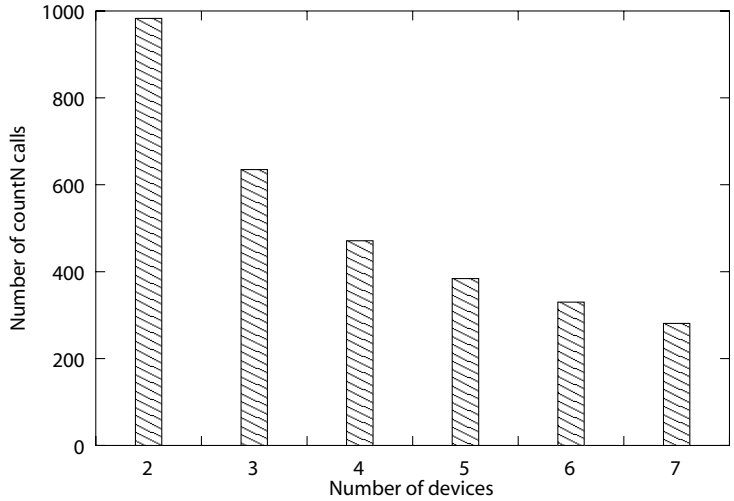

Figure 3.13: Number of *countN* operations executed on a set of cooperating objects in t = 1 min vs. the number of cooperating nodes.

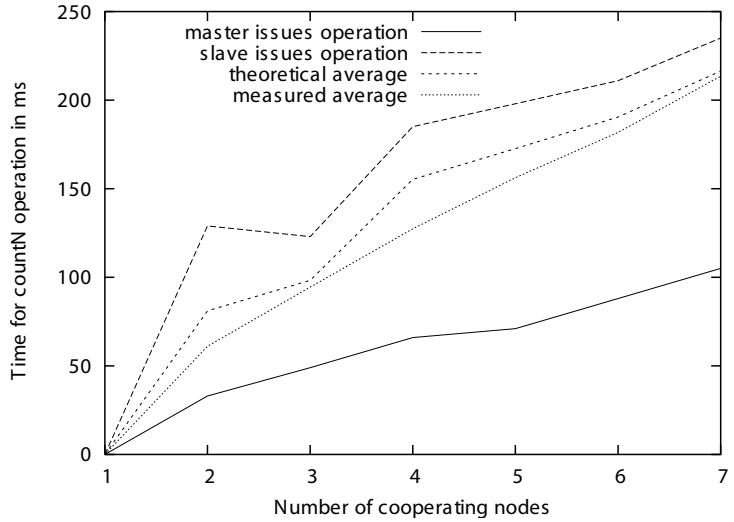

Figure 3.14: Difference between the weighted average time needed for searching tuples and the time needed for such an operation if all cooperating nodes simultaneously access the shared data space.

In consequence, context information or sensory data matching non-private and non-callback tuples should only be retrieved if absolutely necessary. This is the case after the context layer has made sure that a transformation rule can actually be executed – i.e., that there are matches for all left-hand side tuples $t_1, \ldots, t_q$ in the shared data space. It is therefore sensible to check for matching tuples using the *read* and *countN* operations because their performance does not depend on the number of matching remote tuples. Tuples are then only retrieved after it is clear that there are matches for all tuples needed to execute a transformation rule. However, in the case of tuple transformations with restricting functions, all matching remote tuples must eventually be retrieved in order to check which combination of tuples satisfies the restricting functions. Here, previous searches for matching tuples using an additional operation can introduce an overhead if transformation rules are executed frequently.

A question central to our evaluation is how cooperation and simultaneous access to the shared data space by many different nodes affects the efficiency of the context recognition process. Regarding the search for tuples in rule evaluations, the surprising answer to that question is that if multiple objects search for tuples simultaneously this does not increase the average time needed for such an operation (see Fig. 3.14). Given the results from the measurements in Fig. 3.13, it is possible to calculate the average execution time of a single *countN* operation. For example, when seven objects cooperate with each other, we measured an average execution time for a *countN* operation of 105 ms if the operation is issued by the coordinator, and of 235 ms if it is issued by slaves. Hence, the theoretical average time for this operation is $\frac{6*235ms+105ms}{7} \approx 216ms$. Considering the results from the experiment where all cooperating nodes search for tuples (see Fig. 3.13), however, 281 operations are carried out in 1 minute, which results in an average time of 214 ms per operation. Although at first this seems like a surprising result, this behavior is caused by the mechanism by which Bluetooth schedules master/slave transmissions. Furthermore, the master of a piconet can ensure fairness between nodes without requiring additional commands to be sent over the air. As already pointed out, we ensure fairness between nodes by means of adaptive fairness slots. Fairness slots determine how long a master

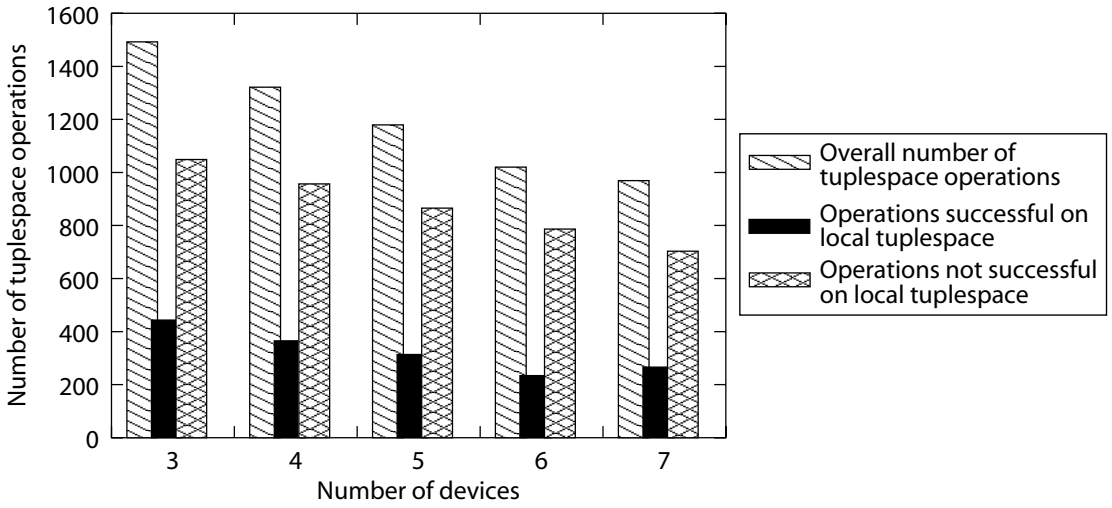

Figure 3.15: Number of operations executed on the shared data structure during the context-recognition process under heavy load in t = 3 min.

waits for remote requests before carrying out an operation on its own.

As the last major aspect in our evaluation of the context-recognition layer, we have tested its performance based on concrete SICL programs. Here, we have created SICL descriptions and uploaded the code generated from these descriptions to smart objects in such a way that every node continually tries to access the entire shared data structure. This is a worst-case scenario because the appropriate use of private and callback tuples usually absolves nodes of the need to constantly query other smart objects. The results of our experiments are depicted in Fig. 3.15; In the experiment every node executes its own SICL program and evaluates its own context rules. When comparing Fig. 3.15 with the results in Fig. 3.13, it can be seen that fewer operations on the entire shared data structure are carried out in the same amount of time. However, the performance of tuple transformations in our last experiment scales better with respect to an increasing number of nodes. This is because we have realized and experimented with different communication protocols for implementing inter-object cooperation, and used a broadcast-based version of our protocols in the most recent experiments. The next section deals with these low-level communication protocols for realizing cooperation between collections of smart objects.

3.5 Tuplespace-Based Inter-Object Collaboration

In order to realize collaborative context-aware services, smart objects need a platform for disseminating information to other nodes, for accessing sensors at various augmented artifacts, and for processing retrieved sensory data. This section describes the infrastructure component that facilitates this kind of cooperation among smart everyday objects: a *distributed tuplespace on context-based broadcast groups*. In our approach, smart objects are grouped according to their real-world context and collaborate on distinct broadcast channels. We argue, that this is an efficient means to support inter-object cooperation in smart environments for several reasons: First, many RF-based active tagging systems – e.g., the Berkeley motes as well as all Bluetooth-enabled tags – support multiple channels. By organizing cooperating objects on distinct channels for cooperation, the

interference between nodes can be reduced and communication protocols can become more effective (cf. Sect. 3.5.4). Second, in order to derive context information about a user, sensory data from smart objects that are in the vicinity of this user are usually most suited to characterize her/his real-world situation. Hence, in order to derive context information, nodes usually need to cooperate that are in close proximity to each other. Third, because of their energy constraints, smart objects generally support only short-range communication technologies, which usually have the characteristic feature that receiving data is approximately as costly as sending data. As a result, augmented artifacts do not save energy by sending data to another smart object over multiple hops if they can reach this object directly. Consequently, we can group nodes in range of each other according to their current real-world context without considering their distance. Finally, cooperation by means of a distributed tuplespace is only effective if there is an efficient way to multicast data to nodes participating in the distributed data structure. Because of the above-mentioned aspects, we can multicast data between cooperating objects by simply broadcasting data on a distinct channel. As wireless RF communications are broadcast-based by themselves – i.e., every data packet is automatically sent to all nodes in transmission range – such broadcasts are an extremely efficient way to multicast data. In the following, we describe our concepts for enabling cooperation among smart everyday objects in greater detail.

3.5.1 Tuplespace-Based Collaboration

Basic Approach

Cooperation requires smart objects to communicate with each other in order to access each other's data and resources. In our approach, a distributed tuplespace handles all basic communication-related issues required for inter-object collaboration. It hides low-level communication issues from higher layers (i.e., context and application layers) and serves as a shared data structure for cooperating smart objects. In order to realize a shared data space, every object provides a certain amount of memory for storing tuples, and the tuplespace implementation deals with the distribution of data among nodes. The context layer (cf. Sect. 3.4) accesses local sensors and embeds the corresponding data into tuples, which are written into the space and are therefore accessible from all smart objects participating in the distributed data structure. Thus, the actual location at which sensory data has been generated – i.e., the smart object from which the sensory data actually stems – becomes transparent for higher layers as long as this information is not explicitly embedded into a tuple. Hence, higher layers can operate on those sensor values as if they were derived from local sources. The tuplespace also serves as a grouping concept in that a set of sensor nodes participating in a distributed tuplespace appears to higher layers as a single node accumulating the resources of cooperating objects. From the perspective of an application programmer, this can simplify the implementation of context-aware services.

We extend the basic concept of distributed tuplespaces by grouping nodes into a space according to their current real-world context, and by assigning such groups to a dedicated broadcast channel for communication (cf. Fig. 3.16). For example, nodes that are located in the same office can be automatically grouped into one distributed tuplespace. We argue that such networking structures, which take the real-world context of smart objects into account, are better suited for the demands of context-aware applications. Sect. 3.6

addresses this claim in greater detail and presents a concrete example of how nodes can be grouped according to their current situation in the real-world. The grouping of nodes can also take application specific information or history information into account. Depending on the specific application setting, it might become necessary for nodes to communicate with objects that are not on the same broadcast channel – for example, if nodes are too far away from each other, or if smart objects belong to different distributed tuplespaces. In this case, multi-hop communication is required for nodes to interact with each other. Sect. 3.5.4 illustrates the effect of multihop communications on the cooperation between computer-augmented everyday artifacts.

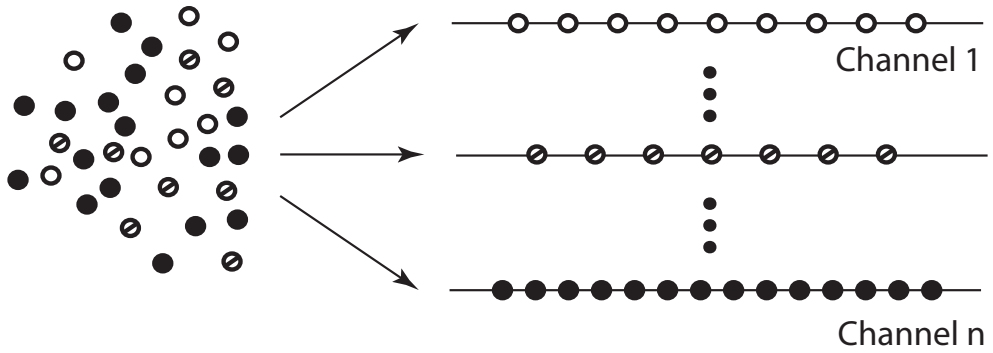

Figure 3.16: Grouping of collaborating smart objects into distributed tuplespaces that operate on different communication channels.

However, we argue that in typical Pervasive Computing environments nodes in the same distributed tuplespace can communicate on a dedicated wireless channel most of the time. This is because networks of omnipresent smart objects are likely to be dense, and cooperating nodes are likely to be close to each other in the case of context-aware applications. Cooperation on a dedicated wireless channel reduces the amount of collisions with other nodes in the same area and considerably improves the effectiveness of tuplespace operations (cf. Sect. 3.5.4). In the case of frequency-hopping spread spectrum technologies, such as Bluetooth, a channel is defined by a unique frequency hopping pattern. For multi-frequency communication modules, a channel is usually defined by a single frequency. The communication modules used on the Berkeley Motes [Mot04], for example, support multiple frequencies but do not themselves implement frequency hopping patterns.

Using the Tuplespace Concept

A distributed tuplespace serves as a shared data structure for cooperating smart objects and hides low-level communication issues from an application. In the following, we describe how such a distributed data structure can facilitate close cooperation in smart environments. We also give an overview of a prototypical implementation that is used to evaluate the infrastructure layer for inter-object cooperation presented in this section.

A tuplespace [CG89] is an associative, content-addressable memory where data is stored in form of tuples. Because the underlying concept focusses on content, tuples are accessed by their form or type rather than by memory addresses. As a content-addressable memory, a tuplespace is suited to supporting cooperation in an ad hoc environment since access to shared data is possible without knowledge of specific memory locations or

service interfaces of particular nodes. A tuplespace decouples the processes producing and consuming data. Furthermore, the physical storage locations of tuples are transparent, which facilitates anonymous cooperation. These are useful properties for dynamic settings such as smart environments of cooperating augmented artifacts. A distributed tuplespace stores tuples not at a single, predefined location, but distributes tuples between different storage locations. Thus, the storage locations of tuples can be adapted dynamically according to the current computing environment. With respect to smart environments, the distribution of tuples among different computing devices is often necessary because of the severe memory restrictions of smart objects; the distribution of tuples in a shared data space also allows smart objects to easily access data at multiple nodes with a single tuplespace operation. A distributed tuplespace also hides low-level communication issues because the corresponding tuplespace API encapsulates the distribution of tuples among smart objects and the underlying memory management for tuples.

In our tuplespace implementation for the BTnode embedded device platform, each node has its own local tuplespace, i.e., a subset of its local memory where local processes can store and retrieve tuples. If a set of nodes want to cooperate with each other, they establish a distributed tuplespace. Such a shared data medium then consists of the interconnected local spaces of all participating entities, and provides operations that consider tuples on all nodes. For example, a distributed *read*(*templ*) operation not only searches for the specified tuple template *templ* on a single node but on all entities cooperating with each other.

The distributed tuplespace for the BTnodes does not only support the standard tuplespace operations *read*, *write* and *take*, but also provides more sophisticated functions that operate on sets of tuples. *countN*(*templ*) returns the number of tuples that match the given template *templ*; *scan*(*templ*) returns all matching tuples; *consumingScan*(*templ*) behaves like *scan* but also removes all tuples found from the shared data structure. In our implementation, there are always three versions of these functions: one that operates on the local tuplespace of a smart object, one that operates on the tuplespace of a distinct remote node, and one that operates on a whole set of cooperating objects. In addition to the basic tuplespace operations, *callbacks* are supported as an event mechanism. Here, so-called callback templates together with associated callback functions can be registered at the distributed data structure. If a new tuple is afterwards written into the tuplespace that matches the registered template, the corresponding callback function is executed at the node that registered the callback.

3.5.2 Efficiency Discussion

Energy-Efficient Clustering

The question arises whether our approach of clustering cooperating nodes into broadcast groups according to their current real-world situation can be effective with respect to the energy consumption of smart objects. In fact, this is not the case for general wireless networks because energy-efficient clustering must consider the distance between devices [RM98]. Hence, the position of wireless nodes is usually the determining factor when creating energy-efficient network topologies.

This is primarily because of an inherent property of RF communication technologies which states that the received signal power decreases exponentially with increasing distance: $P \sim 1/d^n$, $n \geq 2$, where d denotes the distance from a transmitting unit and n

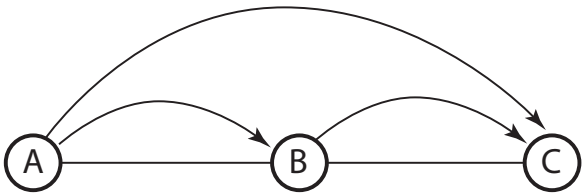

Figure 3.17: Energy-efficient communication over multiple hops.

is typically between 2 and 4. (The constant amount of energy required to receive data and to operate the transmitter modules also affects the energy consumption of nodes, especially in case of short-range communication technologies; this is discussed in greater detail in the next subsection.) Let us consider an example where three similar nodes A, B, and C are positioned on a straight line (cf. Fig. 3.17). The distance between nodes A and C shall be d meters; node B lies in the middle of A and C such that its distance from both nodes is $d/2$ meters. Furthermore, nodes A, B, and C successfully receive a packet only if its corresponding signal power at their receivers is above a threshold t. Consequently, if A wants to send data to node C, the signal power at the transmitter A must be at least $P_{AC} = td^n$. What happens, if A does not send the packet directly to C but over the intermediate node B? In this case, A must send the data with signal power $P_{AB} = t(\frac{d}{2})^n$ to B, which relays it with the same signal power $P_{BC} = t(\frac{d}{2})^n$ to the destination node C. It is now important to understand that the power invested in the actual radio signal P_{AC} for sending a packet directly from A to C is greater than P_{AB} accumulated with P_{BC}. I.e., $P_{AC} > P_{AB} + P_{BC} \Leftrightarrow d^n > \frac{d^n}{2^{n-1}} \Leftrightarrow 1 > \frac{1}{2^{n-1}}$. As $n \geq 2$ it is easy to see that the last inequality holds. As a result, sending data from A to C via the intermediate node B is more efficient than sending it directly from A to C. As a consequence of this result, the distance of nodes is the determining factor for energy-efficient clustering.

With respect to our problem, if A, B, and C are smart objects and A and C need to cooperate with each other it would not be energy efficient to cluster nodes A and C on a single wireless channel. This is because nodes in a single channel communicate with each other by simply broadcasting data, and therefore reach each other directly. Instead, in order for A and C to cooperate, we would have to relay data over node B. This implies that multi-hop communication is necessary to exchange data between cooperating nodes and that there is no simple multicast mechanism to reach all collaborating objects. This would considerably decrease the performance of tuplespace-based operations.

Clustering in Pervasive Computing Settings

In the following, we argue that our approach to grouping nodes is efficient in typical Pervasive Computing environments. This is true for several reasons. The most prominent one is that the above considerations neglect the energy required to receive data and the energy required to operate the circuitry of the transmitter modules. This is justified if the amount of energy invested in the actual radio signal is significantly higher than the constant amount of energy consumed by the receiver and transmitter modules. This can often be observed in typical long-range communication technologies: as the power of the transmitted radio signal grows exponentially with increasing distance, the signal power is the dominating factor in the energy consumption of communication modules if the distance between sender and receiver is large. However, in short-range communication

technologies a totally different picture emerges. Here, the energy required to operate the sender and transmitter circuitry dominates energy consumption. In fact, in many short-range communication technologies, the energy required for sending is approximately as high as that for receiving data. For example, the Berkeley motes consume about $2\mu J$ during reception and $4\mu J$ during transmission – with a relatively high constant component that does not depend on the actual transmission range [HSW+00]. The BTnodes also consume about the same amount of energy during data reception and transmission [BKR+03]. It is immediately clear that under such circumstances it does not make sense to relay packets over intermediate nodes. Instead, if two nodes can reach each other directly they also should exchange data directly between each other. Because of the energy constraints of smart objects, active tagging technologies usually support only short-range communication technologies. As a result, we can cluster smart objects that are in range of each other according to other characteristics – e.g., context information and application specific information – without regarding their distance.

There is another property of cooperating smart objects that is important for justifying our approach to grouping nodes. This is the observation that smart objects often collaborate in order to implement context-aware services. Determining the real-world context of nearby people is therefore one of the core reasons for smart objects to communicate with each other. As a result, there is usually tight cooperation between adjacent nodes. This is because sensor readings originating from smart objects that are in close proximity to a user are likely to provide the most accurate information for characterizing the user's situation. Therefore, in order to realize context-aware applications, it is primarily objects that are in close proximity of each other that need to exchange sensory data and context information. Consequently, cooperating smart objects are often in direct transmission range of each other.

Multicasts between Cooperating Smart Objects

Our concept for realizing context-aware services is based on the idea that smart objects group their resources, and thereby appear to applications as a single, more powerful node. The tight cooperation that is required to achieve such behavior is only possible if there is an efficient way to multicast data between cooperating smart objects. In other words, the distributed tuplespace that is responsible for disseminating data between smart objects depends on an efficient multicast mechanism.

We argue that efficient multicasts are made possible through our approach to clustering nodes. Here, two characteristics of smart objects and context-aware applications are exploited: (1) As previously mentioned, cooperating smart objects are often in direct transmission range of each other and do not save energy by sending data via intermediate smart objects. (2) Several active tagging technologies support multiple channels.

As a result, we can organize nodes that need to cooperate with each other on a single channel. Consequently, by broadcasting data, they automatically reach all objects in their group. As radio communication is itself broadcast based, this is one of the most efficient multicast mechanisms imaginable. The nodes on a single broadcast channel can now establish a distributed tuplespace that can disseminate data between cooperating smart objects using simple broadcasts. As context information can be used to group nodes (cf. Sect. 3.6), we refer to such a tuplespace as a distributed tuplespace operating on context-based broadcast groups.

3.5.3 Implementation

In the following, we describe an implementation of such a distributed tuplespace on context-based broadcast groups for the BTnode device platform [BKM+04].

BTnodes communicate with each other using the Bluetooth communication standard. Nearby Bluetooth units can form a piconet, which is a network with a star topology of at most 8 active nodes. Furthermore, the Bluetooth standard distinguishes *master* and *slave* roles. The master of a piconet is the node in the center of the star topology; slaves cannot communicate directly with each other but need to relay data via the master. The master also determines the frequency hopping pattern of a piconet and implements a TDD (time division duplex) scheme for communicating with slaves. Consequently, a slave can only transmit data when it was previously addressed by the master, and only the master can broadcast data to all units in a piconet. The frequency hopping pattern of a piconet implements a dedicated wireless channel, and as a result, different piconets can coexist within the same area. Multiple piconets can form so called scatternets. In a scatternet, bridge nodes participate in more than one piconet and switch between piconet channels in a time division multiplex (TDM) scheme.

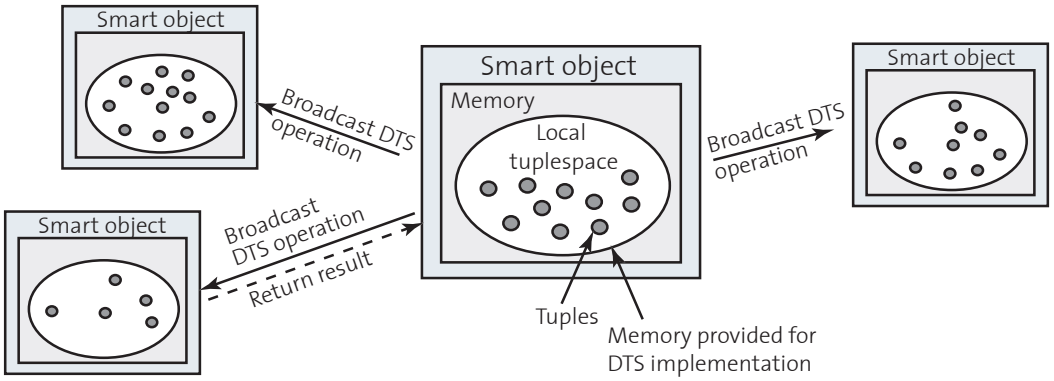

Figure 3.18: Overview of the distributed tuplespace (DTS) implementation: multiple local tuplespaces are connected to form a distributed tuplespace.

Our implementation tries to group collaborating smart objects into one piconet. This corresponds to the aim of grouping tightly cooperating objects so that they can exchange data on a dedicated broadcast channel. Because of its pseudo-random frequency hopping pattern, a piconet constitutes such a dedicated channel on which the master can broadcast data. As Bluetooth nodes in a piconet can be as much as 20m away from each other, in the envisioned settings cooperating objects can be grouped into a piconet most of the time. However, due to the low number of nodes in a piconet, it might become necessary to organize nodes into a scatternet.

Every smart object provides a certain amount of memory for the distributed tuplespace implementation (cf. Fig. 3.18). As embedded systems often do not provide a reliable dynamic memory allocation mechanism, the memory for the tuplespace is statically allocated. We have implemented our own memory management mechanism for tuples, which poses some restrictions on storing data. Because of the memory restrictions of embedded systems, for example, tuple fields always have a predefined size. That is, although a tuple may consist of an arbitrary number of fields, the type information and the actual value of a field have a restricted size. Our implementation also supports tuples with flexible length, so-called *flex tuples*. Flex tuples are a useful concept because of the

size restriction of fields: if field data becomes too large to fit into the predefined size of a field, an additional field can be created to carry the remaining data. This is sometimes necessary for strings. The memory constraints of smart objects also imply that after a time – especially when sensors are read out often – the tuplespace will be full. In this case, the oldest tuples are deleted in order to allow for new tuples to be stored. However, it is also possible to protect tuples that should not be automatically deleted. We call such tuples *owned*; they have to be manually removed from the tuplespace.

The distributed tuplespace on the BTnodes implements typical tuplespace operations such as *write*, *read*, or *take*, but also offers functions such as *scan*, *consumingScan*, and *count*, which originate from other widely-used tuplespace implementations. In our implementation, there are different versions of these functions that operate on different spaces: the tuplespace local to an object, an explicitly specified remote tuplespace, or the distributed tuplespace as a whole.

In the following, we focus on functions that operate on all nodes in a distributed tuplespace and describe the underlying protocol based on an example of a *read*(*templ*) operation.

First, we describe a *read* that is initiated by the master of a piconet. In this case, the master broadcasts a *ready?* message to all nodes on the broadcast channel and waits for their responses. After all nodes have acknowledged the request, the master broadcasts the actual operation identifier together with all necessary parameters to its slaves and awaits the results (cf. Fig. 3.19 (1)). In case of a *read*(*templ*) operation, the master transmits an identifier for *read* together with the template *templ*. The slaves respond with a tuple matching *templ* if such a tuple is in their local tuplespace.

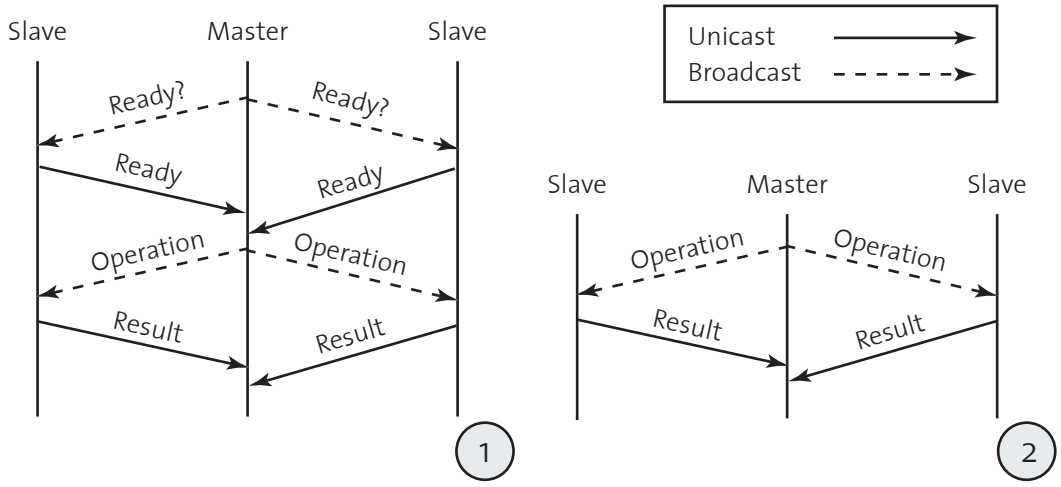

Figure 3.19: The basic protocols of distributed tuplespace operations issued by the master: (1) two-phase protocol with extra *ready?* message, and (2) the fast version of the protocol.

We have decided to implement a two phase protocol in which the master asks its slaves whether they are ready to receive operation parameters because broadcast traffic in wireless networks is generally unacknowleged; this is also true for Bluetooth. With the first *ready?* message we therefore ensure that all slaves are ready for the reception of the parameters of a tuplespace operation. The goal is to reduce the probability of lost operation parameters. Another reason for the proposed protocol design is that the protocol with an extra *ready?* message is easier to debug. However, for evaluation

purposes we have also implemented a fast version of our protocol that does not send the extra *ready?* message (cf. Fig. 3.19 (2)).

If a slave wants to execute a *read* operation on the distributed tuplespace, it must first send the request to the master of its piconet. We have again implemented a two phase protocol in this step – although the communication from slave to master is not broadcast-based and therefore acknowledged. Consequently, the slave first asks the master whether it is ready to receive a new operation by sending a short *ready?* message. Only if the master positively acknowledges this request does the slave send an operation identifier and the operation parameters to its master. After the master has then carried out the requested operation using the protocol described previously, it sends the result per unicast to the slave that issued the request (see Fig. 3.20). There are several reasons why we implemented a two-phase protocol: For our prototype we wanted a reliable protocol where higher-layer data packets are additionally acknowledged by higher layers. Furthermore, the extra *ready?* message prevents the master from being overwhelmed with data from slaves; masters at first receive only short *ready?* messages. Although we have, of course, implemented a flow control mechanism to prevent discarding of data packets, the extra phase also makes the protocol easier to debug. In a system beyond the scope of our research prototype that is based upon more sophisticated Bluetooth modules, it would be possible to get rid of the extra protocol phase.

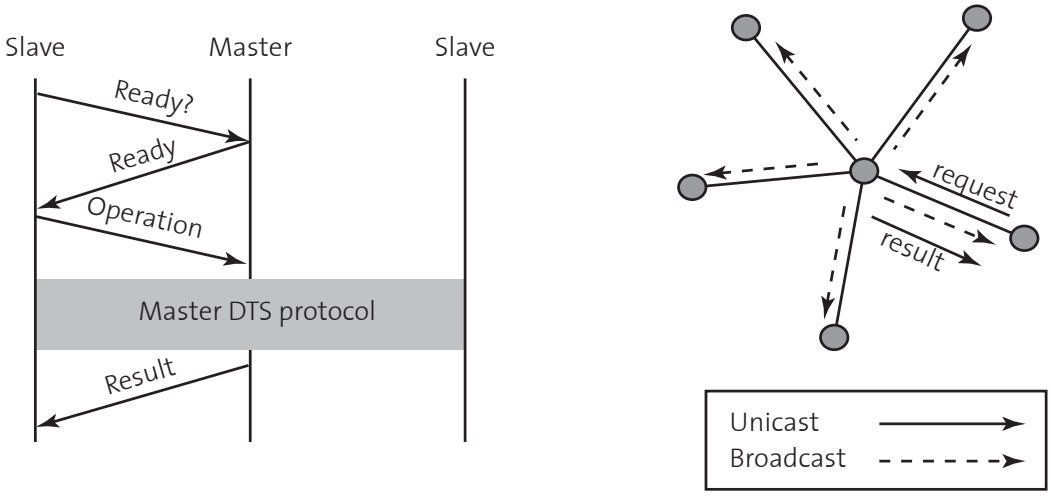

Figure 3.20: The basic protocol layout for distributed tuplespace operations when they are issued by slaves.

We would like to emphasize that the concepts of our implementation are not restricted to a specific communication standard. In fact, porting our code to other device platforms, as for example the Berkeley motes [Mot04], would be possible because they also support multiple communication channels.

