Integrating Handhelds into Environments of Cooperating Smart Everyday Objects

Frank Siegemund and Tobias Krauer

Institute for Pervasive Computing
Department of Computer Science
ETH Zurich, Switzerland
siegemund@inf.ethz.ch

Abstract. Because of their severe resource-restrictions and limited user interfaces, smart everyday objects must often rely on remote resources to realize their services. This paper shows how smart objects can obtain access to such resources by spontaneously exploiting the capabilities of nearby mobile user devices. In our concept, 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 handheld, and facilitates the collaborative processing of sensory data because of the more elaborate storage and processing capabilities of mobile user devices. We present a concrete implementation of our concepts on an embedded sensor node platform, the BTnodes, and illustrate the applicability of our approach with two example applications.

1 Introduction

Smart environments will be populated by different kinds of computing devices with varying processing power, energy resources, memory capacity, and different means for interacting with users. Handheld devices such as mobile phones or PDAs, 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. However, as pointed out by Mark Weiser [11], "the real power of the concept [of Ubiquitous Computing] comes not from any one of these devices; it emerges from the interaction of all of them." One 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.

This becomes even more important in connection with resource-restricted smart everyday objects, which usually possess only very limited user interface capabilities. Such a smart object can achieve very little on its own and must rely on remote resources to realize its services. Thereby, handheld devices are well suited as resource providers for smart objects because of their complementing capabilities: while there are potentially many smart objects present in a smart environment that can collaboratively provide detailed information about the environment and the context of a user, handheld devices are equipped with powerful storage mediums and have more elaborate input and display capabilities.

This paper shows how smart objects can spontaneously exploit the resources of nearby mobile user devices. Thereby, handhelds participate in a shared data structure established by cooperating objects, and serve as an execution platform for code from smart items. Mobile code is developed in Java, embedded into C code, and stored on embedded nodes, which themselves cannot execute Java programs. We also present an implementation of our approach, consisting of a programming framework, a runtime environment for executing code on nearby user devices, and a corresponding frontend.

By smart everyday objects we understand everyday items such as chairs, books, or medicine that are augmented with active sensor-based computing platforms. Hence, smart objects can perceive their environment through sensors, collect information about the context of a nearby user, and collaborate with other objects in their vicinity by means of wireless communication technologies. In this paper, BTnodes [2] serve as a prototyping platform for augmenting everyday items. BTnodes are equipped with an autonomous power supply, connectors for external sensor boards, and Bluetooth modules for communication (cf. Fig. 1).

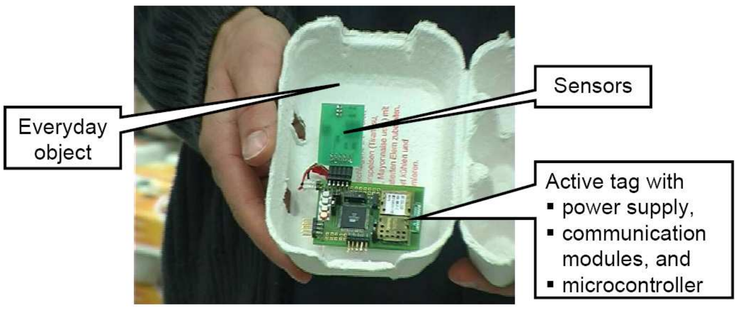


Fig. 1. A smart everyday object: an everyday item augmented with a sensor-based computing platform.

The rest of this paper is structured as follows: Section 2 summarizes related work. Section 3 presents our concepts for integrating handhelds into smart environments, while section 4 describes a programming framework for developing mobile code for smart objects. Section 5 presents the runtime environment for outsourcing computations to handheld devices. Section 6 evaluates our implementation, and section 7 presents two example applications. Section 8 concludes the paper.