3.5.4 Evaluation

By discussing the results of several performance measurements, this section evaluates the presented concepts for enabling cooperation between smart objects based on a distributed tuplespace. We justify our design decisions regarding the basic communication protocols, argue that a platform for inter-object cooperation must consider the properties of the underlying communication technology, and show by examples how the properties

of Bluetooth affect the performance of our distributed tuplespace implementation. The performance measurements are based on experiments with simple 2-hop as well as with multihop topologies. What is more, our experiments were carried out on the BTnode embedded device platform, and are not just based on simulations. However, we also simulated parts of the underlying tuplespace protocols in order to evaluate how the practical constraints of the BTnodes affect the outcome of our experiments.

Fundamental BTnode Performance Measurements

The experimental setup for obtaining the results presented in this section is similar to the one described in Sect. 3.4.6. Most notably, the BTnodes are again used as tagging technology for augmenting everyday objects, and hence Bluetooth is the communication standard upon which our experiments are based. Recently, Bluetooth has received considerable attention in connection with building resource-restricted device platforms designed for smart environments and wireless sensor networks. Among others, the Intel Motes [Int04], the BTnodes [BKM+04], smart Bluetooth badges [PLS+02], and the sensor node platform built at the University of Karlsruhe [BHHZ03] all use Bluetooth for communication. It is therefore interesting to see how the properties of Bluetooth affect the performance of our protocols. However, many conclusions that we draw from our experiments are more general, and are not limited to a specific communication technology.

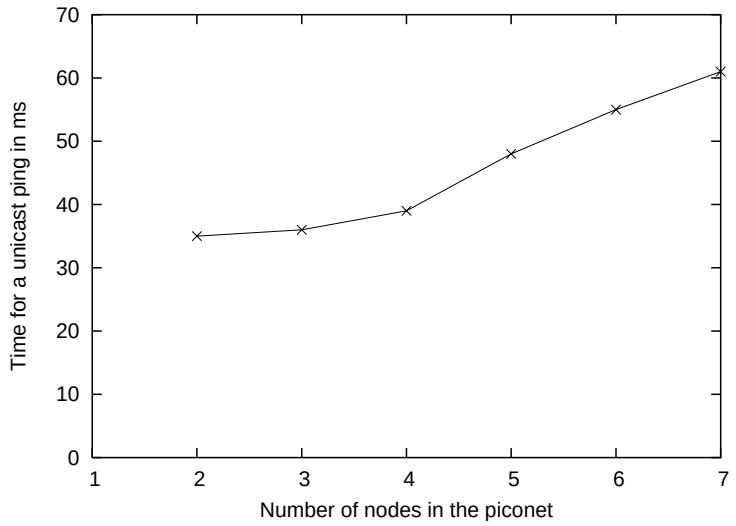

Figure 3.21: Execution time of a unicast ping operation versus the number of smart objects in a piconet.

In order to correctly assess the results of our experiments, we first measured the time needed for simple unicast- and broadcast-based ping operations (cf. Fig. 3.21, Fig. 3.22, and Fig. 3.23). The ping operation was always issued by a master node. A unicast ping consists of a simple message exchange between the master of a piconet and one of its slaves. The master sends one data packet to the slave which sends a response message back, which also fits in a single packet. In the case of a broadcast-based ping operation, the master broadcasts a data packet to all of its slaves and waits until it has received a response packet from each of them. We describe these operations because they are the fundamental building blocks of our tuplespace protocols.

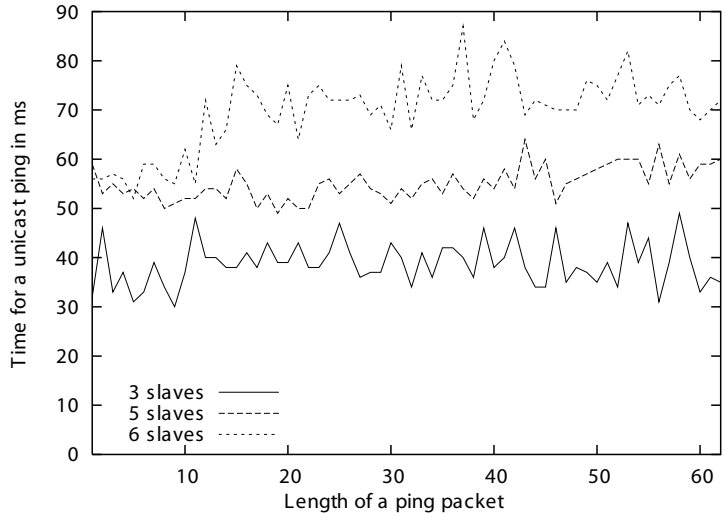

Figure 3.22: Execution time of a unicast ping operation with respect to the length of ping packets.

Unicast-Based Ping. Fig. 3.21 illustrates the dependency of the unicast-based ping operation on the number of smart objects sharing a communication channel – i.e., on the number of devices in a single piconet. The packet sent from the master to the slave and the response message consist of exactly one byte of application data. It can be seen that there is a linear dependency between the execution time of a unicast ping and the number of nodes sharing a Bluetooth channel. This is a result of Bluetooth's TDD (Time Division Duplex) scheme for scheduling transmissions. As Bluetooth has a collision-free MAC (Medium Access Control) layer, the master of a piconet polls a slave before the slave can send data back to the master. The access to slaves is slotted with a slot time of exactly 625 μs[1]. The master sends one data packet to one of its slaves, and this slave can then respond by sending a packet back to the master. Bluetooth supports different kinds of packet types, some of which can occupy multiple slots (either 1, 3, or 5 slots). In the following, we assume that the master polls its slaves in a simple round-robin like fashion, i.e., it assigns identifiers to all its slaves and polls them repeatedly one after the other.

Under these circumstances, the question arises why the time difference between a unicast-based ping in a piconet of 2 nodes and in a piconet of 7 nodes is as much as 25 ms (cf. Fig. 3.21). This has mainly two reasons: (1) If the master sends a ping message to a slave, this slave cannot send the result message back in the immediately following time slot. This is because the slave device first has to process the received ping message. In order to do so, the microcontroller of a BTnode receives the Bluetooth data via a serial interface, the packet is passed through the different layers of our Bluetooth protocol stack, and is then finally handed over to the application layer. This takes time, and the slave can therefore respond to the ping message at the earliest the next time it is polled by the master. As the master polls slaves one after the other, the time between two consecutive pollings depends on the number of slaves. (2) The BTnodes have a much lower data rate than the theoretical Bluetooth data rate of 1 Mb/s if application data is sent in

[1]In the inquiry substate, in which Bluetooth units actively search for other devices, a slot is only 312.5 μs long.

short L2CAP packets. (L2CAP, the Logical Link Control and Adaptation Protocol, is a Bluetooth protocol layer that is responsible, among other things, for data segmentation and reassembly.) This behavior of the BTnodes has also been noticed by other researchers [LDB03]. Furthermore, the Bluetooth module is connected to the microcontroller of a BTnode by means of a serial interface that has a lower data throughput than Bluetooth's maximum data rate. This is likely to be the reason for the relatively large amount of time – about 35 ms – that is required for a unicast ping in a piconet of only 2 nodes (cf. Fig. 3.21). Unfortunately, the lower Bluetooth protocol layers encapsulated by the Bluetooth modules are proprietary. We did not therefore have the opportunity to look into the corresponding program code and can only speculate about the side effects of this proprietary implementation.

In another experiment, we tested whether the performance of a unicast ping operation depends on the size of ping packets. It is intuitively clear that as long as the ping packet fits in one Bluetooth data packet, the length of the packet should not influence the performance of the unicast ping protocol. Our experiments (cf. Fig. 3.22) confirm these expectations.

Broadcast-Based Ping. In a broadcast-based ping operation, the master of a piconet broadcasts a ping packet and collects responses from all of its slaves. In contrast to a unicast ping, the performance of broadcast-based pings does depend on the length of ping packets – even if they fit into a single lower-layer Bluetooth packet (cf. Fig. 3.23).

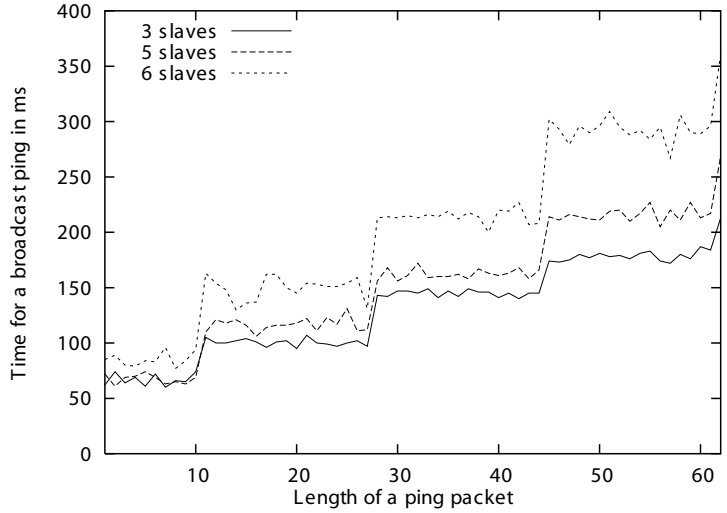

Figure 3.23: Execution time of a broadcast ping operation versus the length of ping packets.

There are two main reasons for this behavior: (1) In Bluetooth as well as in other wireless communication standards broadcast traffic is unacknowledged. This is in order to reduce the amount of traffic after broadcasts, because all nodes in range of a sender would simultaneously start to acknowledge broadcast packets. Hence, there is no guarantee for a master node that all of its slaves actually successfully received the broadcast packet. The Bluetooth standard addresses this problem by retransmitting broadcast packets several times. There is a command specified in the Bluetooth standard that allows application programmers to adapt the number of retransmissions, which is unfortunately not supported by the Bluetooth modules of the BTnodes. As a result, the BTnodes always

transmit broadcast packets five times. This is also the reason why the general performance of the broadcast-based ping operation is relatively poor compared to that of a unicast ping. (2) The Bluetooth modules on the BTnodes seem to segment broadcast data even if they would fit into a single lower-layer Bluetooth packet and transmit packet segments separately. The fact that broadcast packets are sent 5 times is the cause of the step-like curve depicted in Fig. 3.23. So, why does the Bluetooth module segment ping packets? It seems that this is because of the previously-mentioned fact that the serial connection between microcontroller and Bluetooth module on the BTnode carries data at a data rate of only 230.4 kbit/s. Hence, in a single Bluetooth slot of 625 μs only 18 bytes can be transmitted from the microcontroller to the Bluetooth transmitter. The Bluetooth module waits one slot for incoming data and then transmits them over the air. Although such behavior makes sense in the unicast case, it significantly decreases the performance of broadcasts because broadcast packets are transmitted 5 times. Unfortunately, we can only speculate about how broadcasts have been implemented by the Bluetooth modules. However, the fact that the length of the individual steps in Fig. 3.23 is 17 bytes clearly supports our theory.

We would like to conclude our evaluation of the broadcast-based ping operation with the remark that the implementation of broadcasts on the Bluetooth modules seems to be rather preliminary. Newer Bluetooth modules should provide a significantly better broadcast performance.

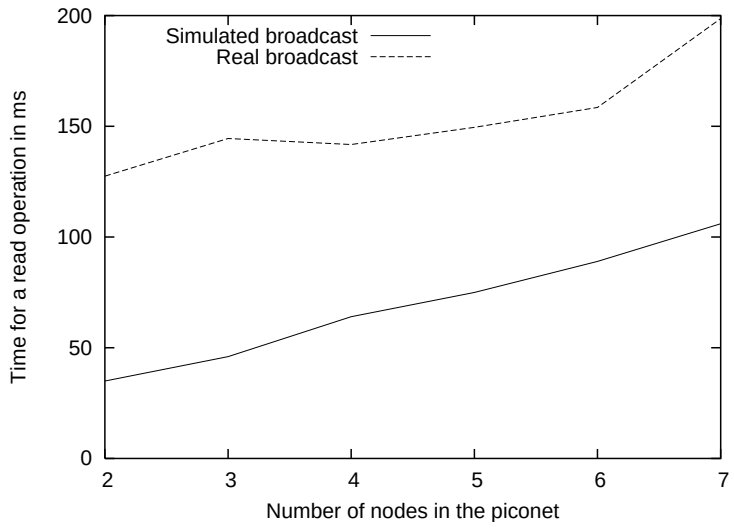

Figure 3.24: The effect of simulated and real broadcast messages on the performance of a distributed *read* operation.

Simulated Broadcast Messages. Although the performance of the broadcast-based ping operation is relatively poor on the BTnodes, broadcasts are still a major component of our protocols for enabling cooperation between smart objects. Using the BTnodes, however, it makes sense to simulate broadcasts in order to achieve a better protocol performance. In a simulated broadcast, the master of a piconet transmits the packet that needs to be broadcast to each of its slaves per unicast. It is important here that the BTnode must not wait for response messages before it has finished sending the broadcast message to all of its slaves. If the master polls slaves in a round-robin like fashion, only

one round is then needed to transmit the broadcast packet to all slaves. On the BTnode device platform, simulated broadcasts are faster than real broadcasts. This is, however, not the case for Bluetooth in general.

We have carried out an experiment to determine the effect of simulated and real broadcast messages on the performance of distributed tuplespace operations. In the experiment, we tested two different versions of a distributed *read*(*templ*) operation. The *read* implementation is based on the protocol design that does not send *ready?* messages at the beginning of new tuplespace operation. In this case, the *read* operation is similar to a broadcast-based ping in that the master broadcasts a tuple template and the slaves respond by sending a matching tuple back. In our experiment, *read* operations are only issued by the master, and there is always exactly one matching tuple at one of the slaves. As can be seen in Fig. 3.24, simulated broadcasts clearly outperform real broadcast messages in the case of the BTnodes.

Distributed Tuplespace Operations

In the following, we evaluate different strategies for realizing distributed tuplespace operations, report on the major design decisions affecting the underlying implementation, and show how the performance of tuplespace operations depends on specific properties of Bluetooth.

We distinguish between three different methods for realizing distributed tuplespace operations: (1) a parallel, (2) a sequential, and (3) a selective scheme.

- **The parallel scheme.** In the parallel scheme, a master transmits an operation identifier and operation parameters per broadcast to all of its slaves. The slaves carry out the requested operation on their local tuplespaces and send results back to the master without further requests. As slaves send result tuples simultaneously, the master must be able to receive results from different nodes in parallel. Thus, it can happen that the master receives more results than it actually needs. For example, a *read*(*templ*) operation returns only one tuple matching the given template *templ*. However, if all slaves have a matching tuple, each of them sends a tuple back to the master in the parallel strategy. The master must therefore discard results that are not needed. In case of consuming tuplespace operations – such as *take* and *consumingScan* – this problem poses another difficulty. For example, *take*(*templ*) removes only one tuple matching *templ* from the distributed tuplespace and returns this tuple as result. After receiving the parameter template of a *take* operation the slaves can therefore not decide whether to remove a matching tuple or not, and hence, can only send back result tuples without removing them from their local spaces. After the master has received the result tuples from slaves, it discards the ones not needed, and must ensure that the selected tuple – and only the selected tuple – is removed from the distributed tuplespace. This requires an additional message that is transmitted to the slave that is allowed to remove the selected tuple from its local space, and a corresponding acknowledgement.

- **The sequential scheme.** In the sequential strategy, operations are carried out successively with one slave after the other. Operation identifiers and parameters are sent per unicast from the master to the slave that is currently executing an operation. This has the advantage that no unnecessary results are transmitted from slaves that would need to be discarded by the master (as in the parallel

strategy). For a *read* operation, for example, this means that if the master receives a result from one slave it does not need to contact other nodes. The sequential strategy does therefore not necessarily require that a master contacts all of its slaves. Furthermore, in the case of consuming operations – *take* and *consumingScan* for example – no additional message is necessary that requests nodes to actually delete previously sent tuples.

- **The selective scheme.** The selective strategy is a variant of the parallel scheme. Its goal is to reduce the number of superfluous result tuples sent from slaves to a master. Upon receiving an operation identifier and operation parameters per broadcast, slaves do not respond with result tuples but merely send a packet back to the master containing the number of results in their local tuplespaces. The master then selects slaves and requests them per unicast to send their results.

In the following, we evaluate the impact of the parallel, sequential, and selective strategies on the performance of tuplespace operations. In our evaluations, we thereby use the two-phase protocol for transmitting operation parameters to slaves (cf. Sect. 3.5.3) and real broadcast messages. Although these versions of our protocols are slower, they are easier to handle for practical reasons (above all, they are easier to debug). Different protocol variants for evaluation purposes were therefore implemented based on these protocol versions.

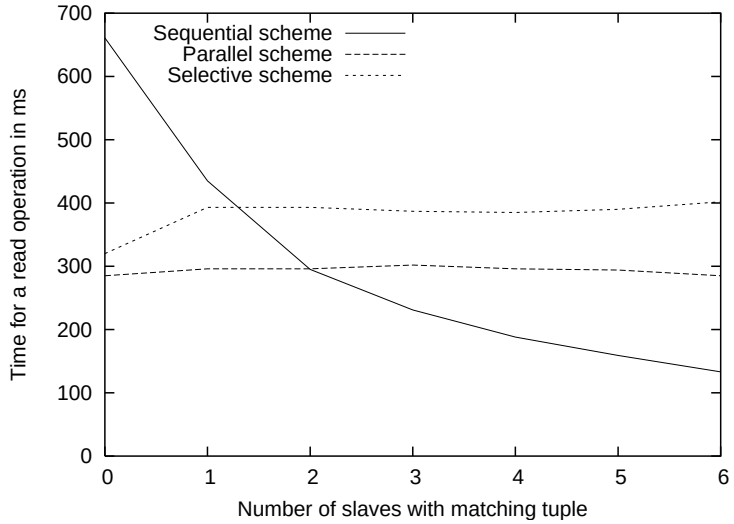

Figure 3.25: Effect of the sequential, parallel, and selective strategies on the performance of a *read* operation in a piconet of 7 smart objects.

Read. The *read*(*templ*) operation searches the distributed tuplespace for exactly one tuple matching the given template *templ*. As the distributed tuplespace that we implemented is a federation of local tuplespaces provided by individual smart objects, the *read* operation needs to query these local tuplespaces for a matching tuple. In our implementation, the distributed *read* operation first tries to find a matching tuple by querying its local tuplespace. If such a tuple is found in the local space, no communication with other smart objects is necessary because the *read* operation must return only one result tuple. Only if it does not find a match locally does it query other nodes.

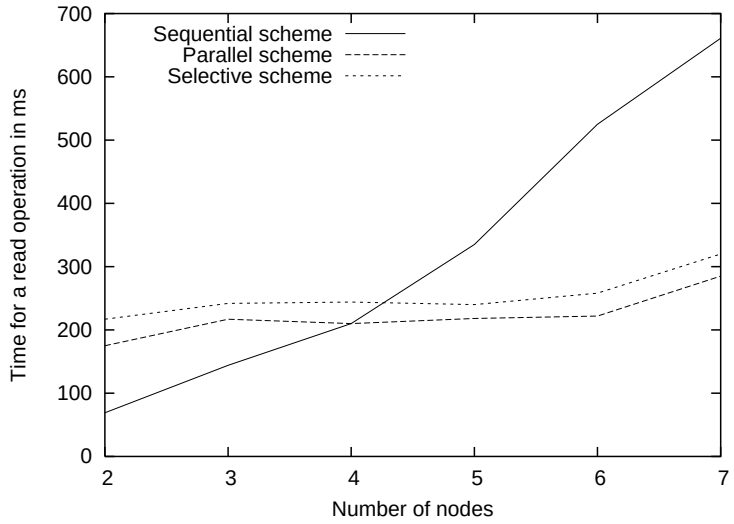

Figure 3.26: Effect of the sequential, parallel, and selective strategies on the performance of a *read* operation if there is no result on any of the nodes in the piconet.

Fig. 3.25 depicts the impact of the parallel, sequential, and selective strategy on the performance of a *read* operation with respect to the number of nodes carrying a result tuple. In the underlying experiment, operations were always issued by the master node, and the result tuples fitted in a single lower-layer Bluetooth packet. Furthermore, in order to take into account a changing number of slaves with a result tuple, we measured the time for a *read* operation for all possible distributions of result tuples and calculated the average. As can be seen in Fig. 3.25, the parallel scheme clearly outperforms the sequential strategy with a decreasing number of slaves carrying a result tuple. This is mainly because of Bluetooth's TDD (Time Devision Duplex) scheme for handling master-slave transmissions. In the sequential strategy, the master executes a remote *read* operation sequentially with one slave after the other until it finds a result tuple. The problem is that on a lower level the Bluetooth module still polls slaves in a round-

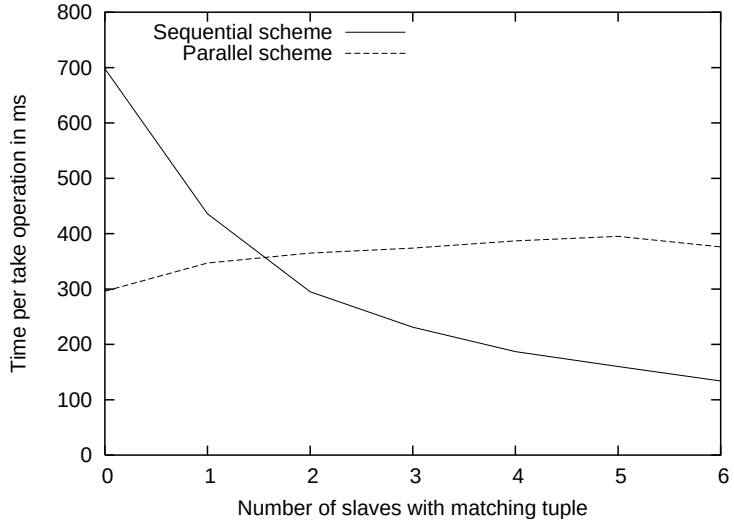

Figure 3.27: Effect of the sequential and parallel strategies on the performance of a *take* operation in a piconet of 7 devices.

robin-like fashion. Hence, when a slave is finally ready to send the result of a remote *read* operation to a master, it must wait until it is polled again by the master. As the other slaves are not involved in this particular remote *read* operation (other slaves are only queried after the current remote *read* operation is finished), this results in many unnecessary polling attempts, and therefore reduces the performance of the sequential approach.

In Fig. 3.25 and Fig. 3.26 it can also be seen that the parallel strategy beats the selective strategy in the investigated scenario. This is because of the additional unicast-based data exchange necessary in the selective strategy. Furthermore, the result tuples in our experiment fit in a single lower-level Bluetooth packet. Consequently, the transmission of a single byte indicating the number of result tuples found by a slave is approximately as fast as the transmission of an entire result tuple. Clearly, the selective strategy only outperforms the parallel strategy if result tuples are so large that they have to be split and transmitted in multiple packets.

Take. Like the *read*(*templ*) operation, the *take*(*templ*) operation searches for a tuple matching the given template *templ* and returns it as a result. Additionally, however, a *take* also removes the result tuple from the distributed tuplespace. Using the parallel strategy, superfluous tuples received from the slaves can therefore not simply be discarded by the master. Instead, the master must explicitly inform the slave whose tuple is chosen as the result of the *take* operation so that this slave can remove the tuple from its local tuplespace. This is done by means of an additional unicast-based message exchange. Fig. 3.27 shows how the parallel and sequential scheme affect the performance of a *take* operation. As the additional message exchange required in the parallel scheme is not necessary in the sequential scheme, the sequential scheme performs slightly better than in the *read* operation. However, if there is no matching tuple or if there are matching tuples at only one of the slaves, the parallel scheme still outperforms the sequential strategy.

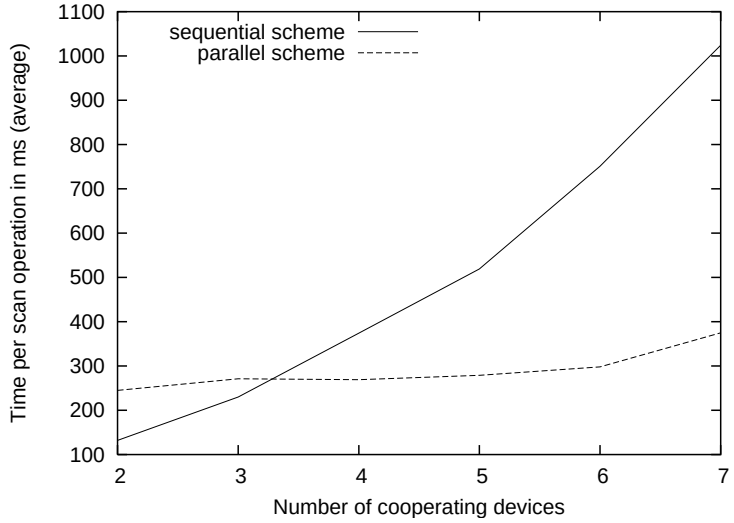

Figure 3.28: The effect of the sequential and parallel scheme on the performance of a *scan* operation that returns 10 result tuples considering all possible tuple distributions.

Scan. In contrast to a *read*, a *scan*(*templ*) operation returns all tuples in a distributed tuplespace matching the given template *templ*. Consequently, both the sequential and the parallel strategy must access all nodes in order to retrieve all matching tuples. This deprives the sequential strategy of one of its most important advantages. (Remember that a *read* or *take* operation using the sequential scheme is finished as soon as it finds a matching tuple at a single remote node.) Fig. 3.28 shows how the sequential and parallel strategies affect the performance of a *scan* operation returning 10 result tuples. In our experiment, a result tuple fitted into a single lower-layer Bluetooth packet, and the *scan* operation was issued by the master of a piconet. Furthermore, the depicted measurements consider all possible distributions of result tuples among slaves.

The performance of a *scan* operation using the parallel scheme is considerably influenced by the distribution of result tuples on remote nodes (cf. Fig. 3.29). This is because of Bluetooth's TDD scheme for scheduling transmissions. If matching tuples are equally distributed, a master almost never polls a slave without receiving a tuple as result. Consequently, in a piconet with 6 slaves it makes almost no difference whether 1-6, 7-12, or 13-18 tuples are received, provided that they are equally distributed. Because the Bluetooth modules on the BTnodes seem to poll nodes in a simple round-robin-like fashion, a master node sequentially accesses one slave after the other; it therefore needs i rounds to receive $6 * (i - 1) + 1$ to $6 * i$ results if the results are equally distributed. In contrast, when all matching tuples are on a single node, i rounds are required to retrieve i results. These considerations also explain the step-like curve in Fig. 3.29.

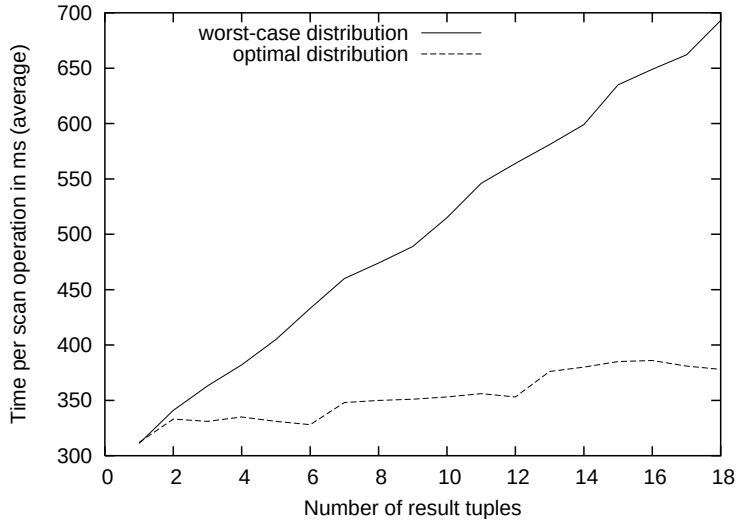

Figure 3.29: The effect of different tuple distributions on the performance of a *scan* operation using the parallel scheme.

Simulation Results

The practical constraints of the BTnode device platform influence the outcome of the previously-presented experiments. Especially the relatively slow serial connection between the Bluetooth module and the microcontroller of a BTnode, and the fact that broadcast packets are transmitted five times, have considerable side effects. In order to assess these side effects, we have simulated some of the distributed tuplespace operations presented above. Here, the network simulator ns-2 [NS04] together with Blueware

[Blu04] – an additional package that supports lower-level Bluetooth protocols in ns-2 – constitute the basis for our simulation experiments. Unfortunately, Bluetooth protocols are still not natively supported by ns-2; and even with Blueware, several changes to lower Bluetooth protocol layers were necessary before the actual tuplespace operations could be simulated.

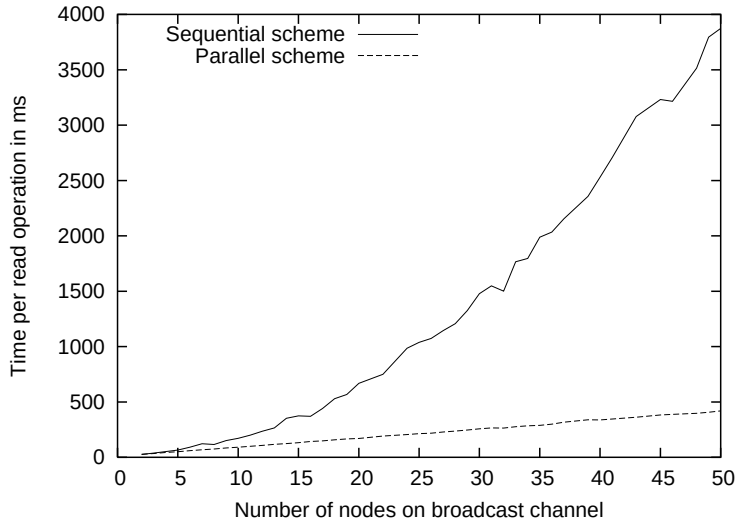

Figure 3.30: The effect of the sequential and parallel scheme on a distributed *read* operation. The results were obtained using the network simulator ns-2.

Fig. 3.30 and 3.31 show the results of our simulation experiments. They indicate that the superiority of the parallel strategy becomes even more dominant without the restrictions of the BTnodes. In the simulation experiments, broadcast packets were transmitted only once, and a result tuple fitted in a single Bluetooth packet. The simulator was also adapted in such a way that more than 8 active nodes could participate in a single piconet. In case of an increasing number of nodes sharing a single Bluetooth channel it becomes

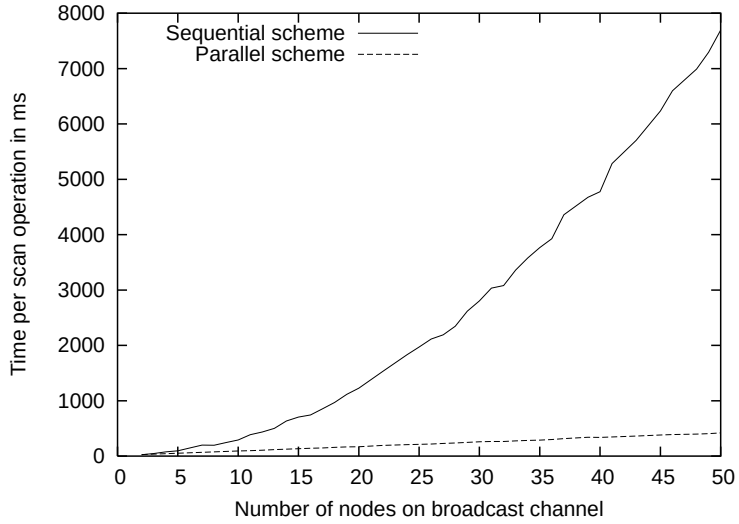

Figure 3.31: The effect of the sequential and parallel scheme on a distributed *scan* operation. The results were obtained using the network simulator ns-2.

clearer that the increase in the execution time of a tuplespace operation is only linear in the parallel scheme.

Using the sequential strategy, the increase is not linear. This is for the following reason: Given that the master uses a simple round-robin mechanism for polling nodes, the master needs more than one full round to request matching tuples from a single slave. As the master addresses its n slaves consecutively in the sequential strategy, at least n rounds are required to receive the results in the case of a *scan* operation. (For a *read* operation, a master only contacts half of its slaves on average.) Hence, at least n full rounds are necessary to complete an operation in the sequential strategy. As the master polls all of its n slaves in one round, the time complexity of a distributed *scan* operation using the sequential scheme is therefore in $O(n^2)$.

Multihop Communication

In the design of the tuplespace-based infrastructure layer for inter-object collaboration, we argued that efficient multicasts are a precondition for effectively exchanging data between cooperating smart objects. Because of the presented approach to cluster smart objects on context-based broadcast channels, it became possible to disseminate packets using simple 1-hop broadcasts, which is a very efficient way to multicast data. In our implementation of this concept on the BTnodes, this meant that the master node can reach all cooperating nodes with a single transmission. Because of the underlying Bluetooth protocols, only slaves have to communicate over multiple hops with other smart objects in the same piconet. In contention-based wireless networks, however, all nodes can usually broadcast data. Hence, multihop communication on a single channel is a special property of Bluetooth. In general, multihop communication only becomes necessary if cooperating nodes are listening at different channels or if they are too far away from each other. In Bluetooth, this corresponds to a scenario in which cooperating nodes participate in different piconets that form a so-called scatternet. In a scatternet, piconets are connected by means of bridge nodes that switch between piconets in a time division multiplex (TDM) scheme.

Based on an experiment with the BTnodes, we evaluated the impact of scatternets on distributed tuplespace operations. The core question is how the performance of the distributed tuplespace decreases if a master node cannot simply broadcast data to all cooperating objects, but must instead use a multihop multicast scheme. Regarding scatternets, the BTnodes unfortunately have the restriction that they support only simple tree topologies. This implies that a node cannot be slave in two piconets. Rather, bridge nodes – which mediate between two piconets – must be master in one piconet and slave in the other. Furthermore, the 1-hop broadcast mechanism of the BTnodes is very unreliable if nodes are organized into scatternets. The experiments on multihop topologies are therefore based on simulated broadcast messages. We described simulated broadcasts at the beginning of this section and showed that they are faster than real broadcasts.

Fig. 3.32 shows how the performance of a *read* operation is affected by multihop topologies. It can also be seen how the BTnodes were organized into scatternets; the *read* operation was always issued by the black node in Fig. 3.32. In our experiment, the issuing node first tries to find a tuple matching the *read* request in its own piconet. In this first step, the issuing master can still reach other nodes by simply broadcasting data. The lower curve in Fig. 3.32 depicts the performance of a *read* operation that is successful in the piconet of the issuing node versus the number of nodes in the entire multihop

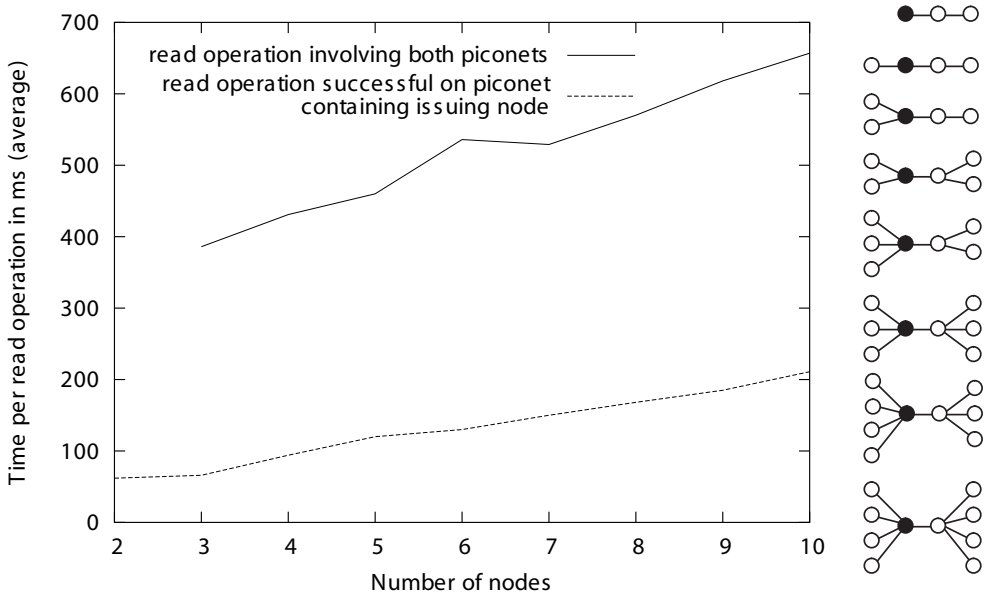

Figure 3.32: The effect of multihop multicasts on the efficiency of distributed tuplespace operations.

topology. Remember that a *read* operation is finished as soon as a single matching tuple is found. If the master finds a tuple in its own piconet, therefore, it does not need to query other nodes in the multihop topology. The upper curve in Fig. 3.32 describes the performance of a *read* operation that is not successful in the local piconet. In this case, the issuing node does not have a simple mechanism to multicast requests to cooperating smart objects but must instead communicate over multiple hops with nodes in the other piconet. Fig. 3.32 clearly shows that this considerably decreases the performance of distributed tuplespace operations. The mechanism presented for clustering cooperating nodes is therefore a major building block of efficient cooperation protocols. This is because the clustering mechanism ensures that cooperating nodes can reach each other by means of a simple multicast mechanism.

3.6 Context-Aware Communication Services

Context-aware communication services are low-level communication services that take a smart object's current situational context into account and adapt networking structures accordingly. In this section, we argue that such communication services are well suited for Pervasive Computing environments. Here, the main motivation behind our approach is as follows: As Pervasive Computing envisions the seamless integration of computation into everyday environments, there is a narrowing gap between the real world (in which people interact with physical objects) and the virtual world (in which computers communicate). In other words, the actions that affect a smart object in the real world – e.g., whether it is currently being used, how it is used, where it is, etc. – have an increasing impact on its communication in the virtual world. By considering the real world environment of smart objects in the design of networking services, we can thus construct networking structures that are more suitable for the demands of context-aware applications.

In the following, we present our approach to context-aware communication in greater detail, and describe how we linked the communication and context layers in order to

enable communication services to access and process context information. We also present two examples of context-aware communication services.

3.6.1 Basic Approach

As pointed out in the previous sections, ad hoc communication can be a critical bottleneck affecting the cooperation between smart objects. Because of the memory restrictions of augmented artifacts, local data processing is relatively fast; however, communication between smart objects – which is a fundamental requirement for cooperation – delays operations involving different artifacts and often dominates a smart object's energy consumption. Improving the effectiveness of low-level communications between smart objects is therefore of paramount importance for realizing the vision of cooperating smart artifacts.

Managing ad hoc communication between smart everyday objects is a very difficult task for several reasons: smart objects might be mobile and are often faced with distinct resource restrictions, communication is dynamic, takes place in a highly heterogeneous environment, and must not rely on a constantly available supporting background infrastructure. Furthermore, as the term Pervasive Computing suggests, there are potentially huge numbers of smart objects in communication range of each other, which further aggravates the problem of keeping communication efficient.

As a result, the question arises whether there is a specific property shared by smart objects that can be used to improve ad hoc wireless communications in Pervasive Computing settings. In this section, we will argue that context-awareness – i.e., the environmental awareness of augmented artifacts – is such a property. Our approach is therefore to consider the real-world context of smart objects in the design of communication services. The underlying motivation is that with the integration of computation into everyday items, the real-world situation of smart objects increasingly affects their communications in the virtual world. Thus, everything that happens to an object in the real world (whether and how, when and how often it is used, its current location and whether there are potential users in range) might influence its behavior in the virtual world.

Location-based routing protocols [BCSW98] already consider sensory information to improve the effectiveness of networking services. Also, Katz [Kat94] points out that mobile systems must be able to "exploit knowledge about their current situation to improve the quality of communications." Context-aware communication services, however, take such adaptiveness to a new level, because the variety of sensors carried by smart objects makes it possible to determine their current situation with unprecedented accuracy. In context-aware environments, changes in the real-world situation of smart objects also increasingly affect their behavior in the virtual world.

Beigl et. al [BKZ+03] reports on a networking protocol stack for context communication, and suggests considering situational context in order to improve its efficiency. In the work of Beigl, contexts such as the battery level, the current processor load, and the link quality serve as indicators for adapting networking parameters. We do not focus on such virtual-world parameters, but are more interested in adaptiveness based on the real-world situation of objects. Furthermore, we concentrate on providing the distributed infrastructure and high-level services that allow such adaptiveness.

In order to achieve these goals, we use the infrastructure layers (see also Fig. 3.33) presented in this chapter to design and implement context-aware communication services. The description language SICL supports programmers in realizing context-aware

communication services and in linking communication and context layers. The context layer derives information about the real-world environment of a smart object, which is the basis for adapting networking structures.

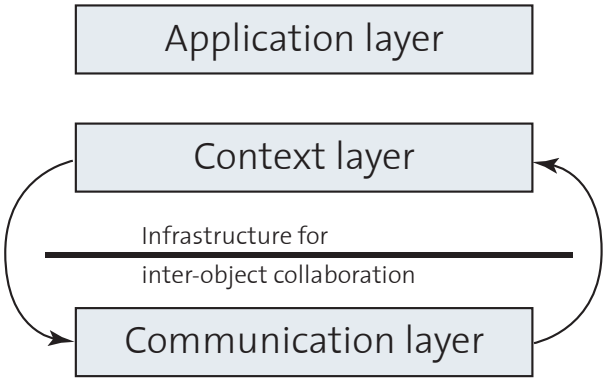

Figure 3.33: Cross-layer optimization: the communication layer accesses services provided by the context layer, which is conceptually above the communication layer.

There are three core properties that can be used to characterize context-aware communication services:

1. **Cross-layer optimization.** Context-aware communication aims at improving the effectiveness of networking services by considering the results derived in the context layer (cf. Fig. 3.33). In this respect, we speak of *cross-layer optimization* because the context layer lies conceptually above the communication layer. Accessing services from higher layers constitutes a fundamental difference to typical layered architectures, where higher layers use the services provided by lower layers, but not the other way around. For example, the application layer in our architecture accesses the context layer in order to adapt application behavior according to contextual information. Lower layers, on the other hand, do typically not depend on the services provided by higher layers. With respect to the previous example, the ability of the context layer to obtain and fuse sensory data does not depend on a concrete application. The advantage of a classical layered architecture is that lower layers are completely independent from higher layers, and can therefore be replaced by other implementations without affecting other parts of the architecture. However, as already motivated, higher layers might possess information that is valuable for lower layers. The real-world environment of a smart object, for example, can have a significant impact on how the object needs to interact with other augmented artifacts and nearby users. Cross-layer optimization is therefore an opportunity to consider the information derived by higher layers in the design of communication services.

2. **Proactiveness.** Context-aware communication services are proactive because they can react to changes in the real-world environment of a smart object before these changes affect the performance of networking services. For example, a context-aware device discovery service can start an inquiry for new devices at the moment the context layer deduces the context "object is moving." It thereby anticipates that the smart object is likely to enter a new environment in which communication and cooperation with previously-unknown objects are likely to become necessary.

This reduces the amount of time it would take to find and establish connections to other smart objects when an application finally needs to communicate with previously-unknown objects.

3. **Object-centered.** In general, context-aware applications consider the real-world situation of a nearby person in order to provide services that better correspond to this person's current needs. In contrast, context-aware communication services adapt their behavior according to the context of augmented artifacts rather than the context of nearby people. This is because at the time context-aware communication services adapt networking parameters, there are often no users in range of smart objects. Instead, context-aware communication services try to anticipate networking structures that are most suitable for context-aware applications by taking into account the current situation of smart objects. For example, the context-aware topology construction algorithm presented in Sect. 3.6.4 organizes nodes into clusters according to their current symbolic location. The underlying rationale is that nodes sharing a certain context – for example, nodes that are in the same room – are more likely to cooperate with each other to realize a context-aware application that is executed when people actually enter the smart object's environment. However, at the time smart objects create the networking topology, there do not necessarily have to be people in range of the smart objects. As a consequence, context-aware communication services usually consider the context of smart objects.

3.6.2 Linking Communication and Context Layer

Accessing the Communication Subsystem from the Context Layer. We designed the context-aware communication layer with the aim of being able to realize context-aware communication services in the same way as other context-aware applications. The first step towards this goal is to think of a communication module as a sensor. A sensor monitors its environment and provides information about its surroundings. In this respect, a communication module does not differ from acceleration sensors or microphones, because it can provide information about what other smart objects are currently in its environment. Therefore, we can export certain aspects of a communication module, for example its device discovery functionality, as a sensor statement in our description language SICL (cf. Fig. 3.34). Here, the sensor statement specifies when to read out the communication module – i.e., when to check for other objects in range – and implicitly makes this information available to all other objects by means of the infrastructure layer for inter-object collaboration. Consequently, a tuple that contains the address of a discovered module must also contain the address of the module that actually found it. This can be seen in Fig. 3.34, where the tuple generated by the "sensor" contains the lower and upper address part of the inquiring unit (lapi and uapi) as well as the corresponding information for the discovered module (lapd and uapd).

Considering communication modules as sensors enables the context layer to integrate the output of communication modules into the general context-recognition process.

Relaying Context Information to Communication Primitives. As already mentioned, one of our major goals is to implement context-aware communication services in the same way as other context-aware applications – i.e., by using the layered architecture presented in this chapter. Context-aware communication services are therefore specified

```
define sensor communication_module {
    <lapi, uapi, lapd, uapd>;
    void start_inquiry();
    random interval [40000, 60000] ms;
};
```

Figure 3.34: Linking communication and context layer by treating communication modules as sensors: sensor statement for a communication module in SICL.

in SICL. Using adaptation rules in SICL, communication services can adapt their behavior according to context changes, and parameters derived during the context-recognition process can be passed to communication primitives. To clarify this approach, Fig. 3.35 depicts a simple example of a context-aware communication service. In the example, an inquiry for new devices is started when the smart object moves. The context "moved" is determined by a local accelerometer. In Fig. 3.35 line 11 the communication primitive for starting a discovery is called when the context "moved" is derived by the context layer. Additionally, left-hand side parameters could be passed to the function on the right-hand side; for example, the parameters x and y could be used as a parameter in the call of the *start_inquiry* function. In this way, additional information derived during the context recognition process can be relayed to communication primitives.

```
    define private callback sensor acceleration {
        <accx, accy>;
        void get_acceleration();
        best effort;
    };

    moved(x, y, status)
    ===================================
    acceleration<x, y> => moved<status>;

11: moved<status>' -> start_inquiry();
```

Figure 3.35: A simple context-aware device discovery service. Context information is relayed to communication services by means of adaptation statements in SICL.

By linking communication and context layer in the above-mentioned way, context-aware communication services can take full advantage of all the services and infrastructural components presented in this chapter. As a result, they can easily access derived context information and sensor readings from collaborating objects. It is therefore possible to implement context-aware communication services in a similar way to general context-aware applications.

3.6.3 Context-Aware Device Discovery

In some communication technologies, device discovery can be extremely slow. Especially in frequency-hopping technologies – such as Bluetooth – where there is no dedicated frequency at which devices can advertise their presence, a lengthy discovery phase is a major drawback. This problem can be addressed by cooperative approaches to device

discovery that are still based on the conventional Bluetooth discovery mechanism [SR03], or by completely substituting this conventional mechanism with a context-aware communication service. In the following solution, the context-aware discovery mechanism does not use the discovery mechanism of the communication module at all.

```
smart object door;
...
define callback sensor rfid {
    /* lower address part (lap) and
       upper address part (uap) of Bluetooth
       device address */
    <lap, uap>;
    void read_device_address_from_rfid();
    best effort;
};

object_already_in_room(lap, uap, olap, ouap)
============================================
rfid<lap, uap>' => old<olap, ouap>;

object_not_in_room(lap, uap, nlap, nuap)
========================================
rfid<lap, uap>' => new<nlap, nuap>;

old<lap, uap>' -> remove(lap, uap);
new<lap, uap>' -> join(lap,uap);
```

Figure 3.36: Subset of the code for the *smart door* in SICL.

In the *smart door* scenario, a door is augmented with a BTnode and an attached RFID reader, which serves as a sensor for the BTnode. Other smart objects have an RFID tag attached that contains their actual device address – in our example the Bluetooth address of the BTnode integrated into a smart object. When smart objects enter a room through the smart door, their device address is read out of the tag by means of the RFID reader and made available to other objects in the room through the previously presented infrastructure layer for inter-object cooperation (cf. Sect. 3.5). In our case, the infrastructure layer is a distributed tuplespace shared by all objects inside the room. The smart door should also try to include the arriving object into the distributed tuplespace shared by augmented artifacts in the room. The corresponding code for the smart door is depicted in Fig. 3.36.

The technology break of using another technology for device discovery is of course an overhead because it requires that every object is additionally equipped with an RFID tag. On the other hand, the presented approach can be faster than conventional Bluetooth device discovery. It is also possible to combine our RFID-based approach with the conventional device discovery mechanism. In this case, discovery results from the ordinary discovery procedure could be enriched with context information.

3.6.4 Context-Aware Topology Construction

As already discussed in Sect. 3.5, it is a goal of the infrastructure layer for inter-object collaboration to group smart objects according to their current context. The underlying motivation of this approach is to create a network topology where nodes that are likely to cooperate with each other are grouped on a single channel (in Bluetooth, this corresponds to organizing cooperating nodes into the same piconet). As smart objects provide context-aware services to nearby users, the recognition of the user's context is one of the main reasons for communication when a user is present. Consequently, nodes that are in the same room are more likely to cooperate because their sensory data are often better suited for characterizing the situation of a nearby person. In contrast, collaboration across room borders is less likely because sensor readings originating from another room are often not appropriate to determine a user's current situation. The context-aware service for topology construction presented in this section therefore organizes nodes that share the same symbolic location into the same cluster. Our goal is to illustrate how the infrastructure layers presented in this chapter support the implementation of such a service.

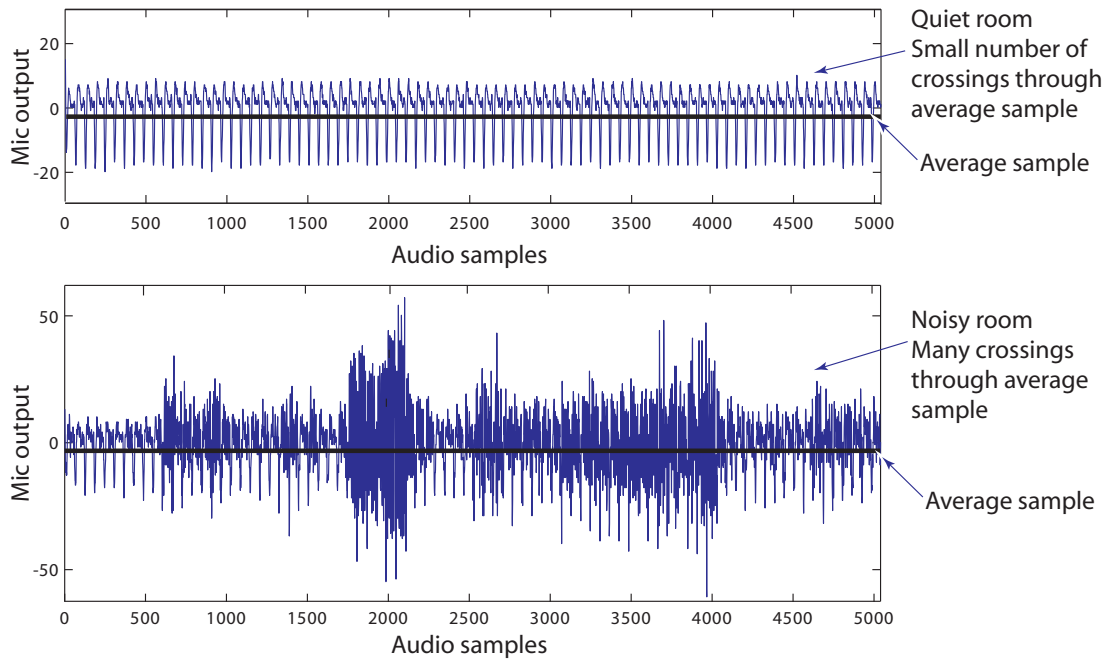

Figure 3.37: Audio output of a microphone sensor (in this example sampled with approximately 5 kHz) and the basic principle for determining the audio feature that is shared among cooperating smart objects.

At first, the context-aware communication service for topology construction assumes that nearby smart objects participate in a single distributed tuplespace. In our implementation, the location of smart objects is determined by audio input from a small microphone attached to every smart object. Every smart object in the distributed tuplespace reads out its microphone once every minute. In our experiment, a sampling rate of about 40 kHz for the duration of about one second was used for querying the microphones (cf. Fig. 3.37). As the access to the microphones has to be synchronized for all objects in the shared data space – i.e., it has to take place at the same time – a

node triggers the sampling by writing a special tuple in the distributed data structure. As a result, the microphone is then simultaneously read out at every augmented artifact.

The corresponding results are again made available to all participating nodes via the infrastructure for inter-object collaboration. Unfortunately, accessing microphones at a sampling rate of 40 kHz generates such a large amount of data that it is impractical to transmit the whole microphone data stream to other devices. Instead, a small number of features are extracted from the audio stream, which are embedded into tuples, written into the space, and thereby shared with other nodes. The extracted features consume significantly less memory than the original data stream, but still carry the information necessary for the context recognition process – i.e., the extracted audio features can still be used to find out which objects are in the same room. In our prototypical implementation, we use the number of crossings through the average audio sample as a feature (cf. Fig. 3.37). The number of crossings indicate the level of activity at a certain location and are the basis for the context-recognition process. However, because a single extracted feature provides only limited information and synchronization problems can lead to inaccurate information, one sensor feature alone is not sufficient to reliably deduce which objects are in the same room. Instead, only after several sensor accesses is it possible to draw a conclusion about the location of a node. The cluster head – in Bluetooth the master node – accesses the history of features by querying the distributed tuplespace and, if necessary, issues a command for organizing nodes into different clusters. Fig. 3.38 shows features that have been extracted by four smart objects over a time frame of about 1 hour. It can be seen that the features of nodes that are in the same room are correlated, i.e., they tend to decrease and increase simultaneously. This characteristic is used in the context-recognition process.

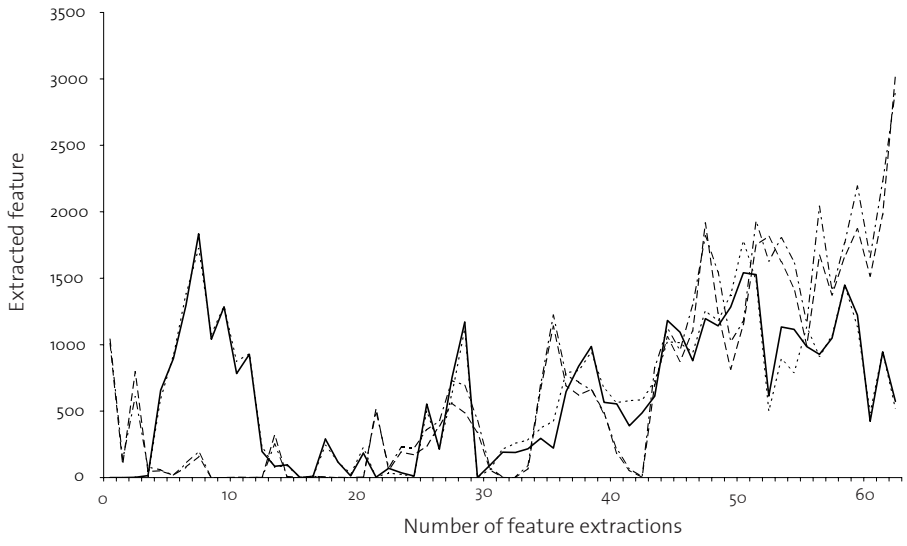

Figure 3.38: Audio features extracted by four smart objects; two of these four devices are in one room and two in another, adjacent room; it can be seen that the audio features of smart objects in the same room are correlated, i.e., they tend to decrease and increase simultaneously.

3.7 Summary

This chapter focused on the cooperation among smart everyday objects. Its major goal was to illustrate how resource-restricted smart objects can benefit from the abundance of information technology embedded into smart environments by forming groups of smart artifacts that bundle their capabilities and realize services collaboratively. In order to discuss the benefits of cooperation, we focused on everyday objects that have been augmented with active sensor-based tagging technologies, and interact with each other without the support of a background infrastructure.

The chapter introduced, described, and evaluated a software platform for smart objects that aims at facilitating the design and implementation of collaborative context-aware services in smart environments. This software platform consists of four main parts: (1) a description language for collaborative context-aware services, (2) a context layer that obtains sensory data from remote smart objects and derives context information in cooperation with other augmented artifacts, (3) a tuplespace-based infrastructure layer for inter-object collaboration, and (4) context-aware communication services. The concepts presented have been prototypically implemented on an embedded sensor node platform, the BTnodes. This implementation is also the basis for the extensive evaluations presented in this chapter.

SILC. The description language SICL aims at helping application designers to build context-aware applications in collaborative environments. The approach taken to achieve this goal involves the following steps. First, SICL provides a high-level notation for describing context recognition in terms of tuple transformations. Second, it offers a programming abstraction for dealing with collections of smart objects that allows programmers to abstract from the distribution of sensory data among cooperating nodes. The notation introduced, together with this programming abstraction, can lead to more compact code compared to programs purely written in the C programming language. When realizing collaborative context-aware services in SICL, programmers can concentrate on the fusion of sensory data instead of low-level communication issues. Third, SICL encourages programmers to write modular code and to distinguish between the different core concerns of context-aware applications. This can contribute to better structured programs.

The Context Layer. The SICL compiler transforms a description of a context-aware service in SICL into instructions that are eventually executed on smart objects. The context layer is responsible for executing the transformation rules for deriving context that were previously specified in SICL. It assembles sensory data and context information from other nodes, derives new context information from this data basis, and shares the results again with other smart objects. In short, the context layer is responsible for realizing the context recognition process in cooperation with other augmented artifacts.

Infrastructure Layer for Inter-Object Cooperation. The infrastructure layer for inter-object cooperation is a shared data space that is establish by cooperating smart objects. By means of this data space, smart objects gain access to each others' resources, and can easily exchange data and context information between each other. The shared data space is also the fundamental infrastructure layer for grouping nodes, and for hiding low-level communication issues from higher layers. It makes it possible that a collection

of smart objects appears to an application as a single node accumulating the resources of its participants. In our approach, the shared data space has been implemented as a distributed tuple space on context-based broadcast groups. We argued that efficient cooperation requires an effective mechanism for multicasting data among collaborating smart objects, and showed how such a mechanism can be realized in multi-channel short-range communication technologies.

Context-Aware Communication. Wireless ad hoc communication is a fundamental requirement for smart objects to cooperate with each other. Unfortunately, ad hoc communication between smart objects is very difficult to manage because smart objects might be mobile, because there are potentially many augmented items within communication range of each other, and because smart objects are usually faced with severe resource restrictions. The question therefore arises whether there is a property shared by smart objects that can be used to improve low-level communications. In this chapter, we argued that the environmental awareness of smart items is such a property, and showed how context-aware communication services can be realized on the basis of the presented software platform.

Chapter 4

Cooperation with Handheld Devices

The main contribution of this dissertation is to show how smart objects can dynamically exploit the capabilities of computing devices present in their local surroundings in order to provide more sophisticated services to people in smart environments. The previous chapter illustrated how smart objects can achieve this goal by cooperating with other smart objects in their vicinity. In this chapter, we focus on the cooperation between smart objects and handheld devices. Our goal is to show how augmented items can exploit the heterogeneity inherent in smart environments by accessing the computational capabilities of other kinds of computing devices.

Smart objects can benefit from interactions with handhelds because of the severe resource restrictions and limited user interfaces of computer-augmented everyday artifacts, which imply substantial problems for the design of applications in smart environments. Some of these problems can be overcome if smart objects are able to exploit the resources, I/O interfaces, and computing capabilities of nearby mobile user devices in an ad-hoc fashion. Handheld devices such as mobile phones and personal digital assistants (PDAs) are especially suited as cooperation partners for smart objects because of their complementary capabilities: while there are potentially many smart objects present in a smart environment that can collaboratively provide detailed information about their surroundings and the context of a user, handheld devices are equipped with powerful storage mediums and have elaborate input and display interfaces. In this chapter, we will argue that the heterogeneity inherent in smart environments (i.e., the presence of many different kinds of computing devices) can help smart objects to cope with their resource restrictions in that it allows them to spontaneously access the resources and different capabilities of other types of devices. We identify recurring usage patterns that describe how smart objects can make use of nearby handhelds in their own applications, show how these patterns can be realized, and present concrete applications in order to illustrate their applicability.

The chapter is organized as follows. The next section explains the reasons for cooperating with handheld devices from a smart object's perspective, and distinguishes our approach from that of other researchers. Sect. 4.3 identifies a set of characteristics shared by mobile user devices that underline why handhelds are such valuable cooperation partners for augmented items. Based on these characteristics, we identify recurring usage patterns that describe how smart objects can make use of handhelds. These patterns together with directions for their implementation are described in Sect. 4.4. Sect. 4.5 focuses on the integration of handhelds into sets of cooperating smart objects. It presents a framework for outsourcing computations to nearby handheld devices and a corresponding

prototypical implementation on an embedded sensor node platform. Sect. 4.6 presents example applications that illustrate the applicability of the described concepts. Sect. 4.7 concludes the chapter by summarizing the lessons learned from our applications.

4.1 Motivation and Background

4.1.1 Why Handhelds and Smart Objects should Cooperate with Each Other

According to the vision of Pervasive Computing, our everyday environments will be populated by vast amounts of computing devices that provide services supporting people in their daily activities. As Pervasive Computing follows a human-centered approach, the computing technology necessary to achieve such a level of user support will be as multifaceted as human activity itself. As a result, handheld devices such as PDAs or mobile phones, computer-augmented everyday artifacts, RFID-enabled consumer products, and wall-sized displays are only some of the devices that are likely to play a role in future smart environments. Consequently, heterogeneity is an inherent property of Pervasive Computing settings, and a core challenge in smart environments is therefore to exploit their heterogeneity by building applications that make use of and combine the specific capabilities provided by different types of computing devices.

Table 4.1: Characteristic properties of handhelds and smart everyday objects.

	Handheld devices	Smart everyday objects
Pervasiveness	low (none – few handhelds per person)	high (many objects embedded into environment)
Sensors	none – few	none – many
Energy resources	medium (regular recharging)	severely limited (recharging difficult)
Computing capabilities	medium	severely limited
User interfaces	sophisticated	severely limited
Mobility	medium (carried by users)	stationary – high mobility

This challenge becomes even more important in connection with smart everyday objects, which possess only severely limited amounts of resources and offer only very restricted, if any, conventional user interfaces. A smart object can therefore achieve very little on its own and must rely on remote resources to realize its services. Handheld devices are well suited as cooperation partners and resource providers for smart objects because of their complementary capabilities (cf. Tab. 4.1): In contrast to smart objects, handheld devices have relatively large amounts of resources (e.g., powerful processors, batteries with high capacity and energy density, and large storage mediums), and offer sophisticated means for interacting with people. In contrast, the computational capabilities of smart objects are very limited because their computing platforms need to be small and unobtrusive; they do not possess conventional I/O interfaces such as keyboards or

displays, which restricts the interaction with users; and finally, because of their limited energy resources, smart objects support only short-range communication technologies, which makes it difficult to access background infrastructure services.

Combined, all these limitations cause severe problems for the design of applications in environments of smart objects. In this chapter, we argue that most of these problems can be overcome if smart objects can spontaneously access the capabilities of nearby handheld devices. In smart environments, people move around who carry their personal devices with them. By exploiting the features of nearby handhelds in an ad hoc fashion, new possibilities for the design of applications on smart objects evolve. Handhelds can enrich the interactions between smart objects, users, and background infrastructure services (cf. Fig. 4.1): Mobile user devices facilitate the ad hoc interaction between smart objects and background services by serving as a mobile access point to the backend infrastructure. A handheld's input and display capabilities enable new forms of user interactions with smart objects. And finally, the cooperation among smart objects themselves can be improved by utilizing handheld devices as resource providers and mediators between smart objects.