2 Related Work

Hartwig et al. [3] integrates small Web servers into embedded devices and enables people to interact with them using mobile phones and a WAP (Wireless Application Protocol) interface. In this paper, we focus on the integration of handhelds into sets of cooperating objects, not on the interaction with a single device. We also want to enable smart objects to outsource code for energy-consuming computations to handheld devices in order to spontaneously exploit their resources. This does not necessarily require user interaction but can instead be transparent for a user. Furthermore, by outsourcing Java code to a mobile device it is possible to support more complex user interfaces than with WAP pages.

Aglets [6] is a programming framework for mobile agents based on Java technology. Whereas mobile agents migrate from execution platform to execution platform transferring their code, data, and execution state, we only transfer code from a smart object to a nearby handheld device. Other data are not transmitted but are made available to all cooperating objects by means of a distributed data structure. Furthermore, programming frameworks like the Aglets rely on RMI for code shipping and a Java Virtual Machine (JVM) that must run on every node. In our case, the embedded platforms are so ressource-restricted that they usually do not support a full-fledged JVM. Instead, virtually all embedded sensor node platforms are programmed using the C programming language.

The Stanford Interactive Workspaces project [4] introduces a tuplespace-based infrastructure layer for coordinating devices in a room. This tuplespace is centralized and runs on a stationary server, whereas we distribute a shared data structure among cooperating smart objects and handheld devices, and do not assume that there is always a powerful server in wireless transmission range.

Want et al. [10] augments everyday items with passive RFID tags and thereby provides them with a representation in the virtual world. Other passive tagging technologies such as barcodes and two-dimensional visual codes have also been used to link real and virtual worlds by attaching them to everyday things [5]. Although handhelds with attached scanning devices are used to read out those tags, an application associated with an object is usually provided by a service in the background infrastructure. In our approach, we do not rely on an always-available background infrastructure link. Instead, because of the active tagging technology used to augment everyday items, we enable smart objects to offer their services independently from a backend infrastructure.

3 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[1]. Thereby, 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 be executed also 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 in which an Applet is downloaded from a remote Web server. Fig. 2 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.

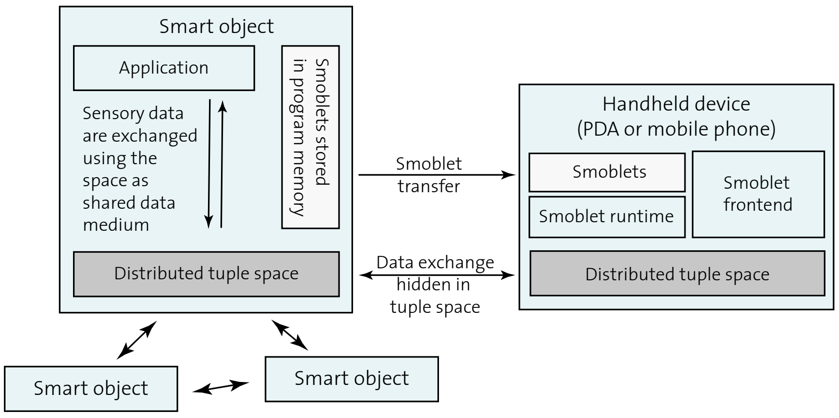

Smart object
Application
Smoblets stored in program memory
Sensory data are exchanged using the space as shared data medium
Distributed tuple space
Smoblet transfer
Data exchange hidden in tuple space
Handheld device (PDA or mobile phone)
Smoblets
Smoblet runtime
Smoblet frontend
Distributed tuple space
Smart object
Smart object

Fig. 2. 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

[1] Please refer to http: www.inf.ethz.ch/˜siegemun/software/ClusterTuplespace.pdf for a more detailed description of our implementation.

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 but still operate on the data basis of the smart object that provided the code. The sole reason for storing Smoblets in program memory is the memory architecture of many embedded sensor node platforms, 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 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 every 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 [8] 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 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. 3): (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.

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

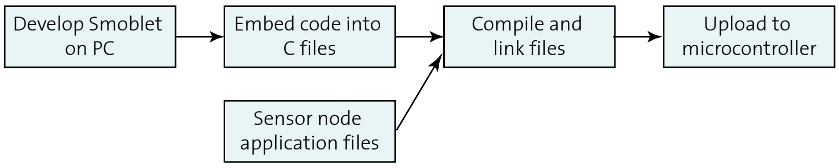


Fig. 3. The process of deploying a Smoblet on a sensor node.