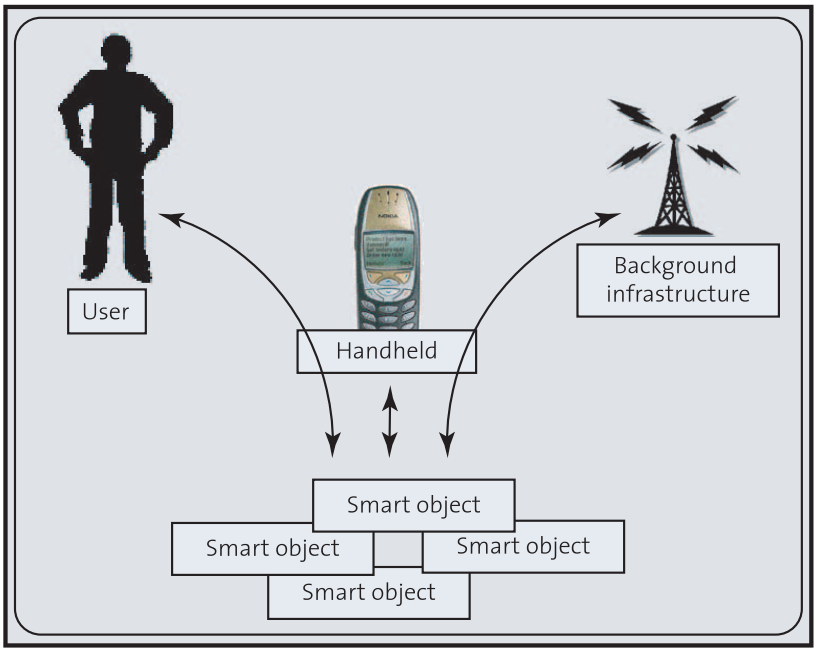

Figure 4.1: Handhelds as mediators in smart environments: handheld devices enrich the interaction between different smart objects, between smart objects and their users, and between smart objects and background infrastructure services.

4.1.2 Background

Previous work on the interaction with smart objects by means of handheld devices (e.g., [Kon02], [FFK+02], [BG99], and [BR01]) focuses on smart objects that have been augmented with passive tagging technologies or active RFID tags. The sole functionality of such smart objects consists in providing a unique identification for a tagged item. Hence, although handheld devices equipped with barcode scanners or RFID readers are used to read out the ID of a smart object, the actual application associated with an augmented item cannot be implemented by the smart object itself. Instead, the handheld device accesses the identification stored on an object's tag and relays this information

to a background infrastructure service – the virtual counterpart of the smart object. The actual application and user interaction is then handled between the handheld device and this infrastructure service – usually without further involvement of the augmented item itself. The "cooperation" between smart objects and handheld devices is therefore restricted to the simple operation of obtaining the object's identification. This is the fundamental difference to our approach: as the everyday objects that we investigate are augmented with active sensor-based computing platforms, they can implement applications on their own without the need of a supporting backend infrastructure. In our work, smart objects can provide their services more independently from background services, access resources of nearby handhelds whenever they deem it appropriate and not only when they are triggered by external reader devices, and finally, people use their handhelds to interact directly with a smart object, not a virtual representation.

Hartwig et al. [HSR02] integrates small Web servers into active Bluetooth-enabled tags, attaches them to physical objects, and controls augmented items with nearby Bluetooth-enabled phones by means of a Wireless Application Protocol (WAP) interface. Interacting with smart objects based on WAP interfaces displayed on handheld devices is one of several possibilities presented in this chapter to exploit the user interface capabilities of mobile user devices (cf. Sect. 4.4.2). As opposed to the approach proposed in Hartwig et al. [HSR02], we show how WAP interfaces can be used for local as well as remote interactions with smart objects. Besides a handheld's role as a remote user interface, this chapter also identifies several other means by which mobile user devices can help smart objects to realize applications. We particularly focus on the integration of handhelds into sets of cooperating objects, not only on the interaction with a single augmented item.

Gellersen et al. [GSB02] proposes integrating low-cost sensors into everyday objects and mobile user devices in order to facilitate the development of context-aware applications. In their MediaCup project [BGS01], active sensor tags are embedded into coffee cups to derive and provide services based on the situational context of users. For example, the context "meeting" can be inferred from the presence of many hot cups in a meeting room. However, their work assumes stationary access points within range of smart objects as well as background infrastructure services that coordinate the cooperation between active artifacts and process sensory data originating from multiple smart objects. In contrast, we explicitly focus on utilizing nearby handheld devices in an ad hoc fashion as mediators between smart objects, users, and background infrastructure services. Augmented items do not have to relay sensory data to a backend service responsible for context processing, but can derive context information in direct cooperation with other smart objects and nearby handhelds.

In the TEA project [TEA04], mobile phones are augmented with sensors in order to enrich phone calls with context information; the goal is to provide cellphones with an increased awareness of the current situation a user is in. As opposed to the Calls.calm [Ped01] and Context-Call [Sch00] applications, a mobile phone can thus automatically – i.e., without explicit user interaction – determine whether to accept or reject a call considering local sensor readings [GSB02]. The difference to our work is that we leave handheld devices unchanged, because the necessary context information will be derived by a collection of smart artifacts in the user's environment.

4.2 Solution Outline

The goal of this chapter is to show how smart objects can come to terms with their resource restrictions by cooperating with nearby handheld devices. In order to achieve this goal, we start by identifying a set of characteristics shared by mobile user devices in smart environments. Based on these characteristics, we then derive six usage patterns that describe how smart objects can make use of nearby handhelds. These usage patterns – in the following also called *roles* because every usage pattern focuses on a specific aspect of handheld devices – are: (1) mobile infrastructure access point, (2) user interface, (3) remote sensor, (4) mobile storage medium, (5) remote resource provider, and (6) weak user identifier. We then describe these roles in more detail and give directions for their implementation. In this step, we also report on our experiences with prototypical implementations of the mentioned usage patterns for Bluetooth-enabled mobile phones and PDAs, which cooperate with smart objects that are augmented with an active sensor-node platform, the BTnodes [BKM+04].

Among all the above-mentioned usage patterns for accessing the capabilities of handheld devices, the role of *remote resource provider* is of special interest in the scope of this dissertation because it describes how handhelds can be integrated into collections of cooperating smart everyday objects. In this case, mobile user devices serve as mediators between different smart objects and facilitate cooperation among these entities. In the approach we present in this chapter, smart objects are enabled to utilize a nearby handheld device as a remote resource provider by outsourcing computations to it. This is achieved by combining two concepts: distributed shared data access and mobile code. First, cooperating smart objects establish a distributed data structure for processing data collaboratively as well as for sharing information and resources (see also Sect. 3.5). As a result, it becomes irrelevant at which of the cooperating nodes program code is executed as long as the program retrieves data by calling functions operating on the shared data space and writes results back to it. Hence, nearby handheld devices can now join the distributed data space and can serve as an execution platform for code from smart objects. We illustrate our approach with a programming framework and runtime environment called *Smoblets*, which has again been prototypically implemented using the BTnode device platform.

Having described the above-mentioned patterns for making use of handheld devices in smart environments, we present several application scenarios and corresponding prototypical implementations that illustrate their applicability. The scenarios include an application supporting remote interactions with smart objects, a smart medicine cabinet, an object tracking and monitoring application, and a smart training assistant. Our goal in prototypically implementing these scenarios was to illustrate the versatility and reusability of the identified usage patterns in different settings.

4.3 Characteristics and Roles of Handhelds

The roles of handheld devices in environments of computer augmented everyday artifacts are manifold. Handhelds can serve as a primary user interface, they can be a mobile infrastructure access point, provide mobile data storage, act as a user identifier, supply energy and computational resources, or offer sensing capabilities. In this section we identify the main reasons for this versatility. We start by naming important character-

istics of handheld devices and from these derive the major roles of handhelds in smart environments.

Habitual presence

As mobile phones, PDAs, and other handheld devices are habitually carried around by their owners, they are likely to be in range of a smart object when a physical interaction with it is about to take place. This is especially important because smart objects themselves generally do not have access to resources beyond their peers, and handheld devices are often the only local devices able to provide powerful resources and sophisticated services. The habitual presence of handheld devices during physical interactions with smart objects is a fundamental characteristic that allows smart objects to rely – to a certain degree – on the presence of more powerful computational resources in their neighborhood. It entails a handheld's general function as a mediator between smart objects, users, and background infrastructure services, and is a basic precondition for the roles of handhelds presented in the remainder of this section.

Wireless network diversity

Mobile phones and PDAs usually support both short-range as well as long-range wireless communication technologies, such as Bluetooth, IrDA, WLAN, GSM, or UMTS. This enables handhelds to not only interact with smart objects directly via short-range communication standards, but also to relay data from augmented items to powerful computers in an infrastructure far away. The characteristic of wireless network diversity makes it possible for handhelds to serve as *mobile infrastructure access points*.

User interface and input capabilities

Tags attached to everyday objects have to be small, unobtrusive and are ideally invisible to human users. Consequently, they do not possess conventional buttons, keyboards, or screens. Interaction with augmented objects therefore has to take place either implicitly by considering the sensory data of smart objects, or explicitly by using the input and display capabilities of other devices [SF03a]. As people are usually familiar with the features provided by their handhelds, interactions with smart objects that are based on these well-known interfaces should imply a more comfortable and easier usage of smart objects. As a result, handhelds often serve as the primary *user interface* for smart objects.

Perception

Handheld devices can serve as *remote sensors* for a smart object, which then accesses sensory data wirelessly using a communication technology supported by all participating devices (i.e., supported by the handhelds as well as the augmented artifact). The way in which handheld devices perceive their environment strongly depends on their functionality. Cellular phones, for example, know to what cell they currently belong and can serve as remote location sensors for augmented items. Other sensors that are often integrated or attached to PDAs and mobile phones range from barcode scanners and RFID readers to digital cameras. Many of these sensors are difficult to integrate into smart objects because they are either too complex and obtrusive to be embedded into everyday things (e.g. digital cameras) or draw too much energy (such as RFID readers). Cooperation

with handheld devices is therefore often the only possibility for smart objects to obtain environmental information from such complex sensors.

Mobility

Active tags can transfer data such as how to reach a smart object from remote locations to a handheld device, where it is permanently stored and accessible for users independently of their current location. Thus, although handhelds are mobile, information stored on them from a smart object remains accessible for users most of the time. This feature can be used in services provided by augmented items, in which handhelds then serve as a *mobile storage medium* for smart objects.

Computational resources and regularly refilled mobile energy reservoirs

Although the energy consumption of a handheld device such as a cellular phone should be as small as possible, people are used to recharging its battery at regular intervals. PDAs are often shipped with a cradle that offers both host access to the device and automatic recharging. A similar procedure, however, is not feasible for smart objects because there are just too many of them. As a result, smart objects may exploit the energy resources of handheld devices in range, for example by carrying out complex and energy consuming computations on them. Because of regularly renewed energy resources, handhelds can also offer more powerful resources regarding memory and bandwidth, which allows smart objects to use them as *remote resource providers*.

Personalization

PDAs and mobile phones are most often personalized, i.e., they belong to a certain person who uses the device exclusively. Smart devices can therefore adapt their behavior according to the current handheld devices in range and thereby offer functions tailored towards certain persons. Handheld devices also offer means for user authentication and authorization such as PIN numbers or fingerprint sensors. In this context, handhelds can serve as *weak user identifiers* for services provided by smart objects.

Table 4.2: The roles of handhelds in smart environments and the underlying characteristics that entail these roles.

Handheld's role	Underlying characteristic
Mobile infrastructure access point	Wireless network diversity
User interface	Input and display capabilities
Remote sensor	Perception
Mobile storage medium	Mobility
Remote resource provider	Computational resources Regularly refilled mobile energy reservoirs
Weak user identifier	Personalization

The relationship between the described characteristics of handheld devices and the roles we derived from these characteristics are summarized in Tab. 4.2.

4.4 Role Descriptions

After having identified the major roles of handhelds in smart environments on a more conceptual level (cf. Tab. 4.2), we now describe these roles in more detail, report on the concepts underlying their realization, and present prototypical implementations. Practical issues are illustrated on the basis of a concrete implementation for the BTnode embedded device platform (cf. Fig. 4.2). BTnodes are small computing devices, consisting of a microcontroller, Bluetooth communication modules, an autonomous power supply, and externally attached sensor boards. As Bluetooth is integrated into an increasing number of consumer devices, BTnodes are suitable for illustrating the roles of handhelds in smart environments. Further advantages of Bluetooth in this context are that it is (1) an open standard and (2) does not require manual user interaction. The latter point becomes clear by comparing Bluetooth with IrDA, the quasi-standard communication technology embedded into almost every mobile phone and PDA: whereas infrared technologies require users to align devices to ensure line-of-sight for communication, Bluetooth requires much less attention from the user side. In fact, interaction between handheld devices and Bluetooth-enabled smart objects can take place without any supervision by human users.

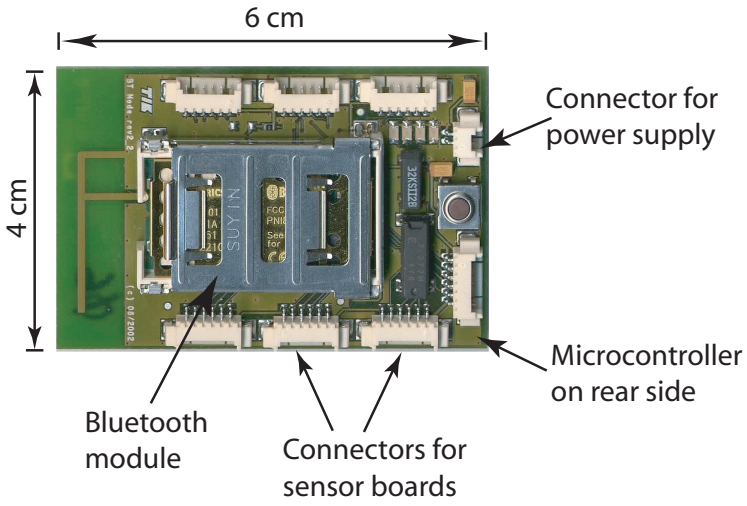

Figure 4.2: BTnodes serve as a device platform for making everyday objects "smart"; prototypical implementations for the BTnodes and Bluetooth-enabled handhelds are used to illustrate the concepts presented in this chapter.

However, besides these advantages of Bluetooth, the basic concepts and usage scenarios of handheld devices presented in this chapter neither depend on an actual device platform nor on the incorporated communication standard. Our choice of Bluetooth-enabled tags to illustrate the previously identified roles of handhelds mainly reflects the fact that Bluetooth modules are integrated into an increasing number of commodity devices and therefore facilitate easy integration of handhelds into smart environments.

4.4.1 Mobile infrastructure access point

Basic Concept

Because of their wireless network diversity – i.e., their support of short-range as well as long-range communication technologies – handheld devices can serve as mobile gateways to background infrastructure services. Technically, this is achieved by establishing a local short-range connection from a smart object to a handheld device, and a long-range communication link from the handheld to a background infrastructure server (cf. Fig. 4.3). The wireless technology used to build up the long-range connection to the backend infrastructure depends on the capabilities of the handheld device. In the case of PDAs this might be an IEEE 802.11 link to a base station, and a GSM (Global System for Mobile Communication) or GPRS (General Packet Radio Service) connection in the case of mobile phones.

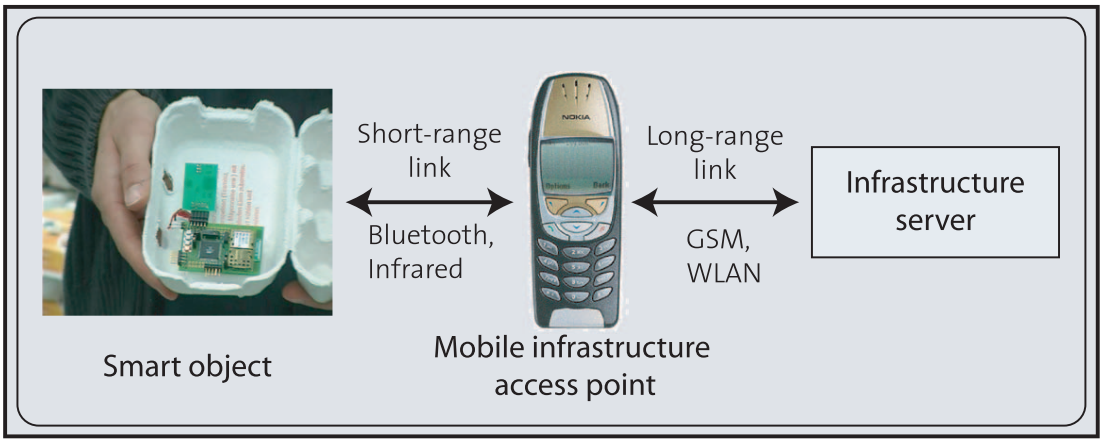

Figure 4.3: Mobile access points: smart objects use nearby handheld devices to communicate with background infrastructure services in an ad hoc fashion.

Motivation

The advantage of using handhelds as mobile infrastructure access points is that smart everyday objects can exchange data with backend services without relying on fixed, stationary installed, and often expensive access points within range. Furthermore, people usually use the services provided by smart objects when they are in close proximity of these objects, for example, during or after physically interacting with them. In this case, mobile phones and PDAs, which are usually carried around by their users, are an interesting alternative to stationary access points because they are likely to be in range of a smart object when it needs to access backend services. To illustrate this aspect, let us consider a cabinet that keeps track of its contents as an example of a smart object. Here, the state of the cabinet relevant for an application changes only when people withdraw items or put additional things inside the cabinet. Consequently, mobile phones carried by the cabinet's users are sufficient for the smart object to send application-relevant data to backend services because application-relevant data are only generated when people physically interact with the cabinet.

The alternatives to using handhelds for accessing backend infrastructure services are dense infrastructures of fixed access points that must be in direct transmission range of

smart objects, and multihop communication with remote fixed access points. Infrastructures of fixed access points are complex and costly because stationary access points need to be as close to the smart objects as possible in order to ensure communication and to save energy. This is because an inherent property of radio-based communications states that the power of a transmitted radio signal P is inversely proportional to the nth power of the distance from the transmitter – $P \sim \frac{1}{r^n}$ with $n \geq 2$. As a consequence, smart objects usually support only short-range communication technologies. Thus, many stationary access points would be necessary to ensure background infrastructure access for smart objects. The downside of the second alternative – multihop communication with remote access points – also results from the energy consumption of nodes. In particular, it has been shown that nodes in close proximity of a base station consume significantly more energy than nodes farther away, because they have to relay data from other objects. This can lead to a shorter lifetime of the smart objects that are in direct transmission range of an access point. Considering the above mentioned issues, from the perspective of smart objects, infrastructure access by means of handheld devices is advantageous because handhelds are likely to be very close. As a result, the transmit power levels of smart objects can be very low, which can considerably extend their lifetime. Because of the mentioned property of radio-based communications ($P \sim \frac{1}{r^n}$ with $n \geq 2$) and the severe energy restrictions of smart objects, it also seems to be a fundamental property of radio-based tags that they cannot communicate with remote backend servers on their own. Optical tags, however, can exchange data with remote servers more efficiently, but have other disadvantages such as their line-of-sight restrictions (cf. Sect. 2.2.3).

Disconnected Operations

A problem with mobile infrastructure access points can arise if the internal state or environmental parameters meaningful in the context of a smart object's application change when no people are nearby. In this case, smart objects operate in a disconnected mode as long as there is no handheld device in vicinity. Considering the memory restrictions of smart objects, this can pose difficulties because it requires that augmented items must buffer data until a connection with the backend infrastructure can be established. Furthermore, if people want to interact with an augmented item from remote locations, their requests cannot address the smart object directly as there is no direct connectivity. A similar problem arises if smart objects that are far away from each other want to communicate via the backend infrastructure.

Our approach to dealing with disconnected operations is to associate a smart object with a backend service that represents the augmented item continuously. We call this service the background infrastructure representative (BIRT) of a smart object. Whenever there is connectivity by means of a nearby handheld, a smart object synchronizes its state with its BIRT, and vice versa. On the other hand, when there is no connectivity and people or other devices want to communicate with the object, the BIRT buffers these data and transmits them to the object during the next synchronization. The BIRT can also emulate the services provided by a smart object based on the data it received during the last synchronization (see Fig. 4.4). Please see Sect. 4.6.2 for an example of using BIRTs in a concrete application scenario.

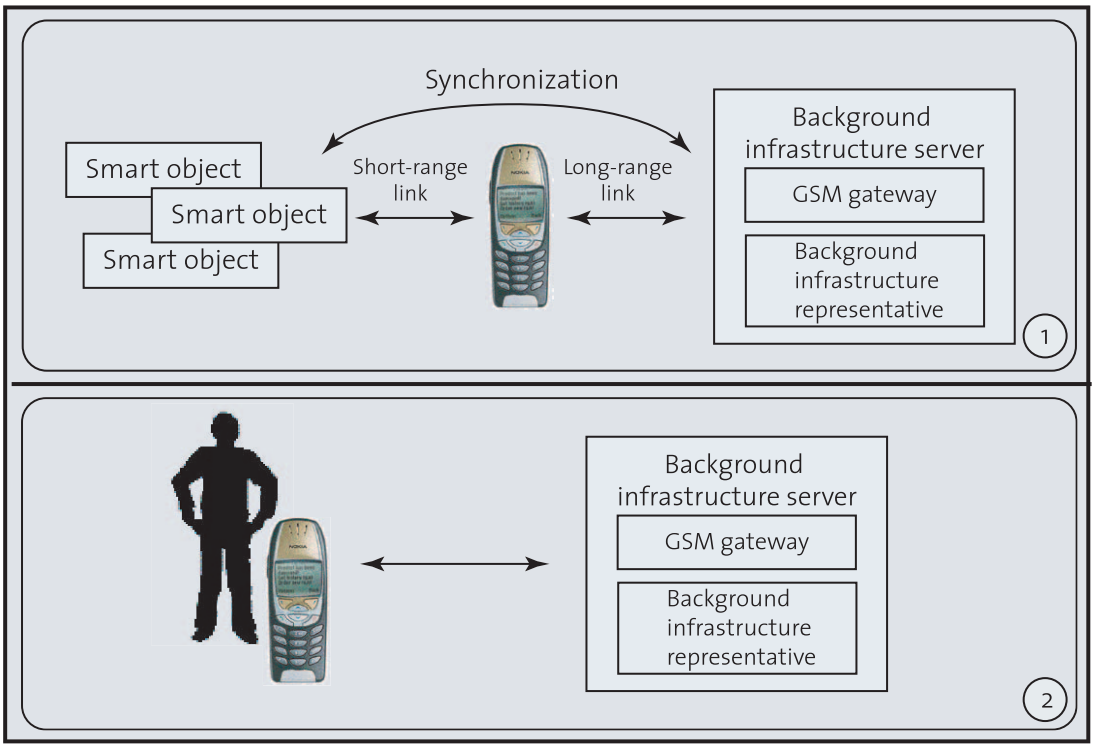

Figure 4.4: Disconnected operations: if a mobile access point is in range of a smart object, it synchronizes its state with its BIRT (1); if there is no direct connectivity to the smart object, requests from remote users and other augmented artifacts address the BIRT of the smart object (2).

Implementation

The following illustrates how a handheld's role as a mobile infrastructure access point can be realized. In order to demonstrate implementation specific issues with a concrete example, we have chosen mobile phones as handheld devices and BTnodes as tags for augmenting everyday objects. In general, using a cellular phone as a mobile access point for a smart object involves three steps: (1) the smart object creates a local Bluetooth connection to the phone, (2) the smart object transmits a command sequence to the mobile phone in order to establish a long-range connection from the handheld to a backend server, and (3) the mobile phone relays data originating from the smart object to the backend server and vice versa (cf. Fig. 4.5). With respect to our choice of Bluetooth-enabled mobile phones and BTnodes, all of these steps can be taken without changing the functionality of or uploading additional software to the user's mobile phone.

Step (1), the establishment of the local Bluetooth connection to the handheld, is carried out after a smart object has discovered a new handheld in its environment. If unknown, the Bluetooth service discovery protocol (SDP) can be used to query for appropriate parameters that are then used during the actual connection establishment procedure. As we connect to a service of the mobile phone that allows smart objects to establish long-range connections (which therefore results in costs for the owner), a smart object must be authenticated to use this service. The necessary data for creating authenticated connections are determined during a previous Bluetooth pairing procedure. Although the pairing procedure must be completed only once, it requires explicit user input in form of a PIN number.

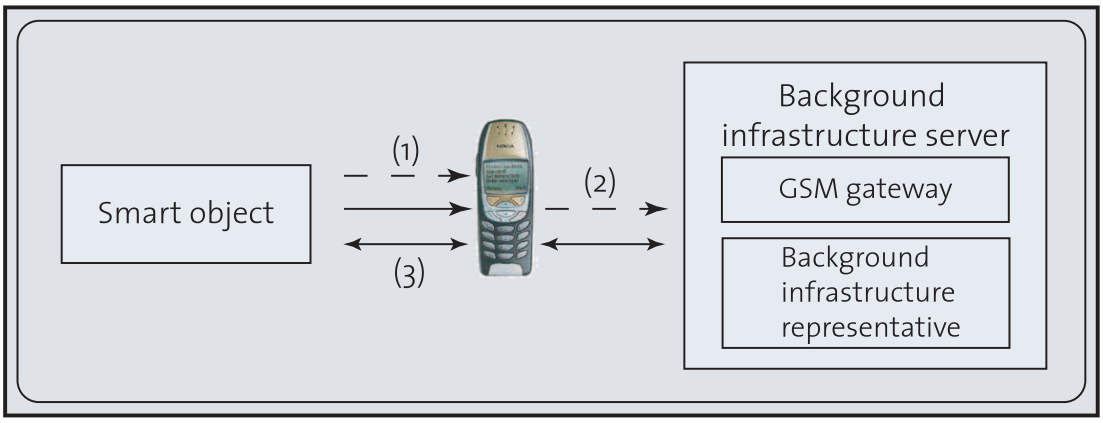

Figure 4.5: Deploying nearby cellular phones as mobile infrastructure access points.

Given the local Bluetooth link, the smart object establishes a connection from the handheld to the backend server. This backend server is connected to a second mobile phone that serves as a GSM gateway. The smart object can therefore establish a long-range connection between its local mobile access point and the remote server by calling the GSM gateway of the remote server. This is done by sending AT commands over the local Bluetooth link from the smart object to the mobile phone that serves as its mobile infrastructure access point. There is a standardized set of AT commands supported by all GSM-enabled mobile phones, which ensures that this can be done with virtually all of today's cellular phones in Europe.

Given the short-range Bluetooth connection to the access point and the long-range link to the backend server, arbitrary data can be exchanged between the smart object and the background server. Besides using explicitly established GSM data connections between the mobile access point and the infrastructure server, it is also possible for smart objects to embed data into SMS (Short Message Service) messages. SMS messages are transmitted via the GSM infrastructure to the backend server, which in turn responds by sending SMS messages back. As this approach does not require the overhead of GSM data connection establishment, we have prototypically implemented this SMS-based approach. It is also the preferred way to exchange data with a background infrastructure server in our applications (see Sect. 4.6). In a recent student thesis supervised by Oliver Kasten, a handheld's role as a mobile infrastructure access point has also been implemented using explicitly established GSM data connections.

4.4.2 User Interface

Basic Concept

The user input and output capabilities of handheld devices can be exploited by smart objects for realizing interactions with nearby people based on a graphical user interface. Mobile phones and PDAs offer several popular features by means of which such an interaction can be realized. They range from custom alarms, SMS messages, WAP pages, calendar entries, and business cards to whole Java user interfaces that can be downloaded over a local connection from a smart object to a handheld device. Realizing user interfaces is one of the core reasons for smart objects to cooperate with handheld devices because the input and display capabilities of augmented items themselves are usually

severely limited. By realizing interactions with smart objects based on widely-accepted usage patterns people associate with their handheld devices (e.g., making phone calls or writing SMS messages), it can become easier for people to deal with Pervasive Computing technologies.

Motivation

The computing platforms embedded into everyday objects need to be small and unobtrusive. Consequently, active and passive tagging technologies generally do not possess screens, buttons, or any other means to process explicit user input and to display output. The tags are ideally invisible to users and unobtrusive to such a degree that they do not disturb the way in which people would normally use these objects if they were not augmented with information technology. In other words, intelligent tags should add functionality to everyday objects without changing their appearance. But if augmented items have such severely limited input and output interfaces, how do users actually interact with smart objects?

To address this question, we classify user interactions with smart objects into three categories: (1) implicit interaction (sometimes also called invisible interaction), (2) explicit interaction, and (3) hybrid approaches to interaction [SF03a].

Implicit Interaction. Implicit interaction is an interaction paradigm where people do not use explicit actions to communicate with the information technology embedded into everyday objects. Instead, based on their ability to sense their environment, augmented items monitor the behavior of nearby people and the way in which they deal with smart artifacts in order to derive instructions for a smart object's behavior. The people do not control smart objects explicitly but through side effects of their ordinary behavior: although users' actions are not intended to control a smart object, an augmented item does draw conclusions for its own behavior from these actions. A simple example of implicit interactions are automatic doors, which can be regarded as everyday objects augmented with small movement sensors. Interactions with automatic doors are implicit because users do not need to carry out explicit actions to open them. Instead, the doors monitor their immediate environment, and automatically derive instructions from the behavior of nearby people. That is, when people approach an automatic door it opens automatically. Regarding implicit interactions, users might not be aware of the effects their behavior is having on a smart object because the intention of their actions is not to interact with augmented artifacts. With respect to our example, the action that opens an automatic door, for instance, is not associated with the door itself, but simply reflects a person's intention to enter a building. The action *approaching the entrance of a building* is mapped by the smart object onto an instruction to open the door, without requiring explicit input from the user. Considering this example, it becomes clear that in order to take part in implicit interactions smart objects must be able to derive their situational context and that of nearby people. In Chap. 3 we have already shown how smart objects can obtain information about their environment by cooperating with other objects. The approaches presented can be used to realize implicit interactions in smart environments.

Explicit Interaction. Explicit interaction is the predominant interaction paradigm with today's personal computers and, as the name implies, is based on explicit user input. By pressing buttons, by moving a computer mouse, and by using other pointing devices or

speech processing software, people interact with computing devices and explicitly trigger their actions. This has the advantage that users remain in full control of their computing devices, can easily influence their behavior, and better understand their actions. In contrast, implicit interactions might result in annoying and unwanted actions taken by computing devices if they misinterpret the instructions hidden in a user's behavior. For example, automatic light switches turn on a lamp without requiring users to search for a switch in the dark and without explicit actions, which certainly is a nice feature. However, if a user wants to say goodbye to his girlfriend (in which case darkness is quite desirable) automatic light switches can also become quite annoying. Implicit interactions can also pose considerable privacy concerns if smart objects do what they think is appropriate without explicit feedback from users. On the other hand, if everyday environments are populated with hundreds or even thousands of computing devices, many of which might be invisible from a user perspective, explicit interactions are likely to become impractical. In such settings it seems not to be feasible to demand explicit actions from nearby users to control every aspect of smart objects' behavior.

Hybrid Approaches. Hybrid approaches to interaction try to combine the advantages of both previously-mentioned interaction paradigms. In the concept of invisible preselection, for example, which we have proposed in [SF03a], people continue to interact with smart objects based on explicit user inputs. However, in a preselection phase, implicit interaction is used to choose interaction partners.

In summary, both explicit and implicit interaction should be supported by smart objects in order to cover a broad range of application scenarios. While Chap. 3 illustrated how to derive environmental context that facilitates implicit interactions, nearby handhelds can be used by smart objects for realizing explicit user interfaces. As augmented items do not possess user interface capabilities on their own, cooperation with other types of computing devices is the only way to access such user interfaces. One advantage of handhelds in this respect is again their high availability – i.e. people can access their handhelds at almost any time, independent of their current location. Another, even more important benefit is that most people know how to operate mobile phones and PDAs even though they often run into problems with other more complex computing devices. If smart objects provide services that draw upon the user interface capabilities of handhelds, people might find it easier to deal with smart environments.

Implementation

Handheld devices such as mobile phones and PDAs offer a wide range of features that can be exploited by smart objects to realize user interfaces. These features include (1) custom alarms, (2) SMS messages, (3) WAP pages, (4) calendar entries and business cards, and (5) whole Java user interfaces that can be downloaded over a local connection from a smart object to a handheld device. We have evaluated and prototypically implemented all these different means to facilitate user interaction with smart objects. BTnodes have been used as a prototyping platform for augmenting everyday objects, and Bluetooth-enabled mobile phones and PDAs as handheld devices.

Custom Alarms. Custom alarms as they are supported by many of today's mobile phones can be used to remind people about actions associated with a smart object. For example in the scenario presented in Sect. 4.6.2, a smart medicine cabinet automatically

stores an alarm in the mobile phone of a patient in order to remind her to take the medicine she has recently removed from the cabinet. Custom alarms have the disadvantage that they often do not display a message when the alarm is triggered. Such additional functionality, however, is provided by calendar entries. With respect to our prototypical implementation, alarms are written to a cellular phone by transmitting standardized AT commands from a smart object over a local Bluetooth connection to the phone. Thus, smart objects first need to establish a local Bluetooth link to the mobile phone, retrieve the phone's local time in a second step, and then send the time at which the alarm is to be triggered (cf. Fig. 4.6). As previously mentioned, the transmission of AT commands to a mobile phone requires an authenticated connection.

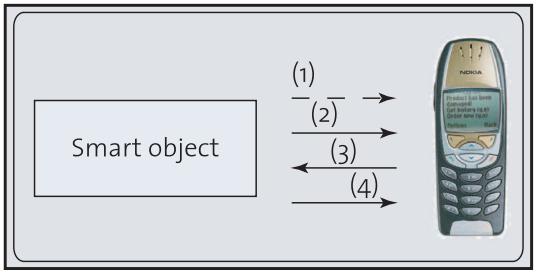

Figure 4.6: Using nearby handhelds for reminding people about actions associated with a smart object: a smart object (1) establishes a Bluetooth link, (2) requests the handheld's local time by sending appropriate AT commands, (3) receives the local time of the handheld, and (4) schedules an alarm by transmitting AT commands over the local Bluetooth connection.

Calendar Entries and Business Cards. Calendar entries and business cards can be exchanged via Bluetooth OBEX (Bluetooth Object Exchange Protocol) with Bluetooth-enabled mobile phones and PDAs in range of a smart object. Calendar entries are an interesting alternative to custom alarms (see previous paragraph). Their advantage is that they not only trigger an acoustic alarm at the time specified but also display information associated with the alarm on the handheld's screen. In contrast to calendar entries, business cards are useful for sending small amounts of information to a nearby handheld device that are immediately displayed on the handheld's screen. The benefit of business cards is that the local Bluetooth link over which they are transmitted does not necessarily need to be authenticated. Fig. 4.7 depicts examples of calendar entries and business cards that have been sent by a smart object to a Bluetooth-enabled mobile phone.

SMS-based Interaction. According to forecasts by GSM world [GSM04], 547.5 billion SMS messages are going to be sent in the year 2004. Such figures underline the overwhelming economic success of the short message service and its broad acceptance by today's mobile phone users. Motivated by these facts, we investigated how the short message service can be used to enrich interactions with smart objects. Due to their limited text-based interface, SMS messages as they are supported by virtually all of today's mobile phones are most suited for transmitting short notifications. However, they also allow more complex interactions that involve consecutive messages exchanged between smart objects and users. The basic idea for realizing SMS-based interactions is to assign phone numbers to smart objects. Using these phone numbers as an addressing scheme, aug-

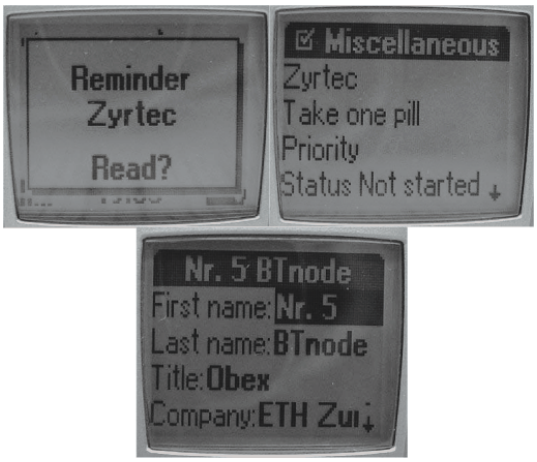

Figure 4.7: Using calendar entries and business cards in interactions with smart objects: a calendar entry stored into a mobile phone from a smart object to remind a patient to take the medicine Zyrtec (top), and a business card with basic information about a smart object (bottom).

mented items cannot only trigger the transmission of text messages but can also receive response messages from users.

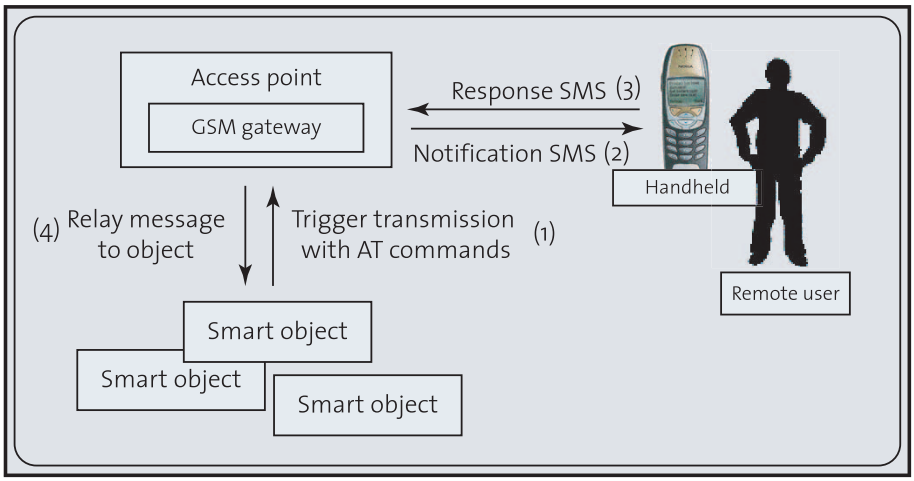

Figure 4.8: Sending SMS messages to remote users.

The implementation setup for supporting SMS-based interactions depends on whether the user is in direct transmission range of an object or at a remote location. We will first describe how augmented items can communicate with remote handhelds (cf. Fig. 4.8). In this case, smart objects initiate the exchange of SMS messages with remote users by sending AT commands to a nearby access point over an authenticated short-range Bluetooth connection. The access point can be stationary, but it can also be a nearby cellular phone that serves as a mobile infrastructure access point. Our prototypical implementation for the BTnodes supports both alternatives. The access point is responsible for transmitting the SMS message to the remote user's mobile phone via the public GSM infrastructure. Having received the message, the user can access the embedded information and send a response message back to the smart object. The phone number to which to send this message is implicitly given because it is the number the first message originated from.

The user can therefore respond by simply replying to the first message. Technically, the response SMS message is sent back to the access point in range of the augmented items, which relays the incoming message to the appropriate smart object. As an access point potentially serves many different smart objects, it must analyze the content of messages in order to determine the intended receiver.

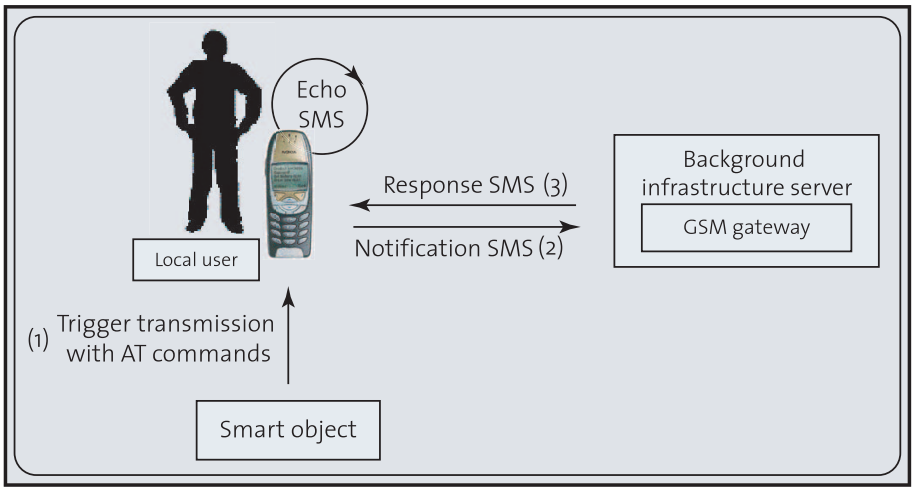

Figure 4.9: Sending SMS messages to local users.

If people are in direct transmission range of a smart object, interaction based on the short message service is possible without the access point necessary in the previously described approach. As can be seen in Fig. 4.9 a smart object initiates the transmission of an SMS message by sending AT commands directly to the user's mobile phone. However, the message sent cannot be directly displayed on the handheld but must be passed over the public GSM infrastructure. This implies that although the interaction appears to be local from the point of view of a user, it nevertheless relies on an infrastructure and results in costs imposed on the user. There are two possibilities for sending the message via the GSM infrastructure. The first is to simply echo the message, meaning that the smart object sends an SMS from the user's mobile phone to the user's mobile phone. The second solution involves a background infrastructure server with a GSM gateway. Here, the augmented item would send a notification message to the backend server that responds by sending an SMS back to the user's mobile phone. The advantage of this approach is that the response message from the backend need not be the same as the notification message sent by the smart object, but could be enriched with more information by the backend.

The drawbacks of using an SMS-based approach for user interactions with smart objects are the costs of transmitting messages via the GSM infrastructure and the necessity of authenticated connections. With respect to the costs of sending SMS messages, we would like to argue that prices are likely to fall in the future. This is because new multimedia services and new generations of mobile phones will eventually decrease the cost of transmitting small amounts of data over mobile phone networks. On the other hand, the costs incurred by transmitting SMS messages could even be used to charge for services provided by smart objects. In our prototypical implementation, local Bluetooth links need to be authenticated in order to prevent arbitrary objects from sending messages at the expense of mobile phone users. Here, the disadvantage of authenticated connections is that people must explicitly establish a trust relationship with every smart object

they want to interact with. As there are potentially vast amounts of objects in a user's environment, this can cause an unacceptable overhead for the user. However, after a relationship between an object and a mobile phone has been established once, consecutive connection establishments do not require manual interactions. Furthermore, there are several techniques for passing trust relationships existing between two entities on to other devices [Roh03, Sta00, Sta02]. Using such approaches, it would not be necessary to manually establish such a relationship between a handheld device and every smart object it needs to cooperate with. Instead, trust relationships could for example be transitive, meaning that if a handheld H trusts a smart object O_1 and O_1 trusts another object O_2, a trust relationship between H and O_2 could automatically be deduced. There are also other, more user-friendly ways for coupling devices that do not require the input of pin numbers. Holmquist et al. [HMS+01], for example, suggests establishing a relationship between smart objects by shaking them together.

WAP Interfaces. User interaction with smart objects can also take place based on WAP (Wireless Application Protocol) interfaces. In our implementation, user interaction with WAP relies on a background infrastructure service, the *background infrastructure representative* (BIRT) of a smart object. As previously described, a BIRT mirrors the state of a smart object and can therefore represent an object in the backend infrastructure in case there is no direct connectivity to the augmented item itself. Another application domain of BIRTs is to exchange data between the smart objects they represent and other entities (e.g., other smart objects, handheld devices, or users) that want to interact with them. The BIRT can transform data it received from a smart object into a more legible format that is easily accessible for people. In case of WAP user interfaces, the BIRT transforms data from smart objects into WML (Wireless Markup Language) pages that are displayed on a user's handheld device.

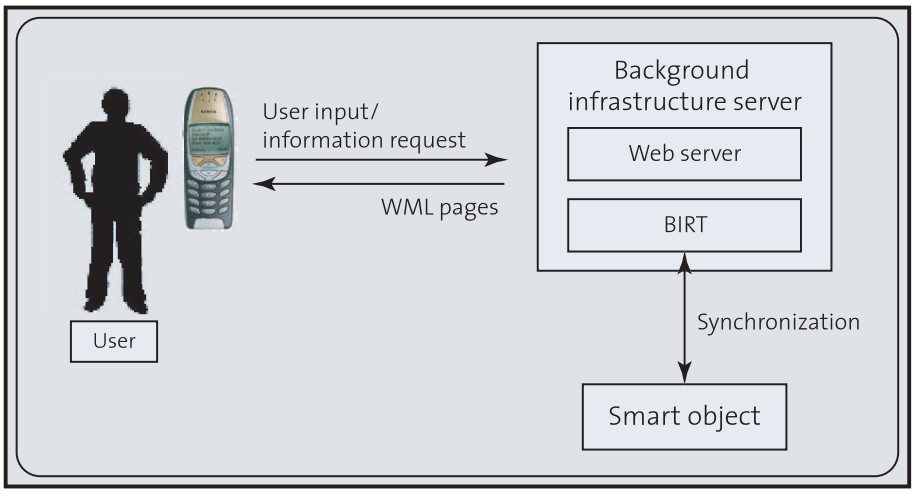

Figure 4.10: WAP-based interaction.

The setup for realizing WAP-based user interfaces for interactions with augmented items is as follows. As can be seen in Fig. 4.10, smart objects synchronize their state with that of their BIRT whenever it becomes necessary – i.e., whenever their application status changes. If this is not possible because of temporary disconnections, augmented items should buffer data until a new link with the backend server hosting the BIRT can be established. The BIRT is responsible for transforming data provided by a smart

object into a WAP format, i.e., into WML (Wireless Markup Language) pages. For this purpose, we have implemented a BIRT as a Servlet [Ser04] that cooperates with a Web server running on the same machine. If a user wants to interact with a smart object, she opens a URI (Uniform Resource Identifier) associated with that object in a WAP browser on her mobile phone. As a result, the Web server running on the backend infrastructure server receives a request, whereupon the BIRT of the addressed object dynamically generates a WML page containing the requested information. This WML page is then displayed on the user's mobile phone.

The mechanism presented not only allows people to retrieve information about an object, but also allows users to send information back to it. In our approach, a BIRT generates WML pages containing form fields that can be used to input data. These data are transmitted back to the BIRT, which relays them to the smart object. In this way, the behavior of a smart object can be controlled with a WAP-based user interface.

Java User Interfaces. Finally, we have also developed concepts for supporting more elaborate interactions with smart objects. Here, Java classes that represent graphical user interfaces are directly stored on smart objects. People can select augmented items in their environment using their handheld device and download code containing the user interface from the selected object. Using Java programs, more sophisticated graphical user interfaces containing widgets such as menu bars and drop-down boxes can be realized. The concepts underlying this approach are described in more detail in Sect. 4.5.

Applicability

Tab. 4.3 compares the described approaches for realizing user interactions with smart objects. Its goal is to present the properties that most significantly influence the choice for an appropriate interaction mechanism when designing applications for smart environments. As can be seen, SMS messages and WAP interfaces are suitable for implementing interactions with objects at remote locations, whereas business cards, for example, are usually exchanged with people that are in close proximity of a smart object. Furthermore, some of the interaction concepts presented are uni-directional. Custom alarms and calendar entries, for instance, are very effective for notifying and reminding people, but they are not suited for sending information back to the smart object and do not, therefore, support information exchange. In contrast, with a WAP interface data can be sent from a smart object to a user and vice versa. Tab. 4.3 also shows if authenticated connections are necessary to realize the presented interaction concepts.

4.4.3 Remote Sensor

Basic Concept

The percepts of handheld devices are of potential interest for smart objects, which often simply do not have sufficient resources to deploy sophisticated sensors. Mobile RFID-readers, for example, consume significant amounts of energy and are therefore difficult to integrate into mobile everyday objects. Barcode scanners and digital cameras are also often attached to or integrated into handheld devices because their complexity and size makes it difficult to embed them into everyday things. Besides smart objects' resource restrictions, there is also another reason why sensory data from mobile user devices might be valuable for augmented items. This reason is that handhelds are usually in close

Table 4.3: Comparison of the approaches for accessing handheld's user interfaces from smart objects.

Approach	Remote interaction	Interaction	Authentication
Alarms	no	one-way	required
Business cards	no	two-way	not required
Calendar entries	no	one-way	required
SMS	yes	two-way	required
WAP	yes	two-way, user-initiated	required
Java user interfaces	no	two-way	not required

proximity of their owners and can therefore provide more accurate information about them. Sensory data originating from smart objects that are farther away from users than the handheld devices they carry might be less suited for characterizing their current situation. In the TEA project [TEA04], for example, sensors have been attached to a mobile phone in order to determine the current situational context of its owner. Sensory data from a handheld device are thereby well suited to determine the current context of people, and enable smart objects to provide better customized context-aware services.

Implementation

In the following, we describe how smart objects that have been augmented with BTnodes can use ordinary mobile phones as remote location sensors. For this purpose we exploit the cellular topology of GSM networks and the fact that mobile phones know which cell they currently belong to (i.e., to which base station they are currently assigned). When BTnodes want to determine their approximate position, they try to establish an authenticated Bluetooth connection to a nearby mobile phone, and retrieve its cell identifier by exchanging AT commands on top of the local Bluetooth communication link. As a GSM cell can cover several square kilometers but the Bluetooth transmission range is only several meters, both the smart object and the handheld device are typically in the same cell. Hence, the location information provided by the handheld can be used to characterize the location of the smart object. Sect. 4.6.4 describes an application where handhelds serve as location sensors for smart objects.

Smart objects can also access sensory data from handheld devices using a similar approach to that presented in Chap. 3. Here, mobile user devices join a distributed data structure established by smart objects, and exchange sensor values by means of this data structure. In our implementation, both, PDAs and smart objects can participate in a distributed tuplespace, which allows them to exchange sensory data in this way.

4.4.4 Mobile Storage Medium

Basic Concept

Data transmitted from a smart object to a handheld device are available to users independent of their current location and their overall situational context. This is because

handhelds (especially mobile phones) are carried around by their owners, who can therefore access data stored on them at almost any place and any time. This feature can be exploited in applications that rely on remote interactions with smart objects. In such applications, one of the main difficulties is to address a specific smart object that is not available for physical interaction because it is at a remote location. Laser pointers to select objects over small distances [Rin02] or techniques that use physical contact to initiate interactions are therefore not applicable in these settings. In our applications (cf. Sect. 4.6) we solve this problem by using handheld devices as mobile storage mediums for so-called *interaction stubs* from smart everyday objects. In this solution, smart objects that offer services which can be accessed remotely store their contact information on users' handhelds when the users are in range of these objects. When people afterwards want to access services from remote locations, they can establish a connection with the corresponding smart object using the data stored on their handhelds. In our approach, we tried to store contact information in a format that reflects the basic functionality of the mobile user device. For example, in the case of mobile phones we store contact information for smart objects in the form of phone book entries.

Implementation

In the following, it is shown how we have prototypically implemented the role *mobile storage medium* using mobile phones as handhelds and BTnodes as tagging technology. When cellular phones serve as handheld devices, the question arises in which data format smart objects can actually store information on them. To enable easy accessibility from a user perspective and in order to support a broad range of mobile phones, we chose to store information on mobile phones as phone book entries and SMS templates. Using phone book entries, people can establish a connection to a smart object that can be addressed using a phone number (cf. Sect. 4.6.1); an SMS template can contain arbitrary textual data from a smart object. In our implementation, smart objects establish an authenticated Bluetooth connection to a nearby mobile phone when they need to store information on it. Afterwards, AT commands are transferred over the just-established local Bluetooth link to store and retrieve data from the handheld device. Afterwards, the smart object shuts down the Bluetooth connection.

4.4.5 Remote Resource Provider

The resource restrictions of smart everyday objects are a core motivation for augmented items to cooperate with handheld devices. If handhelds serve as remote resource providers, they make their computational resources available to collections of nearby smart objects and help them to implement computationally more demanding services. Because the mechanisms for integrating handhelds into sets of cooperating objects are relatively complex, we have devoted an entire section to this issue (cf. Sect. 4.5). In the following, we only present the basic concepts underlying our approach.

Basic Concepts

The central purpose of handheld devices in the previously described usage patterns is to mediate between smart objects and their users, or between smart objects and background infrastructure services. As *remote resource provider*, however, a handheld primarily enriches the interaction among different smart objects in that it provides a platform

for outsourcing complex computations and offers sophisticated data storage capabilities. Our goal was to spontaneously integrate handheld devices into already existing groups of collaborating smart objects.

This goal is achieved by introducing an infrastructure layer facilitating the collaboration between different computational entities. This layer has been realized as a distributed tuplespace [DWFB97, Gel85] for smart objects and handheld devices, which is a part of our prototypical implementation. Smart objects that want to collaborate establish a tuplespace and write their sensory data into the space. When a handheld device comes into the range of collaborating objects, it also joins this distributed tuplespace. As a result, smart objects can instantaneously make use of the memory resources of handheld devices on the basis of resource-aware tuplespace operations. Resource-aware tuplespace operations try to identify the most suitable place to store sensor tuples. As the actual location of a tuple becomes transparent through the shared data structure, tuples are stored on the device having the largest amount of spare resources – which is often the handheld.

The most important reason for introducing the tuplespace abstraction, however, is that it becomes transparent where program code is executed. This is because all devices operate on the same data basis of the shared data space. Smart objects can therefore transfer code for complex and energy-consuming computations to a nearby handheld device participating in the same tuplespace and thereby exploit its computational resources. In our implementation of this concept, Java classes are stored on a smart object and transmitted to nearby handheld devices when a handheld joins the shared data space (cf. Sect. 4.5).

4.4.6 Weak User Identifier

Basic Concept

A smart everyday object usually implements context-aware services, which adapt according to the situation of nearby people in order to provide functionality tailored towards their current needs. Knowledge about the identities of nearby users helps smart objects in realizing personalized services. The identity of a user is also a core aspect of context and often plays an important role in context-aware applications.

Handheld devices can help smart objects to determine the identity of nearby persons in several ways. First, the mere presence of a particular handheld is often sufficient to determine who is currently in range of a smart object. This requires a simple one-to-one mapping between the addresses of mobile user devices and the names of known persons. Although such a mapping exists for most handheld devices because they often belong to one person exclusively, it is easy for an adversary to assume the identity of other users. This is also the reason why we speak of *weak* user identification. Second, handheld devices offer several mechanisms for user identification, for example PINs or finger print sensors. If users must be authenticated based on these mechanisms in order to operate handheld devices, it is much safer to infer the identity of a nearby person from the presence of a handheld device. However, from the perspective of a smart object, the downside of using an external identification mechanism is the necessity to trust in the handheld's correct operation (and in the user to keep the PIN secret). Third, wireless communication technologies such as Bluetooth offer mechanisms to authenticate connections. These mechanisms involve both the smart object and the handheld device and rely not only on

the presence of other devices (i.e., the presence of devices with certain MAC addresses) but also check whether devices are the ones they pretend to be.

Implementation

The software package we developed for the BTnodes to illustrate the roles of handhelds in smart environments supports authentication using PIN codes as part of the implemented Bluetooth protocols. Identification based on the mere presence of other devices is easy to implement as it just requires a mapping from handhelds' Bluetooth device addresses to user names. Bluetooth-enabled handheld devices in range of a smart object can be found using the Bluetooth inquiry procedure.

4.5 Integrating Handhelds into Environments of Cooperating Smart Everyday Objects

This section focuses on a handheld's role as *remote resource provider*, and presents techniques that allow smart objects to dynamically access the computational capabilities of mobile user devices. The computational abilities of handhelds (especially their processing power and data storage capabilities) thereby facilitate the cooperation between smart objects by providing a powerful computing platform for augmented items. Our goal is to integrate handheld devices into collections of cooperating smart everyday objects, not the interaction with a single augmented item. In the proposed approach, handhelds join a distributed data structure shared by cooperating smart objects, which makes the location where data are stored transparent for applications. Smart objects then outsource computations to handhelds and thereby gain access to their resources. As a result, this allows smart items to transfer a graphical user interface to a nearby mobile user device, and facilitates the collaborative processing of sensory data because of the more elaborate storage and processing capabilities of handhelds. In order to evaluate the applicability of our approach, we present a prototypical implementation of our concepts on an embedded sensor node platform, the BTnodes.

4.5.1 Basic Concepts and Architecture Overview

In our vision, smart environments are populated by smart objects that provide context-aware applications to nearby users. Due to their resource restrictions, smart objects thereby need to cooperate with other objects, for example during the context-recognition process, in order to exchange and fuse sensory data. To enable cooperation among different devices, smart objects establish a shared data space for exchanging sensor values and for accessing remote resources. We have implemented such a shared data structure as a distributed tuplespace for the BTnode embedded device platform and handheld devices [KS04]. Here, each node contributes a small subset of its local memory to the distributed tuplespace implementation. Using the tuplespace as an infrastructure layer for accessing data, the actual location where data is stored becomes transparent for applications. The data space hides the location of data, and tuples can be retrieved from all objects that cooperate with each other. Consequently, an application that operates on data in the distributed tuplespace can be executed on every device participating in that shared data structure. The actual node at which it is executed becomes irrelevant. Hence, when a

handheld device joins the distributed tuplespace shared by cooperating smart objects, applications developed for a specific smart item can also be executed on the handheld device.

We have realized our concepts for integrating handhelds into environments of cooperating smart objects in a software framework called *Smoblets*. The term *Smoblet* is composed of the words *smart object* and *Applet*, reflecting that in our approach active Java code is downloaded from smart objects in a similar way to that in which an Applet is downloaded from a remote Web server. Fig. 4.11 depicts the main components of the Smoblet system: (1) a set of Java classes – the actual Smoblets – stored in the program memory of smart objects, (2) a Smoblet frontend that enables users to initiate interactions with nearby items, (3) a Smoblet runtime environment for executing Smoblets on a mobile user device, and (4) a distributed tuplespace implementation for smart objects and handheld devices.

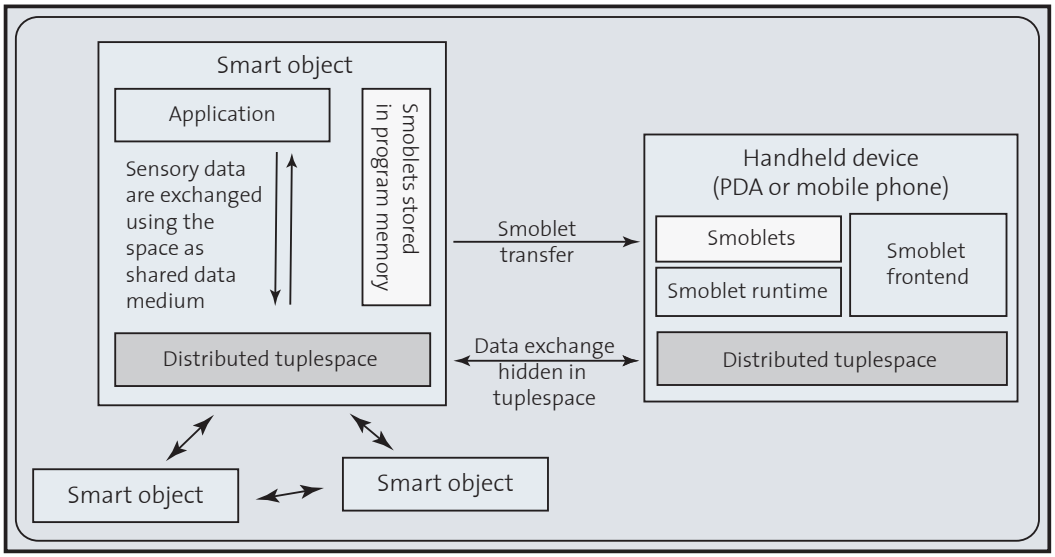

Figure 4.11: Overview of the Smoblet system.

Smoblets. The code that is transferred to a nearby user device – in the following referred to as Smoblet – consists of Java classes that are developed on an ordinary PC during the design of a smart object. However, as the smart objects themselves cannot execute Java code, Smoblets encapsulate computations that are designed to run on a more powerful device while still operating on the data basis of the smart object that provided the code. The sole reason for storing Smoblets in the program memory of augmented artifacts is the memory architecture of many active tagging systems, which often have significantly more program memory than data memory. The BTnodes, for example, offer 128kB of program and 64kB of data memory. Only about half of the program memory on the BTnodes is occupied by typical programs, which leaves ample space for storing additional Java code.

Front- and Backend. The Smoblet backend is responsible for executing downloaded code on a handheld device. It also protects the user device from malicious programs and enables downloaded Java classes to access data on other platforms by providing an

interface to the distributed tuplespace implementation. In contrast, the Smoblet frontend helps users to search for smart objects in the vicinity, to explicitly initiate downloads, and to customize the behavior of the Smoblet backend system.

Distributed Tuplespace. The distributed tuplespace is the core component for integrating handhelds into collections of cooperating objects. Its main purpose is to hide the actual location of data from an application, which makes it possible to execute code on each of the cooperating nodes. Hence, it is possible to design applications that are executed on a handheld device but still operate on the data basis of the smart object the code originates from. The distributed tuplespace also allows cooperating objects to share their resources and sensors. The memory of other objects, for example, can be used to store local data, and it is possible to access remote sensor values. In our application model, smart objects specify how to read out sensors and write the corresponding sensor samples as tuples into the distributed data structure, thereby sharing them with other objects. Please refer to Sect. 3.5 and [Sie04] for a more detailed description and an evaluation of our tuplespace implementation for the BTnodes. In order to build a running Smoblet system, we have ported our implementation to Windows CE. This allows handheld devices to participate in the shared data structure.

4.5.2 The Smoblet Programming Framework

Besides the components for exchanging and executing Smoblets, the Smoblet programming framework supports application developers in realizing the corresponding code for smart objects. Deploying Smoblets involves four steps (cf. Fig. 4.12): (1) using a set of helper classes, Java code containing the computations to be outsourced is implemented on an ordinary PC, (2) the resulting class files are embedded into C code and (3) linked with the code for the basic application running on the embedded sensor node, and (4) the resulting file is uploaded to the smart object's program memory.

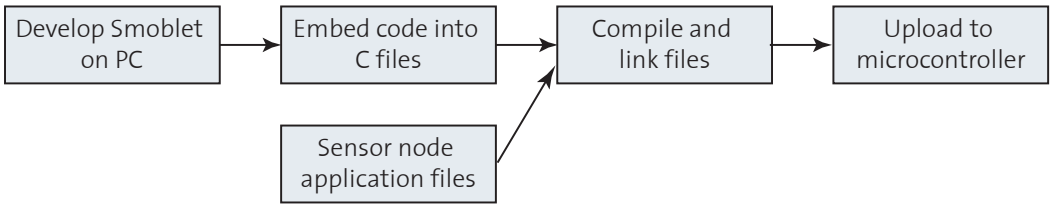

Figure 4.12: The process of deploying a Smoblet on a sensor node.

In the following, we concentrate on the development of the actual code that is outsourced to a nearby mobile user device. This is done on an ordinary PC using Java and a set of supporting classes provided by our framework. Among other things, the framework provides classes for retrieving information about the current execution environment of a Smoblet, for controlling its execution state, and for exchanging data with nearby smart objects. We distinguish between two application domains for Smoblets: applications in which smart objects transfer whole user interfaces to nearby handhelds in order to facilitate user interaction, and applications for outsourcing computations in order to exploit a handheld's computational abilities, without requiring user interaction. These two application domains are represented by the *GraphicalSmoblet* and *BasicSmoblet* classes, which directly inherit from the superclass *Smoblet*. All user-defined code that is

to be outsourced to a nearby handheld must inherit from these classes. This can be seen in Fig. 4.13, which shows selected parts of a Smoblet for collecting microphone samples from nearby smart objects.

```
    import smoblet.*;

    public class MicCollector extends BasicSmoblet {

5:      public String getSmobletName() {
            return "MicCollector";
        }

        public boolean isAutoStart() {
            return true;
11:     }

13:     public boolean onInit() { ... }

        public boolean onRun() {
            while (extractFeatures) {
17:             res = consumingScanTupleDts(micFeature, 20);
                Thread.sleep(2000);
            }
            return true;
21:     }

23:     public void onHomeDeviceLost() { ... }
    }
```

Figure 4.13: Selected methods from a Smoblet without graphical user interface.