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 entire 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, which shows selected parts of a Smoblet for collecting microphone samples from nearby smart objects.

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

```
import smoblet.*;

public class MicCollector extends BasicSmoblet {

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

    public boolean isAutoStart() {
        return true;
    }

    public boolean onInit() { . . . }

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

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

Fig. 4. Selected methods from a Smoblet without graphical user interface.

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 human-readable 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 lines 5-11). These informational methods are mainly used by the tool that embeds Smoblets into C code for storage on an embedded platform. This tool for converting Java files loads Smoblet classes, retrieves information about them, and stores these information in addition to the encoded class files on a sensor node. Consequently, when a user wants to lookup Smoblets on a particular smart object, it is not necessary to download the complete Smoblet to get this information. Instead, the object shares these data by means of the distributed data structure with nearby handhelds.

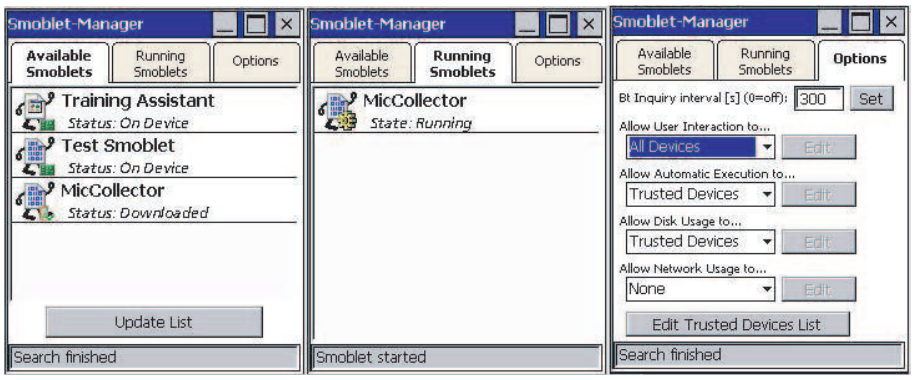
The methods controlling a Smoblet's execution state are executed by the runtime environment on a mobile user device after the code has been successfully downloaded from a smart object. The *onInit* method, for example, is executed immediately after a Smoblet has been started, followed by the *onRun*, *onAbort* or *onExit* methods (cf. lines 13-21 in Fig. 4). 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 is leaving 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). 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 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: functions for accessing the local data store, i.e., the local tuplespace; functions 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). Thereby, it relieves resource-restricted sensor nodes from storing microphone samples by putting them in its own memory.

5 The Smoblet Front- and Backend

The Smoblet frontend (cf. Fig. 5) is a graphical user interface for searching 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. If the user permits it, a handheld device can also serve as an execution platform for nearby smart



Training Assistant
Status: On Device
Test Smoblet
Status: On Device
MicCollector
Status: Downloaded
Search finished

MicCollector
State: Running
Smoblet started

Bt Inquiry interval [s] (0=off): 300
Allow User Interaction to... All Devices
Allow Automatic Execution to... Trusted Devices
Allow Disk Usage to... Trusted Devices
Allow Network Usage to... None
Search finished

Fig. 5. The Smoblet Manager: the tool for searching Smoblets and restricting access to resources.

objects without requiring manual interaction. In this case, the mobile user device continually searches for Smoblets on nearby devices. If a Smoblet wants to be automatically executed its code is then downloaded and started.

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 often takes more than 10 s) [9]. As a possible solution, we propose an explicit selection mechanism based on passive RFID tags to determine the device address of a smart item. Thereby, 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, the mobile code can be immediately downloaded to the handheld device and the graphical user interface be started. To experiment with such explicit selection mechanisms for triggering interactions with smart objects, we connected a small-size (8cm $\times$ 8cm) RFID reader over serial line to a PDA.