There are four basic categories of methods provided by the *Smoblet* class and its subclasses: (1) informational methods, (2) methods for controlling the execution state of a Smoblet, (3) event reporting functions, and (4) functions for accessing data in the distributed tuplespace. Informational methods provide information about the environment of a Smoblet and about the Smoblet itself – for example its name, a user-friendly description of its functionality, information about its requirements regarding a host platform, and whether it should be automatically started after download to a handheld device (cf. Fig. 4.13 lines 5-11). These informational methods are mainly used by the tool that embeds Smoblets into C code for storage on a smart object. This tool for converting Java files loads Smoblet classes, retrieves their informational parameters, and stores this information in addition to the encoded class files on a sensor node. This has the advantage that the sensor node can easily access this data and disseminate it to other nodes. Consequently, when a user wants to lookup Smoblets on a particular smart object, it is not necessary to download the complete Smoblet to get information about its requirements and other characteristic parameters. Instead, smart objects share these data with nearby handhelds by means of the distributed data structure.

The methods controlling a Smoblet's execution state are executed by the runtime environment on a mobile user device after the code has been successfully downloaded from a smart object. The *onInit* method, for example, is executed immediately after a Smoblet has been started, followed by the *onRun*, *onAbort* or *onExit* methods (cf. lines 13-21 in Fig. 4.13). Every Smoblet is executed in a separate Java thread.

Event reporting functions are executed after certain events on the handheld or in its environment have occurred. For example, when the device the code stems from leaves the communication range of the handheld or when new devices come into wireless transmission range, an event is triggered and the corresponding method executed (cf. line 23 in

Fig. 4.13). Event reporting functions are also used to handle callbacks registered on the shared data space. A handheld participating in the space can specify tuple templates and register them with the distributed tuplespace. When data matching the given template is then written into the space, an event reporting function having the matching tuple as an argument is invoked on the handheld.

The last category of methods provided by the *Smoblet* class are methods for accessing the shared data structure. These come in three variants: operations for accessing the local data store, i.e. the local tuplespace; operations for accessing the tuplespace on a single remote device; and functions operating on the tuplespaces of a set of cooperating smart objects. The MicCollector Smoblet, for example, retrieves microphone tuples from all nodes participating in the shared data space (cf. line 17 in Fig. 4.13). It thus relieves resource-restricted sensor nodes from storing microphone samples by putting them in its own memory.

Besides the *Smoblet* class and its subclasses, our programming framework also provides a range of other classes that support application programmers in implementing Smoblets.

4.5.3 The Smoblet Front- and Backend

The Smoblet frontend (cf. Fig. 4.14) is a graphical user interface that is executed on a user's handheld device; it can be used to search for Smoblets provided by nearby smart objects, for adapting security parameters, for retrieving information about downloaded code and its execution state, and for manually downloading as well as starting Smoblets. However, if the user permits it, a handheld device can also serve as an execution platform for nearby smart objects without requiring manual interaction. In this case, the mobile user device continually searches for Smoblets on nearby smart objects, and downloads as well as executes Smoblets whose informational parameters indicate that they can be automatically started.

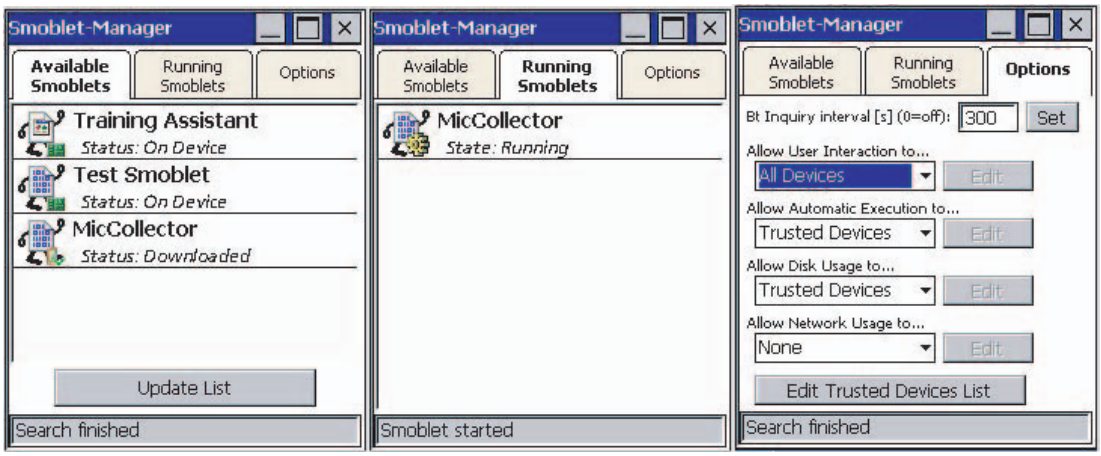

Figure 4.14: The Smoblet Manager: the tool for searching Smoblets and restricting access to a handheld's resources.

Especially for Smoblets that offer a graphical user interface and therefore provide interactive services, the time needed to discover nearby devices can significantly reduce usability. This is especially true for Bluetooth because of its relatively poor device discovery performance (a Bluetooth inquiry can take more than 10 s) [SR03]. In the following,

we therefore shortly discuss how the Smoblet frontend can speed up the process of finding new Smoblets by providing an alternative approach to selecting smart objects for interaction. As a possible solution, we propose an explicit selection mechanism based on passive RFID tags to determine the device address of a smart item. In this approach, a passive tag is attached to a smart object containing the device address of the BTnode integrated into that object. Also, a small-range RFID reader is attached to the mobile user device. A user can then explicitly select an object by holding the RFID reader close to it, thereby retrieving the corresponding BTnode's device address. Having this information, Smoblets from this object can be immediately downloaded to the handheld device and their graphical user interface can be started. To experiment with such explicit selection mechanisms for triggering interactions with smart objects, we connected a small-size (8cm × 8cm) RFID reader over serial line to a PDA. PDAs served as handheld devices in our implementation of the Smoblet framework.

In contrast to the frontend, the Smoblet backend is responsible for providing the actual Smoblet runtime environment and for accessing the distributed data structure shared by cooperating objects. It also handles the download of code, protects the handheld device from malicious code fragments, regularly searches for new devices in range, and forwards events to Smoblets while they are being executed.

4.5.4 Evaluation

In this section we evaluate our prototype implementation together with the underlying concepts. In particular, we discuss time constraints for downloading code, demands on the underlying communication technology, and the performance overhead caused by cooperating with multiple smart objects.

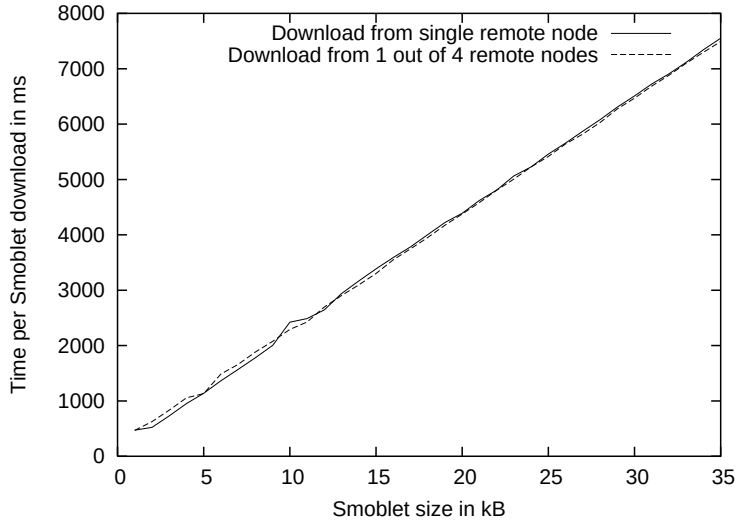

Figure 4.15: Time needed for downloading Smoblets from smart everyday objects.

Fig. 4.15 shows the time needed for downloading code in the realized prototype. It can be seen that the throughput achieved is about 37.4 kbit per second, which is relatively poor compared to the theoretical data rate of Bluetooth but compares well to data rates measured by other researchers in connection with Bluetooth-enabled embedded device platforms [LDB03]. However, Bluetooth itself is not the limiting factor when downloading code from a smart object, but thread synchronization issues on the mobile user device.

As code is usually downloaded in the background, several threads are executed simultaneously during the download: Java threads, threads belonging to the Bluetooth stack, and threads that are used to continuously query for devices in range and for accessing the shared data structure. The thesis that Bluetooth is not the dominating factor is also supported by the experimental results depicted in Fig. 4.15 because the time for downloads does not depend on the number of nodes that share a Bluetooth channel. Because of the TDD (time division duplex) scheme in which Bluetooth schedules transmissions and the fact that the Bluetooth modules used seem to apply a simple round-robin mechanism for polling nodes, an increased number of devices sharing a channel would imply decreased performance, which cannot be observed in our case. However, besides the relatively low throughput, even Smoblets that offer graphical user interfaces are typically downloaded in a few seconds. This is because class files and pictures are compressed before they are stored on a smart object. Typical compression rates for code are around 40-50 % but less for pictures because they are usually already in a compact format. For example, the interactive Smoblet presented in Sect. 4.6.6 has a code size of around 26.9 kB and a size of 15.2 kB after compression. Therefore it takes only about 3.5 s to download the code from a smart object to a handheld.

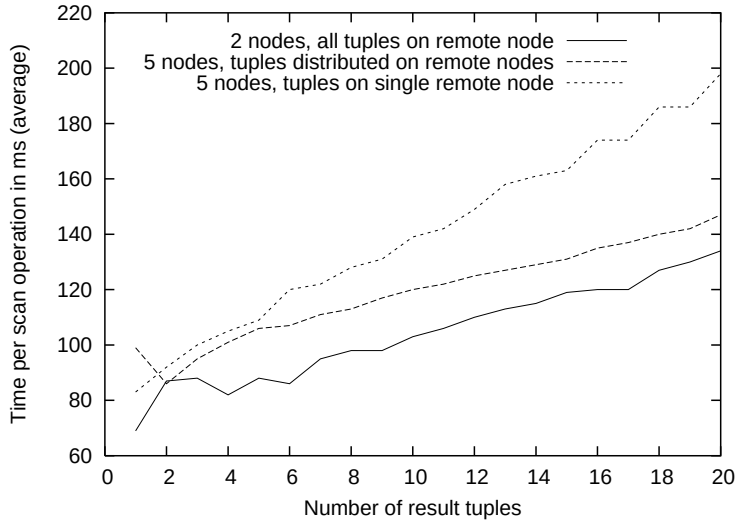

Figure 4.16: Time needed for a *scan* operation on the shared data structure.

As previously discussed, a core concept for integrating handhelds into sets of cooperating smart objects is to make the location where data are stored transparent for mobile code. Hence, as long as Smoblets search and exchange data by means of the distributed tuplespace, they can be executed on every node participating in that data structure without change. For example, in Fig. 4.13 line 17, the MicCollector Smoblet scans for all microphone samples in the distributed data structure and thereby relieves *all* collaborating nodes from storing that data in their local tuplespaces. The same program could be theoretically executed on every device participating in the data structure, always having the same effect. Fig. 4.16 shows the time needed for a *scan* operation on the distributed tuplespace with respect to the number of tuples returned and the number of cooperating smart objects. In the underlying experiment, all smart objects are members of a single piconet; the *scan* operation returns all tuples in the distributed tuplespace matching a given template. As can be seen, the performance for retrieving data depends on the distribution of tuples on the remote devices. This is also a consequence of Bluetooth's

TDD scheme for scheduling transmissions.

We would like to conclude this evaluation with a discussion about the overhead caused by cooperating with collections of remote smart objects. In our approach only code and no data are shipped from a smart object to a mobile device during migration. This is because Smoblets usually operate not only on data from the smart object that provided the code, but on data from multiple cooperating devices. As can be seen in Fig. 4.16, tuplespace operations on multiple remote nodes are thereby almost as efficient as operations on a single device but offer the advantage of operating simultaneously on many objects.

4.6 Applications

The thesis formulated at the beginning of this chapter was that smart objects can realize increasingly sophisticated services if they are able to cooperate with nearby handheld devices in an ad-hoc fashion. In this section, we support this thesis by presenting several application scenarios and corresponding prototypical implementations. Their purpose is to illustrate the previously identified roles of handhelds in smart environments, and to evaluate their applicability in concrete example scenarios. In these scenarios, handhelds often assume different roles in different subparts of an application, which allows us to illustrate how the isolated roles identified previously can be combined in order to develop their full potential. Our goal is to show what kinds of applications become technically feasible in cooperative smart environments; we explicitly do not focus on user studies, but on the technical and infrastructural foundations for deploying handhelds in Pervasive Computing settings.

4.6.1 Smart Object—Human Interaction: as Easy as Making Phone Calls

The application scenario presented in this section demonstrates how mobile user devices can serve as *mobile storage medium*, *weak user identifier*, and as *user interface*. Its purpose is to show how users can interact with remote objects by means of their handheld devices.

People usually associate a specific type of handheld device with a specific way to interact with communication partners: teenagers write SMS messages to arrange a fun meeting using their mobile phones, and business people organize their appointments with PDAs. Adopting device specific behavior to smart environments while maintaining interaction patterns people expect from their handheld devices is a key approach for successfully integrating handhelds into smart environments. In the following, we illustrate how device specific interaction patterns like making phone calls can be used as a metaphor for implementing remote interactions with smart objects.

Enabling remote interactions is a two step process: (1) when a user is in the proximity of a smart object, it stores interaction stubs in the user's handheld device; (2) later, when not in the vicinity of the object, a user selects a suitable interaction stub stored on the handheld to trigger a remote interaction with an augmented item (cf. Fig. 4.17). The interaction stubs are the key mechanism to establish the remote communication link. They consist of a human readable name for a smart object, a set of commands that can be executed by it, and its address. In our current implementation, mobile phones serve as handhelds and the actual communication with a smart object takes place by exchanging

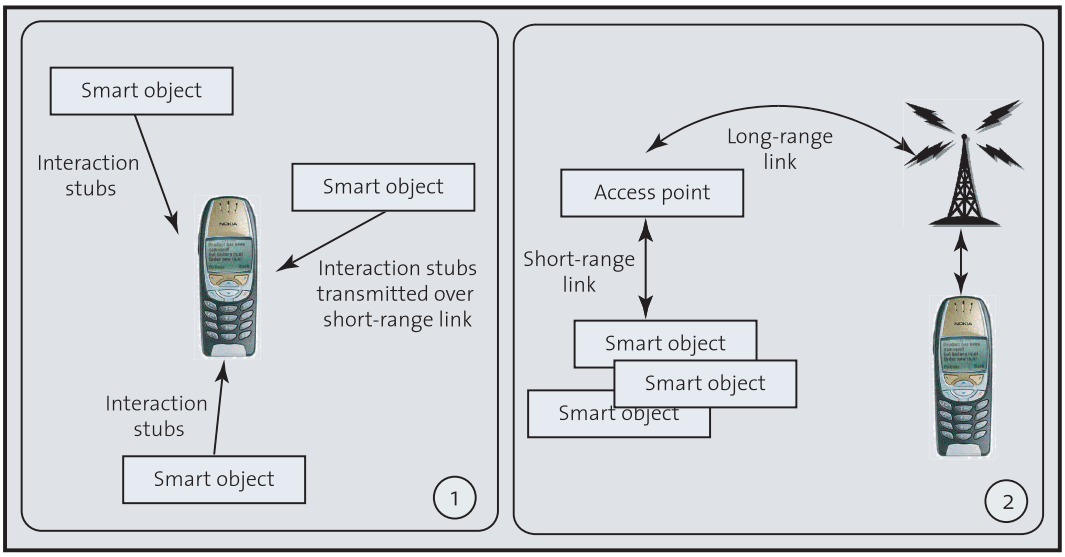

Figure 4.17: Remote interaction with smart objects using handheld devices: when people are in range of smart objects, handhelds serve as a *mobile storage medium* for interaction stubs (1); when far away, interaction stubs are executed to trigger interactions with remote objects using the handheld as a *user interface* (2).

SMS messages. Here, an interaction stub is composed of a phone book entry for the smart object and an SMS template. The phone book entry indicates the human readable name and the object's address, which is a telephone number in this case. The SMS template contains a range of predefined commands that can be activated and sent to the smart object (cf. Fig. 4.18).

We illustrate our approach with an office room as an example of a (rather large) smart object. The "smart office" knows who is currently working in it and can determine the current noise level inside it. A BTnode equipped with several sensors, such as a microphone, is placed in the office and provides information about the noise level inside (cf. [BG03] for a description of the sensor boards used). Furthermore, according to the concept of weak identification we can infer from a handheld's presence who is currently

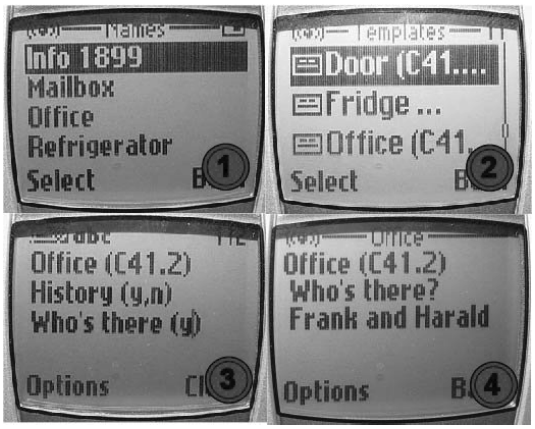

Figure 4.18: Interaction stubs transmitted from smart objects to a mobile phone (phone book entries (1), SMS templates (2)), an edited SMS template with activated command (3), and the corresponding reply from a remote smart object (4).

in the room, and utilize the handheld as a *weak user identifier*. Given these capabilities, the smart office can keep track of entering and leaving persons, maintain a short history of events, and could also derive its current situational context (e.g., an ongoing meeting). Based on this information, interaction stubs (phone book entries and SMS templates) are transmitted to a user's handheld device. For example, the person most frequently in the office is given a special stub that allows her to remotely interact with the office after hours.

Interaction stubs (phone book entries and SMS templates) are transferred over a short-range wireless link between smart object and handheld (cf. Sect. 4.4.2). These transmissions are completely transparent to the user and are initiated by a smart object based on its current context and history information. Later, when people want to remotely interact with the smart office, they select the corresponding phone book entry from their phone book and compose a new SMS message using the appropriate SMS template. The SMS message is received by a stationary access point with a GSM gateway and relayed to the corresponding smart object in range of the access point. The smart object then executes the embedded commands and returns a message to the user's mobile phone (cf. Fig. 4.18). Consecutive messages can be exchanged between user and smart object.

Direct remote interaction with a smart object requires a nearby stationary access point. The next section shows how we can get rid of this stationary gateway by using nearby handheld devices as *mobile infrastructure access points*.

4.6.2 The Smart Medicine Cabinet

The smart medicine cabinet illustrates a handheld's role as *mobile infrastructure access point*, *mobile storage medium*, *weak user identifier*, and *user interface*. It was designed to improve the drug compliance of patients with chronic diseases by reminding them to take their medicine. It also knows about its contents and can be queried remotely with a WAP-enabled mobile phone. Interaction with the information technology inside the cabinet is implicit – i.e., transparent for the patients – who might not even know that the cabinet is "smart."

Several technologies are necessary to implement the interaction with a patient, and to augment the medicine inside the cabinet and the cabinet itself: (1) passive RFID tags are attached to the folding boxes that contain the actual medicine, (2) an RFID reader is mounted to the back of an ordinary medicine cabinet, and (3) a BTnode processes the information from the RFID reader and communicates via Bluetooth with (4) the patient's mobile phone (cf. Fig. 4.19).

By using small RFID tags attached to the folding boxes and an off-the-shelf medicine cabinet equipped with a BTnode connected to an RFID reader, the information technology becomes completely invisible to the users (cf. Fig. 4.19). When a patient removes a certain kind of medicine she needs to take during the day, the active tag in the cabinet establishes a connection through the user's Bluetooth-enabled mobile phone to a background infrastructure service, querying it about prescription information concerning this medicine. In this situation, the mobile phone serves as a *mobile infrastructure access point* for the BTnode within the cabinet. As the backend service identifies patients based on their phone numbers, the mobile phone also serves as a *weak user identifier*. Subsequently, the active tag utilizes the user's mobile phone as a *mobile storage medium* and stores an alarm on it according to the prescription information previously obtained from

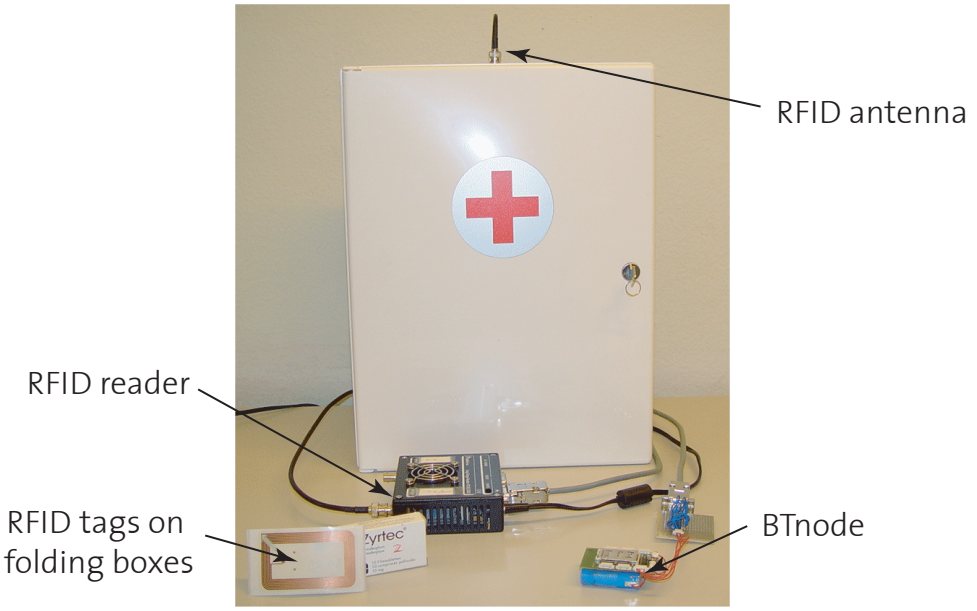

Figure 4.19: The smart medicine cabinet.

the background infrastructure service. Hence, at the time scheduled, the mobile phone issues an alarm that reminds the patient to take the medicine.

A WAP-based interface allows patients to remotely interact with the smart medicine cabinet – or its representation in the background infrastructure, the background infrastructure representative (BIRT) of the cabinet. Since the patient's mobile phone serves as a *mobile infrastructure access point* for the cabinet, it operates in a disconnected mode whenever there is no mobile phone present. It hence requires a BIRT in the background infrastructure that represents the medicine cabinet continuously. Whenever a mobile phone is in the vicinity of the medicine cabinet and provides connectivity, the cabinet synchronizes its state with the BIRT. Afterwards, remote WAP-based queries need to address the BIRT of the cabinet. Information displayed on WAP pages regarding the contents of the cabinet and recently taken medicine therefore reflect the status of the cabinet during the last synchronization (cf. Fig. 4.20).

An actual use case would be the following: A patient approaches the smart medicine cabinet and takes a package out of the cabinet. The active tag (i.e., the BTnode) in the cabinet notices that a box of medicine has disappeared and connects to a background service (the BIRT of the cabinet). Here, the patient's mobile phone acts as a *mobile infrastructure access point* to the cellular phone network for the smart object, and no stationary access point is required near the cabinet. The Bluetooth-enabled active tag in the cabinet queries the BIRT about when the patient has to take the medicine. It then stores a corresponding alarm in her mobile phone that reminds the patient to take the medicine during the day. While there is a connection to the background infrastructure through the patient's mobile phone, the cabinet also synchronizes its state with that of the BIRT, which provides WAP pages based on this information. When the patient visits a pharmacist, the WAP interface can be used to query the contents of the cabinet (cf. Fig. 4.21). This information is a good basis for the pharmacist to decide whether another kind of medicine is compatible to that in the smart cabinet.

The concept of a medicine cabinet augmented by information technology has been demonstrated previously by Wan [Wan99], who integrated a personal computer, an LCD

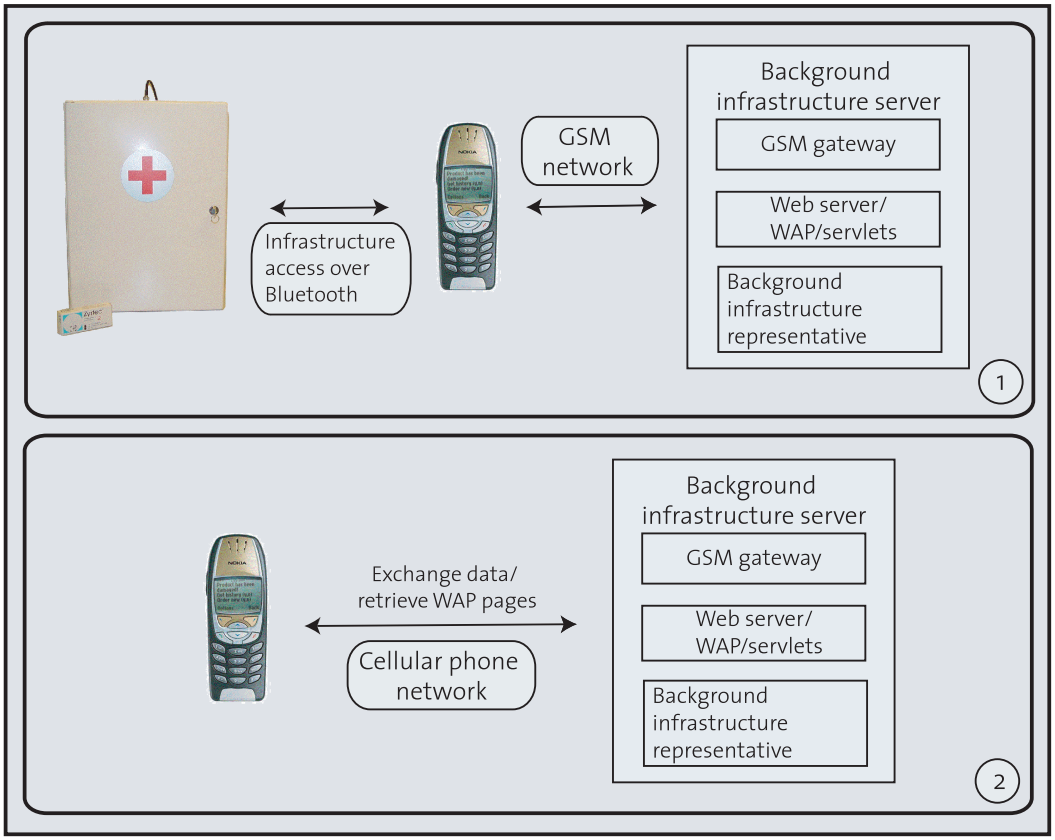

Figure 4.20: When the patient is in range of the cabinet her mobile phone serves as a *mobile access point* to the background infrastructure server (1), when not in range of the cabinet she interacts with the background infrastructure representative of the cabinet using the handheld as a *user interface* (2).

screen and a broadband Internet connection into a medicine cabinet. The medicine cabinet presented as part of this work, however, was designed with the goal of leaving the medicine cabinet practically unmodified from a user perspective.

4.6.3 The Smart Blister Pack

The smart blister pack [FS03] is an application developed by Christian Floerkemeier, and makes use of our software package for interfacing handheld devices from smart objects. It is described here as an example of how other researchers can apply the described usage patterns for handhelds in their work. The smart blister pack also illustrates a handheld's role as *mobile infrastructure access point* and *user interface*.

As in the smart medicine cabinet scenario, the main goal of the smart blister pack is to improve the drug compliance of patients. It monitors how patients take their medicine and compares their actual drug intake with a prescribed treatment. The main application area for which the smart blister pack was designed is clinical trials, where it is very important for pharmaceutical companies to ensure that patients take a new drug according to a scheduled treatment.

In order to monitor the drug intake of patients, an ordinary blister pack is augmented with small metal stripes that are destroyed when a pill is removed from the blister. When a patient needs to take a pill, she attaches the augmented blister pack to a small

Figure 4.21: The WAP interface to the smart medicine cabinet.

casing containing a BTnode (cf. Fig. 4.22). Here, the augmented blister pack serves as a sensor for the BTnode, which notices when a pill is removed. When such a pill removal event occurs, an SMS message is automatically sent to a backend infrastructure service informing the pharmaceutical company that the pill has been taken. The patient's mobile phone serves as *mobile infrastructure access point* for the smart blister pack. The advantage of this approach is that it does not require any manual input from the patient. Instead, the connection to the phone is established automatically and people do not have to input further data. Should a patient forget to take her medication, the backend service sends an SMS message as a reminder to her mobile phone. The handheld thereby serves as a *user interface*.

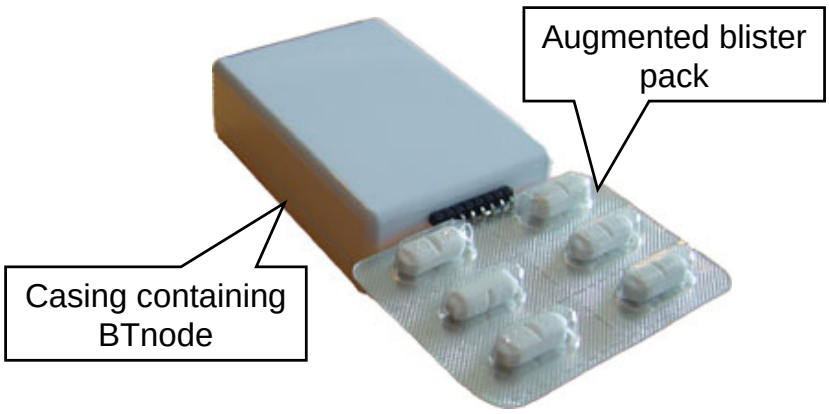

Figure 4.22: The smart blister pack.

4.6.4 Object Tracking and Monitoring

The application presented in this section illustrates a handheld's role as *remote sensor*, *mobile infrastructure access point*, and *user interface*. Its goal is to monitor the state of goods during transport and storage in such a way that the owner of a product can query the product's state and is notified whenever it is damaged. The difference to other approaches to product monitoring (e.g. [Par04]) is that goods are monitored continuously without relying on extra stationary installed hardware. Instead, smart products use mobile user devices (e.g., the mobile phone of a driver) to communicate with backend services, to obtain location information, and to notify remote users.

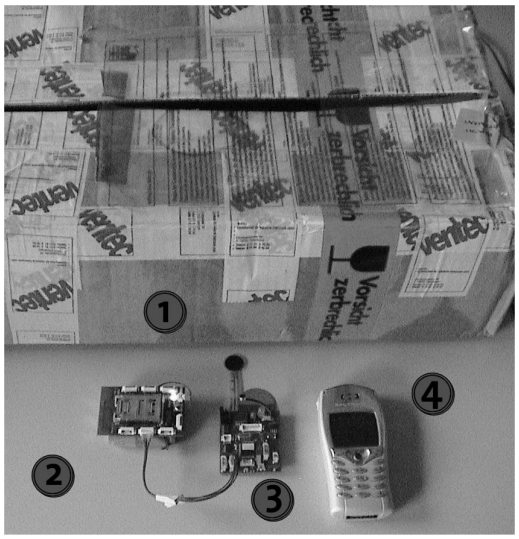

Figure 4.23: The main components of the monitoring application: (1) a smart product, (2) a BTnode with an attached (3) sensor board, and a (4) mobile phone.

A product is augmented with an active sensor-based tag that monitors its state continuously (cf. Fig. 4.23) – in our prototypical implementation, BTnodes connected to sensor boards serve as tags (see Beigl et al. [BZK+03] for a detailed description of the sensor boards used). The tags are equipped with acceleration sensors to derive the product's current status and to determine the reason why a product has been damaged. When a product is destroyed, the attached BTnode tries to notify the owner of the product by sending an SMS message through a nearby access point. In our exploratory implementation, background infrastructure access takes place using nearby Bluetooth-enabled mobile phones as access points. In reality, this could for example be a handheld device carried by the driver of a truck. When a handheld is in range of augmented products, they use it as *mobile access point* for communicating with a background infrastructure service, the background infrastructure representative (BIRT) of the product. The BIRT can be informed if the product's status changes or has changed while no infrastructure access has been possible.

Additionally, the mobile phone is used as a *remote location sensor*. When a product is damaged, the corresponding BTnode obtains the current cell id and location area code from the mobile phone (cf. Sect. 4.4.3). The information about where an event occurred (i.e., where a product is damaged or destroyed) is then embedded into a message sent to the BIRT of the product.

The user interaction consists of a WAP interface for querying the status of a product.

Figure 4.24: The WAP interface for checking the status and location of products.

The WAP interface is provided by the BIRT of a product, which has again been implemented as a Servlet. The BIRT generates WAP pages containing information about the product's current state based on the messages transmitted by the BTnode attached to the product (cf. Figure 4.24).

4.6.5 Collaborative Context Recognition and Streaming Data

The applications presented in this and the following section deal with a handheld's role as *remote resource provider*, and are used to illustrate the applicability of the Smoblet framework (cf. Sect. 4.5). There are two major application domains for Smoblets: (1) exploiting the computational resources of handheld devices in order to facilitate collaborative context recognition in smart environments and (2) enabling graphical user interaction with smart objects by outsourcing user interfaces to nearby handhelds. In this section, we present an example of the first of these two application areas, and show how handhelds can help collections of smart everyday objects to collaboratively derive context information.

In order to provide context-aware services, smart objects must be able to determine their own situational context and that of nearby people. This usually requires cooperation with other objects and the ability to process local sensor readings together with sensory data provided by remote nodes. A significant problem that often arises in these settings are streaming data, e.g. from microphones and accelerometers, which are difficult to exchange between and difficult to store on smart objects because of their resource restrictions. For example, it is nearly impossible for smart objects to store microphone data sampled over longer periods as a result of their limited data storage capacities. This is because of the high microphone sampling rates (often above 5 kHz), which are necessary to extract context information from a microphone data stream. In the case of accelerometers, Lester et al. [LHB04] reports that a sampling rate of 20 Hz is sufficient to extract features relevant to human activity. However, considering that an accelerometer sample for one dimension on the adxl202 accelerometer used on the TecO sensor boards [BZK+03] is 2 Bytes long, it becomes clear that a node such as the BTnode with 64kBytes of RAM (much of which is already occupied) cannot store accelerometer samples for long time periods even at sample rates of 20 Hz.

Basically, there are three options for dealing with large amounts of sensory data: (1) process streaming data on the fly, i.e. compute context information from sensory data without storing them, (2) extract characteristic features from a data stream, which are smaller in size but still carry the relevant information required in the context recognition process, and (3) transmit sensory data to other devices with more resources. Depending

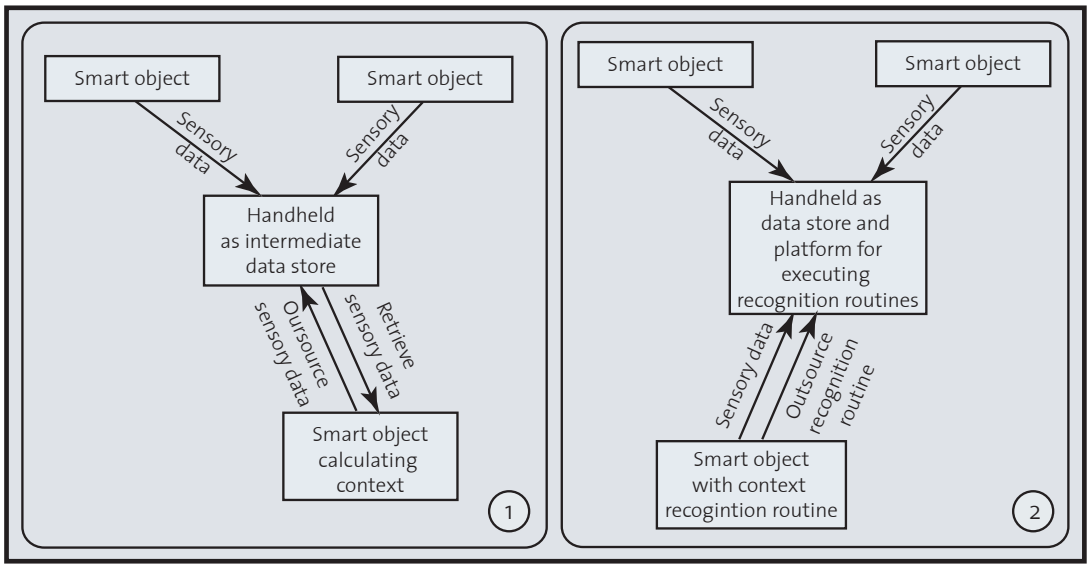

Figure 4.25: Collaborative context recognition with handheld devices: (1) if handhelds serve as intermediate data store for sensory data, the smart object calculating context information transmits sensory data to the handheld and must afterwards retrieve them in smaller parts for processing; (2) if the handheld both stores sensory data and processes them locally, the object interested in a certain context does not have to retrieve sensory data from the handheld, which can significantly decrease the communication overhead.

on the context processing routine, option (1) is not always feasible. Option (2) has the disadvantage that even extracted features might consume too much memory if they need to be stored for long time periods. The drawback of option (3) is that it requires constant communication, and can therefore consume significant amounts of energy. Although collaborative context recognition is always based on communication, in option (3) sensory data must be transmitted twice. First, the sensory data are transmitted for intermediate storage to the more powerful computer. But since the context recognition routine is still carried out by a smart object, the relevant data must again be retrieved from the intermediate data storage in a second step. Both steps involve communication, which is the dominating factor in a smart object's overall energy consumption. A possible solution to this problem is to outsource the context recognition routine together with the sensory data to a nearby handheld device. As the handheld then carries out the context recognition itself, fewer sensory data have to be transmitted between cooperating nodes (cf. Fig. 4.25). This effect becomes even more significant with an increasing number of cooperating devices because more devices generally generate more sensory data that must be stored on an intermediate storage medium. The cost for outsourcing the context recognition routine to a handheld device (which itself requires communication) is in such cases likely to be lower than the cost for repeatedly transmitting sensory data.

Based on the Smoblet framework presented in section 4.5, we now show how nearby handheld devices can help smart objects in handling large amounts of sensory data during the collaborative context recognition process. As an example, we have implemented a Smoblet for retrieving and processing microphone data samples from different smart objects. The Smoblet evaluates which smart objects are in the same room and can be used for finding out what is happening at a specific location. The data for the context recognition process are retrieved from low-cost microphones attached to smart objects

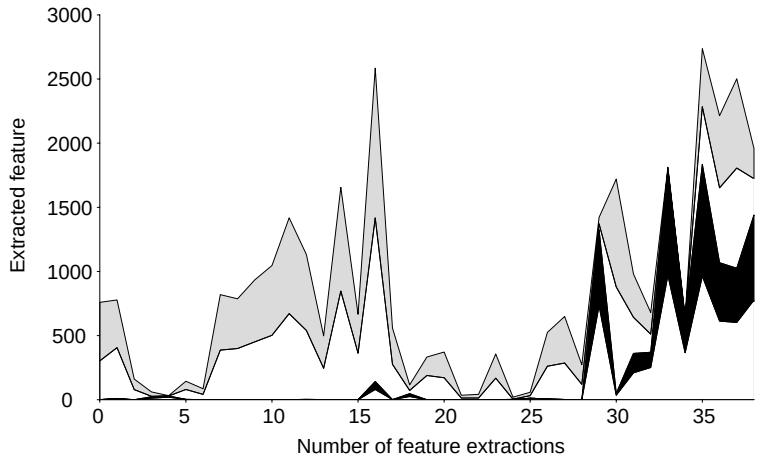

Figure 4.26: Small subset of the data assembled by the MicCollector Smoblet; the depicted data are the microphone measurements of four remote smart objects retrieved by the handheld carrying out the MicCollector Smoblet.

(for our experiments we have used the sensor boards described in Beigl and Gellersen [BG03]).

Usually, a smart object that is interested in context information indicating what objects are in the same room would have to collect microphone data from nearby augmented items and process these data itself. In our case, we assume that there is currently a handheld device in range of the smart object (for example, a person could be working nearby who carries a handheld) and that the microphones generate too many sensor samples for the smart object to store locally. In order to cope with the latter problem, the smart object automatically transmits a MicCollector Smoblet (cf. Fig. 4.13) to the nearby handheld device, where the Smoblet is automatically started. In our implementation, smart objects continuously sample their microphones at approximately 40 kHz and extract a feature from these sensor readings indicating the level of activity in their room. In this concrete example, we sample microphones continuously for approximately 500 ms and use the number of crossings through the average microphone sample as a feature. The MicCollector Smoblet that has been outsourced from one of the smart objects to the handheld retrieves these features from *all* augmented items in range, thereby relieving them from storing these data. After several readings, the MicCollector Smoblet can derive the location of smart objects (i.e., whether they are in the same room) and determine what is happening in a room. Because of the more sophisticated computational capabilities of handhelds, the Smoblet can thereby carry out more demanding algorithms for evaluating sensory data. Furthermore, the Smoblet relieves smart objects from storing sensory data, and as sensor features are processed on the handheld (i.e., at the same location at which the sensor features are stored), they do not have to be retransmitted between smart objects and mobile user device.

Fig. 4.26 shows the microphone features collected by the MicCollector Smoblet from four remote smart objects. In the figure we have filled the area between sensor features from smart objects in the same room. As can be seen, the features of augmented items in the same room are correlated, meaning that they decrease and increase simultaneously. This fact is exploited during the context recognition process. This context recognition process is encapsulated in a Smoblet which has been outsourced from one of the smart objects to a handheld device.

4.6.6 Training Assistant – Outsourcing Java User Interfaces to Handheld Devices

Besides supporting smart objects during the context recognition process, the Smoblet concept (cf. Sect. 4.5) also allows augmented items to provide complex graphical user interfaces to nearby people. As a proof of concept, we present in the following an application that helps sportsmen in evaluating their training: a Smoblet encapsulates a graphical interface that is transmitted from a smart object monitoring training progress to a user's handheld device. The Training Assistant application illustrates a handheld's role as *remote resource provider* and *user interface*.

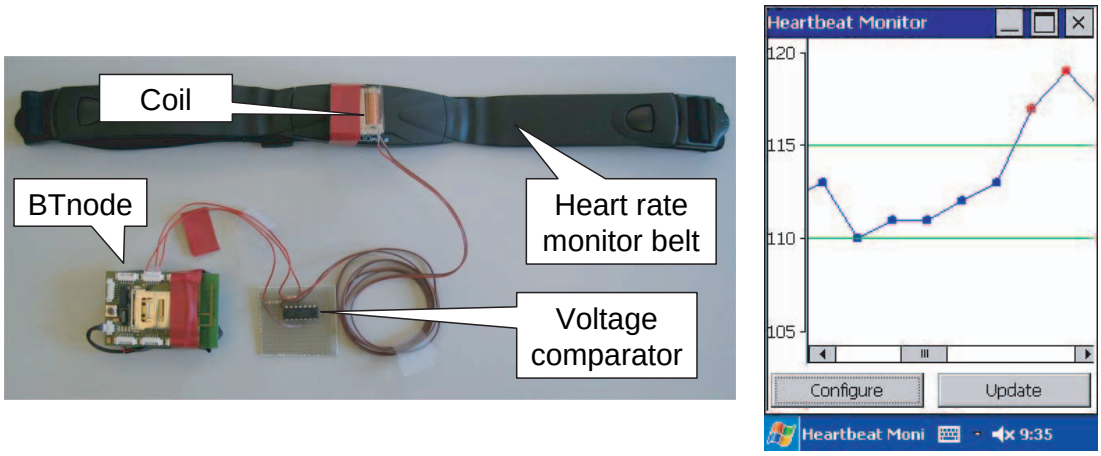

Figure 4.27: (a) A BTnode with an attached heart rate sensor; (b) the user interface downloaded from the BTnode to visualize the pulse data collected during the last training.

In our prototypical implementation, we have augmented an ordinary heart rate monitor belt with a BTnode that records the electromagnetic pulses generated by the belt during a training session (cf. Fig. 4.27). Technically the belt serves as a sensor for the BTnode, which can therefore access pulse measurements and exchange them wirelessly with other computing devices in range. During an ongoing training session, the BTnode buffers pulse measurements. When an athlete wants to evaluate the data recorded during a training session, she uses the Smoblet frontend to search for Smoblets provided by nearby smart objects. As a result, a list containing all Smoblets found is displayed on her handheld. The user can then select as well as start the Training Assistant Smoblet, whereupon a Java graphical user interface is downloaded from the augmented heart rate monitor belt and displayed on her PDA (cf. Fig. 4.27). Access to the heart rate data is possible by means of the distributed tuplespace that is established by cooperating devices. The Smoblet that is executed on the user's handheld can therefore retrieve the heart rate data and display them on the mobile user device.

In a more sophisticated version of the Training Assistant (which we do not describe here in detail), it is also possible to configure the behavior of the augmented heart rate monitor belt. Here, another BTnode with a small loudspeaker is used to notify the athlete during a training session when her pulse exceeds previously set thresholds. The user interface provided by the Smoblet can be used to set the corresponding thresholds. User input is embedded into tuples, transferred to the smart objects, and stored in their local tuplespaces so that the handheld does not have to be carried during a training session.

4.7 Summary

With respect to the overall argumentation of this dissertation, this chapter showed how smart objects can benefit from the heterogeneity of smart environments by accessing the capabilities of other kinds of computing devices. The chapter focused on the interaction between smart objects and handhelds, and formulated as well as supported the hypothesis that smart objects can provide increasingly sophisticated services to people in smart environments if they are able to exploit the capabilities of nearby handheld devices in an ad hoc fashion.

First, this chapter argued that handheld devices are well suited for cooperation with smart objects because of their complementary capabilities regarding user interfaces, processing power, and their ability to sense the environment. In specific terms, the following reasons were identified to explain why handheld devices are such valuable cooperation partners: (1) their habitual presence during physical interactions with smart objects, (2) their wireless network diversity, (3) their user interface capabilities, (4) their ability to perceive their environment with complex sensors, (5) their high availability and accessibility for users, (6) their computational resources and regularly refilled mobile energy reservoirs, and (7) the fact that they are most often personalized.

Table 4.4: The roles of handhelds in the applications presented in this chapter.

Application	Handheld's role
Remote interaction	Mobile storage medium, user interface, weak user identifier
Smart medicine cabinet	Mobile infrastructure access point, mobile storage medium, user interface, weak user identifier
Smart blister pack	Mobile infrastructure access point, user interface
Object tracking	Mobile infrastructure access point, remote sensor, user interface
Collaborative context recognition	Remote resource provider
Training assistant	Remote resource provider, user interface

Second, we presented several usage patterns describing how smart objects can make use of handheld devices: (1) as a mobile infrastructure access point, (2) as a user interface, (3) as a remote sensor, (4) as a mobile storage medium, (5) as a remote resource provider, and (6) as a weak user identifier. We gave a detailed presentation of these usage patterns, containing descriptions of the underlying concepts as well as directions for their implementation. The BTnodes as examples of active sensor-based tags, together with mobile phones and PDAs, were used to present prototypical implementations of the identified usage patterns.

Third, the chapter dealt with the problem of integrating handhelds into collections of cooperating smart everyday objects. Here, the proposed solution was to establish a shared data structure among cooperating objects and handheld devices, and to outsource code for complex and energy-consuming computations from smart objects to handhelds. In order to evaluate the applicability of the proposed concepts, we again presented a prototypical implementation of these concepts on an embedded sensor-node platform, the BTnodes.

Fourth, we illustrated the applicability of the presented concepts with six applications: (1) a remote interaction scenario, (2) a smart medicine cabinet, (3) a smart blister pack, (4) an object tracking application, (5) an application for collaboratively deriving context information, and (6) a training assistant. These applications showed how the individual usage patterns identified previously can be combined in concrete application scenarios in order to develop their full potential. Table 4.4 summarizes the roles of handhelds in the presented applications.

Chapter 5

Conclusions and Future Work

In this chapter, we conclude this dissertation by (1) summarizing its main contributions and (2) suggesting directions for future work.

We would also like to point out that extracts of this thesis have been published in [BKM+04, BKR+03, FS03, SF03a, SFV04, Sie02a, Sie02b, Sie04, SK04a, SK04b, SR03, SSB+03]. Several student projects and master's theses, which were supervised within the scope of this dissertation, also dealt with concepts presented in the previous chapters. Of especial relevance to the interested reader might be [Dea03, Kel03, Kel04, Kra04, Sch03].

5.1 Main Contribution

Pervasive Computing envisions the seamless integration of computation into everyday environments in order to support people in their daily activities. Due to recent developments, such a level of integration of technology into our daily surroundings has become feasible at least from a technological perspective. Sensor-based computing platforms, for example, have become so small that it is possible to embed them into arbitrary everyday things. Such computationally-augmented things – so-called smart everyday objects – have the ability to perceive their environment through sensors and can communicate with peers by means of short-range communication technologies.

However, smart everyday objects are usually faced with severe resource restrictions and can therefore only provide limited services to people in smart environments. As the computing platforms of smart objects need to be small in order to be unobtrusively integrated into everyday things, they do not usually possess conventional user interfaces. Furthermore, smart objects often provide context-aware services that adapt application behavior according to the current situation of nearby people. Sensor readings from a single object alone are thereby usually not sufficient to obtain enough information about a user's context. Together, all these constraints considerably restrict a smart object's ability to realize services in smart environments.

With respect to these problems, the core contribution of this dissertation is to show how smart objects can cope with the above-mentioned restrictions by cooperating with other computing devices in their vicinity. Cooperation with other devices enables augmented artifacts to exploit two key properties of smart environments: (1) pervasiveness of computation and (2) heterogeneity.

Chap. 3 illustrated how smart objects can benefit from the pervasiveness of computation in smart environments by cooperating with other augmented artifacts. In the presented approach, smart objects bundle their resources and appear to applications as

a single node accumulating the capabilities of cooperating artifacts. The chapter also described a software architecture for smart objects that aims at facilitating the design of collaborative context-aware services in smart environments.

Chap. 4 focused on the cooperation of smart objects with handheld devices. Its major goal was to show how augmented artifacts can benefit from the heterogeneity of smart environments by accessing the capabilities of other types of computing devices. The chapter presented several usage patterns that illustrated how smart objects can make use of nearby handhelds, described prototypical implementations of these usage patterns, and illustrated their applicability with several example applications.

5.2 Other Contributions

In the following, we list the main contributions of this dissertation individually:

- **Context-aware collaboration.** In Chap. 3, we have presented a software platform that enables smart objects to realize services in cooperation with other augmented artifacts. The software platform consists of (1) a description language for collaborative context-aware services – called SICL, (2) a context layer for deriving contextual information from multiple sensor sources, (3) an infrastructure layer for inter-object cooperation that allows smart objects to share their resources, and (4) context-aware communication services. We have described a prototypical implementation of this software platform on embedded sensor nodes, and thoroughly evaluated the presented concepts. The evaluation was based upon experiments with real sensor nodes as well as simulations.

- **Programming abstraction for collaborative services.** To overcome the resource restrictions and limited sensing capabilities of individual smart objects, we suggested grouping cooperating augmented artifacts. Nodes in a group bundle their resources and appear to an application as a single more powerful object accumulating the resources of the group's participants. As the distribution of data between nodes in a group is hidden in the lower layers of our software platform, an application programmer can disregard these issues and deal with the whole group as if it were a single node. In order to effectively realize this concept, we suggested clustering collaborating nodes on a distinct broadcast channel. As shown in Sect. 3.5.4, this results in an efficient mechanism for multicasting data among the participants of a group and therefore enables efficient cooperation.

- **Context-aware communication services.** Networks of cooperating smart objects are difficult to manage for several reasons (cf. Sect. 3.6). However, as communication is a precondition for cooperation, we asked whether there is a property shared by smart objects that could be used to improve wireless communications. In Sect. 3.6 we argued that context-awareness is such a property because in context-aware applications the real-world surroundings of smart objects have an increasing impact on their communication in the virtual world – i.e. their interaction with other computing devices. In this dissertation, we showed how context-aware communication services can be used to proactively construct networking structures according to the demands of context-aware applications. It has also been shown how the previously presented software platform for collaborative services can be used to realize context-aware communication services.

- **Accessing the capabilities of handheld devices from smart objects.** In Chap. 4 we presented several usage patterns that described how resource-restricted smart objects can exploit the capabilities of handheld devices. These usage patterns are: (1) mobile infrastructure access point, (2) user interface, (3) remote sensor, (4) mobile storage medium, (5) remote resource provider, and (6) weak user identifier.

- **Smoblets – integrating handheld devices into environments of cooperating smart objects.** The Smoblet system (cf. Sect. 4.5) allows smart objects to dynamically exploit the computational resources and user interfaces of handheld devices. In our concept, handhelds join a shared data space established between cooperating smart objects. Thus, the location where program code is executed becomes transparent for objects in the shared data space. As a result, augmented items can outsource complex and energy consuming computations to nearby handheld devices. We presented a prototypical implementation of the Smoblet system, and evaluated its applicability in two example scenarios (cf. Sect. 4.6.5 and Sect. 4.6.6).

- **Embedding smart objects into the everyday communication infrastructure.** Smart objects not only need to cooperate with nearby computing devices, but also with services in a backend infrastructure. Furthermore, remote users and smart objects that are not in the local environment of an augmented artifact can also only be reached by means of a powerful communication infrastructure. To enable smart objects to interact with remote users and background infrastructure services, we described how augmented items can be embedded in and make use of the everyday communication infrastructure. In Chap. 4 we described how the Internet and the mobile phone network, as two examples, can be used for this purpose.

- **Translating established interaction patterns to smart everyday objects.** As smart objects do usually not possess conventional user interfaces such as buttons or displays, the question arises how people can actually interact with them. With respect to this problem, we suggested adapting well-established interaction patters – like making phone calls or writing SMS messages – to smart environments. Established interaction patterns have the advantage that they are accepted by many users and can therefore make it easier for people to deal with smart objects. In our approach, phone numbers are assigned to smart everyday objects in order to enable users to "call" their items or to communicate with them by means of SMS messages (cf. Sect. 4.4.2, Sect. 4.6.1, and Sect. 4.6.2).

- **Applications.** In the scope of this thesis, several application scenarios have been prototypically implemented that illustrate the applicability as well as the limits of our concepts. On the basis of concrete examples, we wanted to show how a smart object can realize novel kinds of services if it is able to cooperate with other computing devices. The presented applications (cf. Sect. 4.6) include a smart training assistant, a smart medicine cabinet, an object tracking, and an inventory monitoring application.

5.3 Future Work

This thesis focused on the infrastructural foundations for enabling smart everyday objects to cooperate with other computing devices present in smart environments. The general topic of cooperation in smart environments, however, is so extensive that we could only address a small subset of the problems associated with it. In the following, we identify some further research problems in this domain and present directions for future work.

In the scope of this dissertation, the BTnode embedded device platform has been used to build prototypical implementations. Although the BTnodes are suitable as a prototyping platform, they have the disadvantage that Bluetooth is given as a predefined communication standard. (This will change in new revisions of the BTnode platform, which will be equipped with two communication modules. The BTnodes will then support an additional communication technology besides Bluetooth.) The major problem with Bluetooth for research is that we did not have access to lower communication layers, which makes it practically impossible to tweak parameter settings on this level. It would be interesting to implement and evaluate the presented protocols for inter-object cooperation on other multi-channel communication technologies. In this respect, it could also be investigated how a contention-based instead of a polling-based MAC layer influences the performance of protocols for cooperation.

In Chap. 3 we presented a software platform that enables smart objects to realize services in cooperation with other augmented artifacts. In Sect. 4.5 we then showed how handheld devices can participate in groups of cooperating objects. This allows augmented items to benefit from the computational capabilities of handhelds and to outsource complex computations. In the same way, also a stationary computer could join groups of smart objects and make its resources available to them. As our prototypical implementation runs on handhelds and an embedded sensor node platform, some software adaptations would be necessary to port our system to stationary computers. If there is a stationary computer in range of cooperating objects, the question arises of how smart objects could further benefit from its resources.

In Chap. 4 we have investigated different options for people to interact with smart objects. In our work, we primarily focused on explicit user interactions with augmented items by means of handheld devices. However, in environments populated by large amounts of smart objects, implicit forms of interactions become increasingly important. With respect to this thesis, the question arises to what extent the software platform presented for realizing collaborative context-aware services can be used to facilitate such implicit interactions in smart environments.

In this dissertation, we have argued that cooperation helps smart objects to gain access to the resources they need for realizing sophisticated services. To take this approach to a new level, it is also possible to argue that because of the abundance of computation in Pervasive Computing settings, cooperation enables smart objects to access even more resources than they actually require. This results in an interesting research question, namely how the abundance of computing resources can lead to the implementation of more reliable services. For example, data could be stored redundantly on collaborating objects in order to achieve a higher degree of reliability in case of data losses.

As we have already emphasized in our motivation (cf. Chap. 2), cooperating smart objects that exchange sensory data and context information about nearby persons can be a cause for increased privacy concerns. Ensuring personal privacy in smart environments is certainly a problem of major political and scientific interest. The constraints of the

investigated settings – i.e., the presence of many devices with severe resource restrictions that might not always be able to access backend services – make it even more difficult to find an appropriate solution. Although there are approaches to tackling the privacy problem – which are, for example, based upon privacy profiles and sensors that disguise the identity of people – privacy will remain an important topic for future research.

Bibliography

[ABC+03] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System Support For MultimodAl NeTworks of In-situ Sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA) 2003*, pages 50–59, San Diego, USA, September 2003.

[ACH+01] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a Sentient Computing System. *IEEE Computer*, 34(8):50–56, August 2001.

[Aut04] Auto-ID Labs. http://www.autoidlabs.org, November 2004.

[Bar04] Barcode 1. http://www.adams1.com/pub/russadam/barcode1.cgi, November 2004.

[BBC97] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware Applications: from the Laboratory to the Marketplace. *IEEE Personal Communications*, 4(5):58–64, October 1997.

[BCE+97] F. Bennett, D. Clarke, J. B. Evans, A. Hopper, A. Jones, and D. Leask. Piconet – Embedded Mobile Networking. *IEEE Personal Communications*, 4(5):8–15, 1997.

[BCSW98] S. Basagni, I. Chlamtac, V. R. Syrotiuk, and B. A. Woodward. A distance routing effect algorithm for mobility (DREAM). In *ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom)*, pages 76–84, Dallas, USA, 1998.

[Bei02] M. Beigl. Context Aware Computing. Handouts of a tutorial on context at the International Conference on Architecture of Computing Systems (ARCS 2002), April 2002.

[BG99] A. B. Brody and E. J. Gottsman. Pocket BargainFinder: A Handheld Device for Augmented Commerce. In *Proceedings 1st International Symposium on Handheld and Ubiquitous Computing (HUC '99)*, Karlsruhe, Germany, September 1999.

[BG03] M. Beigl and H. W. Gellersen. Smart-Its: An Embedded Platform for Smart Objects. In *Smart Objects Conference (SOC) 2003*, Grenoble, France, May 2003.

[BGS01] M. Beigl, H.W. Gellersen, and A. Schmidt. MediaCups: Experience with Design and Use of Computer-Augmented Everyday Artefacts. *Computer Networks, Special Issue on Pervasive Computing*, 25(4):401–409, March 2001.

[BHHZ03] E.-O. Blass, H.-J. Hof, B. Hurler, and M. Zitterbart. Erste Erfahrungen mit der Karsruher Sensornetz-Plattform. In *GI/ITG KuVS Fachgespräch*, 2003.

[BK01] J. Barton and T. Kindberg. The Challenges and Opportunities of Integrating the Physical World and Networked Systems. In *7th Annual Conference on Mobile Computing and Networking (MOBICOM 2001)*, Rome, Italy, July 2001. Callenge paper.

[BKM+04] J. Beutel, O. Kasten, F. Mattern, K. Roemer, F. Siegemund, and L. Thiele. Prototyping Sensor Network Applications with BTnodes. In *IEEE European Workshop on Wireless Sensor Networks (EWSN)*, LNCS, pages 323–338, Berlin, Germany, January 2004.

[BKR+03] J. Beutel, O. Kasten, M. Ringwald, F. Siegemund, and L. Thiele. Bluetooth Smart Nodes for Ad-hoc Networks. TIK Report 167, ETH Zurich, April 2003.

[BKZ+03] M. Beigl, A. Krohn, T. Zimmer, C. Decker, and P. Robinson. AwareCon: Situation Aware Context Communication. In *Fifth International Conference on Ubiquitous Computing (Ubicomp 2003)*, pages 132–139, Seattle, USA, October 2003.

[BL03] T. Berners-Lee. Semantic Web Tutorial Using N3. http://www.w3.org/2000/10/swap/doc/, 2003.

[Blu04] Blueware: Bluetooth Simulator for ns. http://nms.lcs.mit.edu/projects/blueware/software/, November 2004.

[BM98] R. Barrett and P. P. Maglio. Informative Things: How to Attach Information to the Real World. In *Proc. 11th Annual ACM Symposium on User Interface Software and Technology (UIST 98)*, pages 81–88, San Francisco, USA, November 1998.

[BMS97] H. W. P. Beadle, G. Q. Maguire, and M. T. Smith. Using location and environment awareness in mobile communications. In *Proceedings of the International Conference on Information, Communications and Signal Processing (ICICS 1997)*, volume 3, pages 1781–1785, Singapore, September 1997.

[BR01] J. Bohn and M. Rohs. Klicken in der realen Welt. In *Konferenz Mensch und Computer 2001, Workshop Mensch-Computer-Interaktion in allgegenwärtigen Informationssystemen*, Bad Honnef, Germany, March 2001.

[Bro96] P. J. Brown. The Stick-e Document: a Framework for Creating Context-aware Applications. In *Proceedings of EP'96, Palo Alto*, pages 259–272. also published in *EP–odd*, January 1996.

[Bro02] D. L. Brock. Smart Medicine, The Application of Auto-ID Technology to Healthcare. White paper, Auto-ID Center, February 2002.

[Btn04] BTnodes, revision 3. http://btnode.ethz.ch, November 2004.

[BZK+03] M. Beigl, T. Zimmer, A. Krohn, C. Decker, and P. Robinson. Smart-Its - Communication and Sensing Technology for UbiComp Environments. Technical report, TecO, Technical University of Karlsruhe, 2003. Technical Report ISSN 1432-7864 2003/2.

[Cap04] C. H. Cap. Identifikation mit klassischen Techniken. Lecture notes "Smart Cards, Smart Labels, Smart Devices", http://wwwiuk.informatik.uni-rostock.de/sites/lehre/lehrveranstaltungen/smartx_vorlesung.htm, 2004.

[CCRR02] J. L. Crowley, J. Coutaz, G. Rey, and P. Reignier. Perceptual Components for Context Aware Computing. In Geatano Borriello and Lars Erik Holmquist, editors, *Proceedings of the 4th International Conference on Ubiquitous Computing (UbiComp 2002)*, pages 117–134, Gothenburg, Sweden, September 2002.

[CG89] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4), April 1989.

[CGS+03] G. Chappell, L. Ginsburg, P. Schmidt, J. Smith, and J. Tobolski. Auto-ID on the Line: The Value of Auto-ID Technology in Manufacturing. White paper, Auto-ID Center, February 2003.

[CK00] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical report, Department of Computer Science, Dartmouth College, 2000. Technical Report TR2000-381.

[CMY+02] A. Chen, R. R. Muntz, S. Yuen, I. Locher, S. I. Park, and M. B. Srivastava. A Support Infrastructure for the Smart Kindergarten. *IEEE Pervasive Computing*, 2(1):49–57, 2002.

[Coo04] The Cooltown project. http://www.cooltown.com, November 2004.

[Cro04a] Crossbow Technology Inc. MICA2DOT datasheet. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/6020-0043-03_A_MICA2DOT.pdf, November 2004.

[Cro04b] Crossbow Technonogy Inc. MTS/MDA Sensor and Data Acquisition Boards User's Manual. Rev. A, October 2003, Document 7430-0020-02, November 2004.

[DA00] A. K. Dey and G. D. Abowd. Towards a Better Understanding of Context and Context-Awareness. In *Proceeding of the Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of the 2000 Conference on Human Factors in Computing Systems (CHI 2000)*, The Hague, The Netherlands, April 2000. Appeared also as GVU Technical Report GIT-GVU-99-22.

[Dea03] O. Deak. Interaction with Smart Objects over WAP. Semester thesis, ETH Zurich, 2003.

[Dey00] A. K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.

[Dey01] A. K. Dey. Understanding and Using Context. *Personal and Ubiquitous Computing Journal*, 5(1):4–7, 2001.

[Dey04] A. K. Dey. The Context Toolkit. http://www.cs.berkeley.edu/ dey/context.html, November 2004.

[dIMH02] D. L. de Ipina, P. R. S. Mendonca, and A. Hopper. Trip: a low-cost vision-based location system for ubiquitous computing. *Journal on Personal and Ubiquitous Computing*, 6(3):206–219, March 2002.

[DSA01] A. K. Dey, D. Salber, and G. D. Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction (HCI) Journal*, 16(2–4):97–166, 2001.

[DWFB97] N. Davies, S. Wade, A. Friday, and G. Blair. Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications. In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, pages 291–302, Toronto, Canada, May 1997.

[ECPS02] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, January 2002.

[EHAB99] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges: Data-centric networking for invisible computing. In *Proc. 5th Annual International Conference on Mobile Computing and Networking (Mobicom 99)*, pages 256–262, Seattle, Washington, USA, August 1999.

[EPC04] EPCglobal. http://www.epcglobalinc.org, November 2004.

[Eye04] EYES: Energy Efficient Sensor Networks. http://eyes.eu.org, November 2004.

[FAOH03] C. Floerkemeier, D. Anarkat, T. Osinski, and M. Harrison. PML Core Specification Version 1.0. Auto-ID Center Recommendation, September 2003.

[FFK+02] M. Fleck, M. Frid, T. Kindberg, E. O'Brien-Strain, R. Rajani, and M. Spasojevic. Rememberer: A Tool for Capturing Museum Visits. In Geatano Borriello and Lars Erik Holmquist, editors, *Proceedings of the 4th International Conference on Ubiquitous Computing (UbiComp 2002)*, pages 48–55, Gteborg, Sweden, September 2002.

[FS03] C. Floerkemeier and F. Siegemund. Improving the Effectiveness of Medical Treatment with Pervasive Computing Technologies. Workshop on Ubiquitous Computing for Pervasive Healthcare Applications at Ubicomp 2003, October 2003.

[Gai04] The Gaia Project, Active Spaces for Ubiquitous Computing. http://choices.cs.uiuc.edu/gaia/, November 2004.

[GBK99] H. W. Gellersen, M. Beigl, and H. Krull. The MediaCup: Awareness Technology embedded in an Everyday Object. In *1st Intl. Symposium on Handheld and Ubiquitous Computing (HUC '99)*, Karlsruhe, Germany, September 1999.

[GC92] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2), February 1992.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Gel04] H. W. Gellersen. Ubiquitäre Informationstechnologien. http://www.vs.inf.ethz.ch/publ/papers/UbiqGell.pdf, November 2004.

[GSB02] H.W. Gellersen, A. Schmidt, and M. Beigl. Multi-Sensor Context-Awareness in Mobile Devices and Smart Artifacts. *Mobile Networks and Applications (MONET)*, October 2002.

[GSM04] GSM World. SMS Usage Statistics and Forecasts. http://www.gsmworld.com/news/statistics/, November 2004.

[GWOS00] G. Girling, J. L. K. Wa, P. Osborn, and R. Stefanova. The Design and Implementation of a Low Power Ad Hoc Protocol Stack. In *IEEE Wireless Communications and Networking Conference*, Chicago, USA, September 2000.

[GWPZ04] T. Gu, X. H. Wang, H. K. Pung, and D. Q. Zhang. An Ontology-Based Context Model in Intelligent Environments. In *Proceedings of the Communication Networks and Distributed Systems Modeling and Simulation Conference*, San Diego, California, USA, January 2004.

[Hec01] D. L. Hecht. Printed Embedded Data Graphical User Interfaces. *IEEE Computer*, 34(3):47–55, March 2001.

[HIR02] K. Henricksen, J. Indulska, and A. Rakotonirainy. Modeling Context Information in Pervasive Computing Systems. In *Pervasive 2002*, Zurich, Switzerland, August 2002.

[Hit04] Hitachi Ltd. Hitachi μ-Chip – The World's smallest RFID IC. http://www.hitachi.co.jp/Prod/mu-chip/, November 2004.

[HMS+01] L. E. Holmquist, F. Mattern, B. Schiele, P. Alahuhta, M. Beigl, and H.-W. Gellersen. Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artefacts. In *Ubicomp 2001*, pages 116–122, Atlanta, USA, September 2001.

[Hor00] A. S. Hornby. *Oxford Advanced Learner's Dictionary of Current English*. Oxford University Press, sixth edition, 2000.

[Hor02] I. Horrocks. DAML+OIL: a Description Logic for the Semantic Web. *IEEE Data Engineering Bulletin*, 25(1):4–9, 2002.

[HSR02] S. Hartwig, J. P. Strömann, and P. Resch. Wireless Microservers. *IEEE Pervasive Computing*, 2(1):58–66, 2002.

[HSW+00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS 2000*, Cambridge, USA, Nov. 2000.

[I2C00] The I2C-Bus Specification. Version 2.1, http://www.semiconductors.philips.com/acrobat/literature/9398/39340011.pdf, January 2000.

[iBu04] The iButton home page. http://www.ibutton.com, November 2004.

[Int04] Intel mote. http://www.intel.com/research/exploratory/motes.htm, November 2004.

[JF01] B. Johanson and A. Fox. Tuplespaces as Coordination Infrastructure for Interactive Workspaces. In *UbiTools '01 Workshop at Ubicomp 2001*, Atlanta, USA, 2001.

[Kat94] R. H. Katz. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1(1):6–17, 1994.

[Kel03] P. Keller. Implementierung eines verteilten TupleSpace für Smart-Its. Semester thesis, ETH Zurich, 2003.

[Kel04] P. Keller. Generierung von Kontextinformationen in Umgebungen kooperierender smarter Alltagsgegenstände. Master's thesis, ETH Zurich, 2004.

[KH01] B. Kreller and J. Harmann. The Field Trial Scenario of an Inter-Modal, End-To-End and Real-Time Tracking and Tracing System. In *8th World Congress on Intelligent Transport Systems*, Sydney, Australia, October 2001.

[Kin02] T. Kindberg. Implementing Physical Hyperlinks Using Ubiquitous Identifier Resolution. In *Proceedings of the 11th International World Wide Web Conference*, Honolulu, HI, USA, May 2002.

[KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile Networking for Smart Dust. In *MobiCom 99*, Seattle, USA, August 1999.

[KKP00] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Emerging Challenges: Mobile Networking for Smart Dust. *Journal of Communications and Networks*, 2(3):188–196, September 2000.

[KKPG98] J. Kymissis, C. Kendall, J. Paradiso, and N. Gershenfeld. Parasitic Power Harvesting in Shoes. In *Proceedings 2nd International Symposium on Wearable Computers*, pages 132–139, Pittsburgh, Pennsylvania, USA, October 1998.

[Kon02] S. Konomi. QueryLens: Beyond ID-Based Information Access. In *Proceedings of the 4th international conference on Ubiquitous Computing (Ubicomp 2002)*, pages 210–218, Gothenburg, Sweden, October 2002.

[KPPF03] C. Kintzig, G. Poulain, G. Privat, and P. N. Favennec. *Communicating with smart objects: Developing technology for usable pervasive computing systems*. Kogan Page Science, 2003.

[Kra04] T. Krauer. Spontane Integration mobiler Nutzergeräte in Umbegungen kooperierender smarter Alltagsgegenstände. Master's thesis, ETH Zurich, 2004.

[KS03] A. Kobsa and J. Schreck. Privacy Through Pseudonymity in User-Adaptive Systems. *ACM Transactions on IT*, 3(2), 2003.

[KS04] P. Keller and F. Siegemund. Cluster Tuplespace Reference Manual, 2004. http://www.inf.ethz.ch/siegemun/software/ClusterTuplespace.pdf.

[Lan01] M. Langheinrich. Privacy by Design – Principles of Privacy-Aware Ubiquitous Systems. In *Proc. Ubicomp 2001*, LNCS, pages 273–291, 2001.

[Lan02] M. Langheinrich. A Privacy Awareness System for Ubiquitous Computing Environments. In L.E. Holmquist G. Borriello, editor, *4th International Conference on Ubiquitous Computing (UbiComp2002)*, LNCS, pages 237–245, September 2002.

[LDB03] M. Leopold, M. B. Dydensborg, and P. Bonnet. Bluetooth and Sensor Networks: A Reality Check. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, pages 103–113, Los Angeles, Clifornia, USA, November 2003.

[LH99] P. Ljungstrand and L. Holmquist. WebStickers: Using Physical Objects as WWW Bookmarks. In *ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '99), Extended Abstracts*, Pittsburgh, Pennsylvania, USA, 1999.

[LHB04] J. Lester, B. Hannaford, and G. Boriello. Are You with Me? – Using Accelerometers to Determine If Two Devices Are Carried by the Same Person. In A. Ferscha and F. Mattern, editors, *Pervasive Computing: Second International Conference, PERVASIVE 2004*, LNCS, pages 33–50, Linz/Vienna, Austria, April 2004.

[LMRV00] M. Langheinrich, F. Mattern, K. Römer, and H. Vogt. First Steps Towards an Event–Based Infrastructure for Smart Things. In *Ubiquitous Computing Workshop, PACT 2000*, Philadelphia, USA, October 2000.

[LRH00] P. Ljungstrand, J. Redström, and L. Holmquist. WebStickers: Using Physical Tokens to Access, Manage and Share Bookmarks to the Web. In *Designing Augmented Reality Environments (DARE 2000)*, Elsinore, Denmark, April 2000.

[MAC+99] H. O. Marcy, J. R. Agre, C. Chien, L. P. Clare, N. Romanov, and A. Twarowski. Wireless sensor networks for area monitoring and integrated vehicle health management applications. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, Portland, OR, USA, August 1999.

[Mat01] F. Mattern. The Vision and Technical Foundations of Ubiquitous Computing. *Upgrade*, 2(5):2–6, October 2001.

[Mat02] F. Mattern. Ubiquitous & Pervasive Computing: A Technology-driven Motivation. Presentation, Summer School on Ubiquitous and Pervasive Computing, August 2002.

[Mat04] F. Mattern. Ubiquitous computing: Scenarios for an informatized world. English translation of „Ubiquitous Computing: Szenarien einer informatisierten Welt“ (In: A. Zerdick, A. Picot, K. Schrape, J. C. Burgelman, R. Silverstone, V. Feldmann, D. K. Heger, C. Wolff (Eds.): E-Merging Media – Kommunikation und Medienwirtschaft der Zukunft, Springer-Verlag, pp. 155-174), 2004.

[MC03] R. Min and A. Chandrakasan. Top Five Myths about the Energy Consumption of Wireless Communication. *ACM Sigmobile Mobile Communication and Communications Review (MC2R)*, 6(4), 2003. extended abstract.

[McF02] D. McFarlane. Auto-ID Based Control, An Overview. White paper, Auto-ID Center, February 2002.

[McN03] S. McNealy. The Case Against Absolute Privacy. Opinion article, Washington Post, May 2003.

[Med04] The MediaCup project. http://mediacup.teco.edu, November 2004.

[MKVA03] E. J. Malm, J. Kaartinen, E. Vildjiounaite, and P. Alahuhta. Smart-It Context Architecture. Technical report, VTT, Finland, 2003. Internal report of the Smart-Its project.

[MM71] R. McLeish and J. Marsh. Hydraulic Power from the Heel. *Human Locomotor Engineering*, pages 126–132, 1971.

[Moo65] G. E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8):114–117, April 1965.

[Moo02] S. K. Moore. Just One Word – Plastics. *IEEE Spectrum Online*, September 2002.

[Moo04] Moore's law. http://www.intel.com/research/silicon/mooreslaw.htm, November 2004.

[Mot04] Berkeley Motes. http://www.xbow.com/Products/Wireless_Sensor_Networks.htm, November 2004.

[MSB98] G. Q. Maguire, M. T. Smith, and H. W. P. Beadle. SmartBadges: a wearable computer and communication system. 6th International Workshop on Hardware/Software Codesign, Invited Talk, March 1998.

[Mül97] R. Müller. The CLINICON Framework for Context Representation in Electronic Patient Records. In *Proceedings of the 1997 Fall Symposium of the American Medical Informatics Association*, Nashville, USA, October 1997.

[MZA+00] A. Mehra, X. Zhang, A. A. Ayon, I. A. Waitz, M. A. Schmidt, and C M. Spadaccini. A six-wafer combustion system for a silicon micro gas turbine engine. *Journal of Microelectromechanical Systems*, 9(4):517–526, December 2000.

[NIPA99] L. Nelson, S. Ichimura, E. R. Pedersen, and L. Adams. Palette: A Paper Interface for Giving Presentations. In *ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 354–361, Pittsburgh, Pennsylvania, USA, May 1999.

[NS04] The Network Simulator – ns-2. http://www.isi.edu/nsnam/ns/, November 2004.

[Par04] The IST project ParcelCall. http://www-i4.informatik.rwth-aachen.de/parcelcall/, November 2004.

[Ped01] E. R. Pedersen. Calls.calm: Enabling Caller and Callee to Collaborate. In *Conference on Human Factors in Computing Systems (CHI 2001)*, pages 235–236, Seattle, USA, April 2001.

[Pes04] D. Pescovitz. Smart Buildings Admit Their Faults. Lab Notes, Research from the College of Engineering, University of California, Berkeley, http://www.coe.berkeley.edu/labnotes/1101smartbuildings.html, November 2004.

[Pic04] The PicoRadio project. http://bwrc.eecs.berkeley.edu/Research/Pico_Radio/Default.htm, November 2004.

[PLS+02] S. Park, I. Locher, A. Savvides, M. B. Srivastava, A. Chen, R. Muntz, and S. Yuen. Design of a Wearable Sensor Badge for Smart Kindergarten. In *Sixth International Symposium on Wearable Computers*, Seattle, Washington, USA, October 2002.

[RA00] J. Rekimoto and Y. Ayatsuka. Cybercode: Designing augmented reality environments with visual tags. In *Designing Augmented Reality Environments (DARE 2000)*, Elsinore, Denmark, April 2000.

[Rab03] J. M. Rabaey. Ultra-low Power Computation and Communication enables Ambient Intelligence. Keynote Presentations, ELMO Seminar, Helsinki, Finland and CoolChips VI, Yokohama, Japan, 2003.

[RAK+02] J. M. Rabaey, J. Ammer, T. Karalar, S. Li, B. Otis, M. Sheets, and T. Tuan. PicoRadios for Wireless Sensor Networks – The Next Challenge in Ultra-Low Power Design. In *IEEE International Solid-State Circuits Conference (ISSCC 2002)*, San Francisco, USA, February 2002.

[RB03] M. Rohs and J. Bohn. Entry Points into a Smart Campus Environment – Overview of the ETHOC System. In *International Workshop on Smart Appliances and Wearable Computing (IWSAWC), Proc. 23rd International Conference on Distributed Computing Systems – Workshops (ICDCS 2003 Workshops)*, pages 260–266, Providence, Rhode Island, USA, May 2003.

[RC03] A. Ranganathan and R. H. Campbell. An Infrastructure for Context-Awareness based on First Order Logic. *Personal Ubiquitous Computing*, 2003(7):353–364, 2003.

[RCN02] J. M. Rabaey, A. Chandrakasan, and B. Nikolic. Digital Integrated Circuits, A Design Perspective, July 2002. Presentation available at http://bwrc.eecs.berkeley.edu/IcBook/slides.htm.

[RCRM02] A. Ranganathan, R. Campbell, A. Ravi, and A. Mahajan. ConChat: A Context-Aware Chat Program. *IEEE Pervasive Computing*, 1(3):52–58, 2002.

[RG04] M. Rohs and B. Gfeller. Using Camera-Equipped Mobile Phones for Interacting with Real-World Objects. In A. Ferscha, H. Hoertner, and G. Kotsis, editors, *Advances in Pervasive Computing*, volume 176. Austrian Computer Society (OCG), 2004. ISBN 3-85403-176-9.

[RHC+02a] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A Middleware Platform for Active Spaces. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):65–67, October 2002.

[RHC+02b] M. Roman, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. *IEEE Pervasive Computing*, 2(4):74–83, 2002.

[Rin02] M. Ringwald. Spontaneous Interaction with Everyday Devices Using a PDA. Workshop on Supporting Spontaneous Interaction in Ubiquitous Computing Settings at Ubicomp 2002, September 2002.

[RM98] V. Rodoplu and T. Meng. Minimum energy mobile wireless networks. In *Proceedings of the 1998 IEEE International Conference on Communications (ICC 98)*, volume 3, pages 1633–1639, Atlanta, USA, June 1998.

[RMDS02] K. Römer, F. Mattern, T. Dübendorfer, and J. Senn. Infrastructure for Virtual Counterparts of Real World Objects, April 2002. http://www.inf.ethz.ch/vs/publ/papers/ivc.pdf.

[ROC+03] S. Roundy, B. P. Otis, Y. H. Chee, J. M. Rabaey, and P. Wright. A 1.9GHz RF Transmit Beacon using Environmentally Scavenged Energy. In *IEEE International Symposium on Low Power Elec. and Devices*, 2003.

[Roh03] C. Rohner. *Security in Ad-Hoc Distributed Systems*. PhD thesis, ETH Zurich, 2003.

[Ros04] The RosettaNet Consortium. http://www.rosettanet.org, November 2004.

[RSPS02] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy-Aware Wireless Microsensor Networks. *IEEE Signal Processing Magazine*, 19(2):40–50, March 2002.

[RWR03] S. Roundy, P. K. Wright, and J. Rabaey. A study of low level vibrations as a power source for wireless sensor nodes. *Computer Communications*, 26:1131–1144, 2003.

[SAW94] B. N. Schilit, N. I. Adams, and R. Want. Context-Aware Computing Applications. In *Workshop on Mobile Computing Systems and Applications*, pages 85–90, Santa Cruz, CA, December 1994. IEEE Computer Society.

[SBS02] E. Shih, P. Bahl, and M. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *Mobicom 2002*, Atlanta, USA, September 2002.

[Sch95] B. N. Schilit. *A Context-Aware System Architecture for Mobile Distributed Computing*. PhD thesis, Columbia University, May 1995.

[Sch00] A. Schmidt. Implicit Human Computer Interaction Through Context. *Personal Technologies*, 4(3&4):191–199, June 2000.

[Sch02] A. Schmidt. *Ubiquitous Computing – Computing in Context*. PhD thesis, Computing Department, Lancaster University, November 2002.

[Sch03] R. Schneider. Smoblets – Java-Code zur Interaktion mit aktiven Tags. Semester thesis, ETH Zurich, 2003.

[SDA99] D. Salber, A. K. Dey, and G. D. Abowd. The Context Toolkit: Aiding the Development of Context-Enabled Applications. In *ACM Conference on Human Factors in Computing Systems (CHI 99)*, Pittsburgh, USA, May 1999.

[Sem04] The Semantic Web: An Introduction. http://infomesh.net/2001/swintro/, November 2004.

[Ser04] Java Servlet Technology. http://java.sun.com/products/servlet/, November 2004.

[SF03a] F. Siegemund and C. Floerkemeier. Interaction in Pervasive Computing Settings using Bluetooth-enabled Active Tags and Passive RFID Technology together with Mobile Phones. In *IEEE Intl. Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 378–387, March 2003.

[SF03b] M. Strassner and E. Fleisch. The Promise of Auto-ID in the Automotive Industry. White paper, Auto-ID Center, February 2003.

[SFV04] F. Siegemund, C. Floerkemeier, and H. Vogt. The Value of Handhelds in Smart Environments. *Personal and Ubiquitous Computing*, December 2004.

[Sie02a] F. Siegemund. Kontextbasierte Bluetooth-Scatternetz-Formierung in ubiquitaeren Systemen. In *Michael Weber; Frank Kargl (Eds): Proc. First German Workshop on Mobile Ad Hoc Networks (WMAN 2002)*, pages 79–90, Ulm, Germany, March 2002.

[Sie02b] F. Siegemund. Spontaneous Interaction in Ubiquitous Computing Settings using Mobile Phones and Short Text Messages. In *Workshop on Supporting Spontaneous Interaction in Ubiquitous Computing Settings*, September 2002.

[Sie04] F. Siegemund. A Context-Aware Communication Platform for Smart Objects. In A. Ferscha and F. Mattern, editors, *Pervasive Computing: Second International Conference, PERVASIVE 2004*, LNCS, pages 69–86, Linz/Vienna, Austria, April 2004. Springer-Verlag.

[SK04a] F. Siegemund and P. Keller. Tuplespace-Based Collaboration for Bluetooth-Enabled Devices in Smart Environments. In *Proceedings INFORMATIK 2004, 34. Jahrestagung der Gesellschaft fuer Informatik*, Ulm, Germany, September 2004.

[SK04b] F. Siegemund and T. Krauer. Integrating Handhelds into Environments of Cooperating Smart Everyday Objects. In *Proceedings of the 2nd European Symposium on Ambient Intelligence (EUSAI 2004)*, LNCS, pages 163–173, Eindhoven, The Netherlands, November 2004. Springer-Verlag.

[SL01] A. Schmidt and K. Van Laerhoven. How to Build Smart Appliances? *IEEE Personal Communications*, 8(4):66–71, August 2001.

[Sma04a] Smart Dust: Autonomous sensing and communication in a cubic millimeter. http://robotics.eecs.berkeley.edu/∼pister/SmartDust/, November 2004.

[Sma04b] The Smart-Its Project: Unobtrusive, deeply interconnected smart devices. http://www.smart-its.org, November 2004.

[Sma04c] Smart-Its Particle Prototypes. http://smart-its.teco.edu/artefacts/particle/, November 2004.

[Sow04] John F. Sowa. Conceptual graphs. http://users.bestweb.net/ sowa/cg/cgstand.htm, November 2004.

[SR03] F. Siegemund and M. Rohs. Rendezvous Layer Protocols for Bluetooth-Enabled Smart Devices. *Personal Ubiquitous Computing*, 2003(7):91–101, 2003.

[SS02] A. Savvides and M. B. Srivastava. A Distributed Computation Platform for Wireless Embedded Sensing. In *International Conference on Computer Design (ICCD 2002)*, Freiburg, Germany, September 2002. Invited paper.

[SSB+03] A. Schmidt, F. Siegemund, M. Beigl, S. Antifakos, F. Michahelles, and H. W. Gellersen. Mobile Ad-hoc Communication Issues in Ubiquitous Computing – The Smart-Its Experimentation Platforms. In *Proc. 8th Intl. Conference on Personal Wireless Communication (PWC 2003)*, LNCS, pages 213–218, Venice, Italy, September 2003.

[ST94] B. N. Schilit and M. M. Theimer. Disseminating Active Map Information to Mobile Hosts. *IEEE Network*, 8(5):22–32, 1994.

[Sta00] F. Stajano. The Resurrecting Duckling – What Next? In *8th International Workshop on Security Protocols*, pages 204–214, Springer-Verlag LNCS 2133, April 2000.

[Sta02] F. Stajano. *Security for Ubiquitous Computing*. John Wiley and Sons, February 2002.

[Sta03] T. E. Starner. Powerful Change: Batteries and Possible Alternatives for the Mobile Market. *IEEE Pervasive Computing*, 2(4):86–88, October 2003.

[STM00] A. Schmidt, A. Takaluoma, and J. Mntyjrvi. Context-Aware Telephony over WAP. *Personal Technologies*, 4(4):225–229, December 2000.

[TEA04] The TEA Project, Technology for Enabling Awareness. http://www.teco.edu/tea/, November 2004.

[vH03] L. van Hoesel. Medium access protocols for the eyes sensor nodes. Presentation of the EYES project, February 2003.

[VMKA03] E. Vildjiounaite, E. J. Malm, J. Kaartinen, and P. Alahuhta. Context Awareness of Everyday Objects in a Household. In *1st European Symposium on Ambient Intelligence (EUSAI 2003)*, pages 177–191, Veldhoven, The Netherlands, November 2003.

[Wan99] D. Wan. Magic Medicine Cabinet: A Situated Portal for Consumer Healthcare. In *1st Intl. Symposium on Handheld and Ubiquitous Computing (HUC '99)*, Karlsruhe, Germany, September 1999.

[WB96] M. D. Weiser and J. S. Brown. The Coming Age of Calm Technology. http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm, October 1996.

[Wei91] M. D. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, September 1991.

[Wei93] M. D. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[WFGH99] R. Want, K. Fishkin, A. Gujar, and B. Harrison. Bridging Physical and Virtual Worlds with Electronic Tags. In *ACM Conference on Human Factors in Computing Systems (CHI 99)*, Pittsburgh, USA, May 1999.

[WGZP04] X. Wang, T. Gu, D. Zhang, and H. K. Pung. Ontology-Based Context Modeling and Reasoning using OWL. In *Workshop on Context Modeling and Reasoning (CoMoRea) at IEEE International Conference on Pervasive Computing and Communication (PerCom'04)*, Orlando, Florida, March 2004.

[WHFG92] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The Active Badge Location System. *ACM Transactions on Information Systems*, 10(1):91–102, 1992.

[WKK03] R. Want, L. Krishnamurthy, and R. Kling. Innovative Intel Wireless Platforms Enabling Ubiquitous Computing, September 2003. Intel Developers Forum.

[WLLP01] B. Warneke, M. Last, B. Leibowitz, and K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *IEEE Computer Magazine*, 34(1):44–51, January 2001.

[WSBC04] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A Neighborhood Abstraction for Sensor Networks. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '04)*, pages 99–110, Boston, MA, USA, June 2004.

[Xer04a] Xerox Parc. Electronic Reusable Paper. http://www2.parc.com/dhl/projects/gyricon/, November 2004.

[Xer04b] Xerox Parc. Printed Organic Electronics. http://www.xeroxtechnology.com/ip1.nsf/sedan1?readform& unid=2D1FF1AC91C40AA985256D1A00616714, November 2004.

[XML04] The XML/EDI Group. http://www.xmledi-group.org, November 2004.

[YHE02] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *IEEE Infocom*, New York, USA, June 2002.

[ZWA+02] A. A. Zaharudin, C. Y. Wong, V. Agarwal, D. McFarlane, R. Koh, and Y. Y. Kang. The Intelligent Product Driven Supply Chain. White paper, Auto-ID Center, February 2002.

Appendix A

SICL Syntax

```
program:
    app_stmt stmt_lst
    | app_stmt c_decls stmt_lst c_funcs
    |
    ;

app_stmt:
    SMART OBJECT IDENTIFIER ';'
    ;

stmt_lst:
    stmt ';' stmt_lst
    |
    ;

stmt:
    def_sensor_stmt
    | tuple_decl_stmt
    | search_policy_stmt
    | tuple_stmt
    | actuator_stmt
    ;

def_sensor_stmt:
    DEFINE search_policy_keywords location_specifier
    SENSOR IDENTIFIER '{' selements '}'
    ;

location_specifier:
    LOCAL
    | REMOTE
    |
    ;
```

```
selements:
    tuple_def ';' classifiers
    ;

classifiers:
    classifier ';' classifiers
    |
    ;

tuple_def:
    '<' tuple_fields '>'
    ;

type_decl:
    IDENTIFIER
    ;

tuple_fields:
    type_decl
    | type_decl ',' tuple_fields
    ;

classifier:
    qos
    | getter_fnc
    ;

qos:
    EVERY UNSIGNED_INT MS
    | BEST EFFORT
    | random_interval
    ;

random_interval:
    RANDOM INTERVAL '[' UNSIGNED_INT ',' UNSIGNED_INT ']' MS
    ;

getter_fnc:
    IDENTIFIER IDENTIFIER '(' ')'

tuple_decl_stmt:
    DEFINE search_policy_keywords TUPLE IDENTIFIER '{' tuple_def ';' '}'
    ;

search_policy_stmt:
    search_policy_keyword search_policy_keywords tuple_name_list
    ;
```

```
tuple_name_list:
    IDENTIFIER
    | tuple_name_list ',' IDENTIFIER
    ;

search_policy_keywords:
    search_policy_keyword search_policy_keywords
    |
    ;

search_policy_keyword:
    PRIVATE
    | NON_PRIVATE
    | CALLBACK
    | NON_CALLBACK
    ;

tuple_stmt:
    func_lst LINE tuple_exp TRANSFORM tuple
    | tuple_exp TRANSFORM tuple
    ;

tuple_exp:
    simple_tuple_exp
    | tuple_add_exp
    ;

simple_tuple_exp:
    tuple
    | unary_tuple_exp
    ;

unary_tuple_exp:
    tuple '\''
    ;

tuple_add_exp:
    simple_tuple_exp '+' non_empty_tuple_add_exp
    ;

non_empty_tuple_add_exp:
    simple_tuple_exp
    | simple_tuple_exp '+' non_empty_tuple_add_exp
    ;

tuple:
    IDENTIFIER '<' id_lst '>'
    ;
```

```
id_lst:
    non_empty_id_lst
    |
    ;

non_empty_id_lst:
    IDENTIFIER
    | IDENTIFIER ',' non_empty_id_lst
    ;

func_lst:
    non_empty_func_lst
    |
    ;

func:
    IDENTIFIER '(' id_lst ')'
    ;

non_empty_func_lst:
    func
    | func ',' non_empty_func_lst
    ;

actuator_stmt:
    simple_tuple_exp EXECUTE func
    ;

c_decls:
    CDECLS

c_funcs:
    CCODE
```

Appendix B

Energy Characteristics of Active Tagging Systems

In Tab. 2.3 in Sect. 2.2.3 we have compared the energy consumption of different active tagging systems. However, existing tagging technologies are equipped with a fixed set of components for sensing the environment, data processing, and communication. When comparing concrete solutions with each other, it is therefore not so easy to see which components contribute to low energy consumption. For this reason, Tab. B.1 compares the energy characteristics of commonly used components for the processing and communication subsystems of active tags.

Table B.1: Energy characteristics of core components of active tagging systems.

Data processing	
ATmega128L (according to data sheet)	
Idle, 4 MHz, 4 MIPS, 3 V	2.5 mA
Active, 8 MHz, 8 MIPS, 5V	19 mA
PIC16F876 [BZK+03]	
Idle, 20 MHz, 5 MIPS, 3 V	2.3 mA
Idle, 20 MHz, 5 MIPS, 5 V	4.7 mA
Idle, 4 MHz, 1 MIPS, 3 V	0.4 mA
Communication	
TR1001 [vH03]	
Receive	14.4 mJ/s
Transmit	20 mJ/s
Chipcon CC1000 (according to data sheet)	
Receive, 868 MHz	9.6 mA
Transmit, 868 MHz, 1 mW	16.5 mA
Transmit, 868 MHz, 3 mW	25.4 mA
Bluetooth Zeevo TC2000 (according to data sheet)	
Inquiry/Page	15 mA
Connection	35 mA

The ATmega128L processor is a core component of the Berkeley Motes [Mot04], the BTnodes [Btn04], and the nodes built in the scope of the *Smart Kindergarten* project

[PLS+02]. The Smart-Its hardware platforms built by the TecO institute in Karlsruhe are usually equipped with PIC processors. According to [BZK+03], in the case of the PIC16F876 microcontroller (cf. Tab. B.1) the difference in energy consumption between idle and active mode is negligible.

With respect to communication, the TR1001 modules are used on some of the Smart-Its modules developed by the TecO institute in Karlsruhe [Sma04c], early versions of the Berkeley motes, and the sensor nodes developed in the EYES project [Eye04]. Newer versions of the Berkeley motes and BTnodes are equipped with Chipcon CC1000 communication modules. The BTnodes (revision 3) and the Intel motes [Int04] deploy Bluetooth communication modules similar to the TC2000. In Tab. B.1, the energy consumption of the Bluetooth modules is given for the connection state in general. It is difficult to distinguish between transmit and receive modes since Bluetooth deploys a polling mechanism for scheduling data transmissions. In Bluetooth, each transmission by a master is followed by the reception of a data packet from a slave.

Appendix C

Curriculum Vitae

	Personal Data
	Frank Siegemund Born May 23, 1976 Citizen of Germany
	Education
1995	University entrance diploma, mark 1.0, best student of the year
1995 – 2000	Studied computer science with a minor in mathematics at the University of Rostock
1997	Computer science courses at Queen Mary & Westfield College, University of London
2000	Computer science diploma, mark 1.0, best student of the year
2001 – 2004	Research assistant and PhD student at ETH Zurich
	Professional
1995 – 2000	Part-time jobs to finance university studies Software development for the Fraunhofer-Gesellschaft, the Computer Graphics Center in Rostock, the Department of Computer Science at the University of Rostock, and a local software company Work as a teaching assistant and giving private lessons for students
1998	6-month internship at SAP AG
	Additional Information
Languages	German (native language), English