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

6 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.

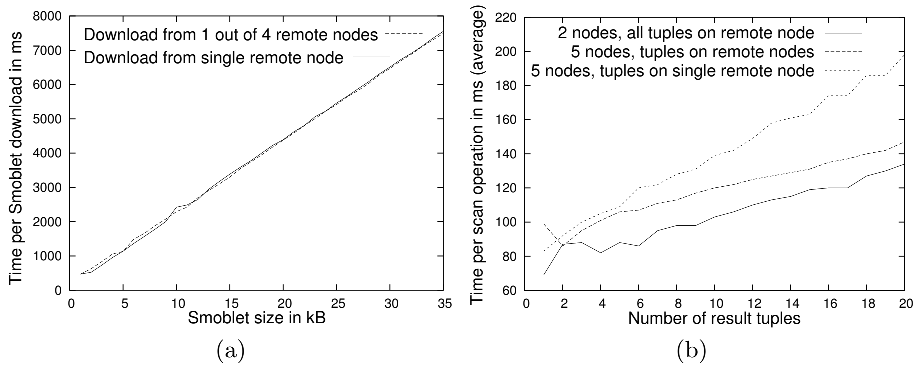


Fig. 6. (a) Time needed for downloading Smoblets from smart everyday objects; (b) time needed for a *scan* operation on the shared data structure.

Fig. 6 a) 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 [7]. 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, threads that are used to continuously query for devices in range and for accessing the shared data structure. That Bluetooth is not the dominating factor can also be seen in Fig. 6 a) because the time for download 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 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 Section 7 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.

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 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. 6 b) 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. Thereby, 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 section 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. 6 b), 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.

7 Applications

There are two major application domains for Smoblets: (1) exploiting the computational resources of nearby handhelds in order to facilitate collaborative context recognition and (2) enabling graphical user interaction with smart objects by outsourcing user interfaces to nearby handhelds. In this section, we present one example from each of these two application areas.

Handling large amounts of sensory data. 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. By using the proposed concepts, nearby handheld devices can help in handling large amounts of sensory data during the collaborative context recognition process.

As an example, we have implemented a Smoblet for evaluating which smart objects are in the same room and for finding out what is happening at a specific location. This is done by means of low-cost microphones attached to smart objects (for our experiments we have used the sensor boards described in [1]). When a user carrying a handheld device comes into the range of a smart object, it automatically transmits a MicCollector Smoblet (cf. Fig. 4) to the mobile user device, where it is automatically started. Smart objects continuously sample their microphones at approximately 40 kHz and extract a feature from these readings indicating the level of activity in their room. In this example, we sam-

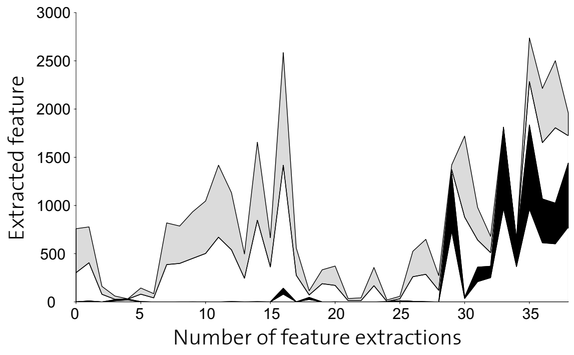


Fig. 7. Small subset of the data assembled by the MicCollector Smoblet reflecting the microphone measurements of four remote smart objects.

ple microphones continuously for approximately 500 ms and use the number of crossings through the average microphone sample as feature. The MicCollector Smoblet running on a handheld device retrieves these features from *all* smart objects in range, thereby relieving them from storing these data. After several readings, the MicCollector Smoblet can then 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 powerful computational capabilities of handhelds it can thereby carry out more demanding algorithms for evaluating sensory data. Fig. 7 shows the microphone features collected by the MicCollector Smoblet from four devices. 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 devices in one room are correlated, meaning that they decrease and increase simultaneously. This fact is exploited during the context recognition process on the handheld.

Providing user interfaces. The presented approach for outsourcing computations can facilitate user interaction with smart objects, which usually do not

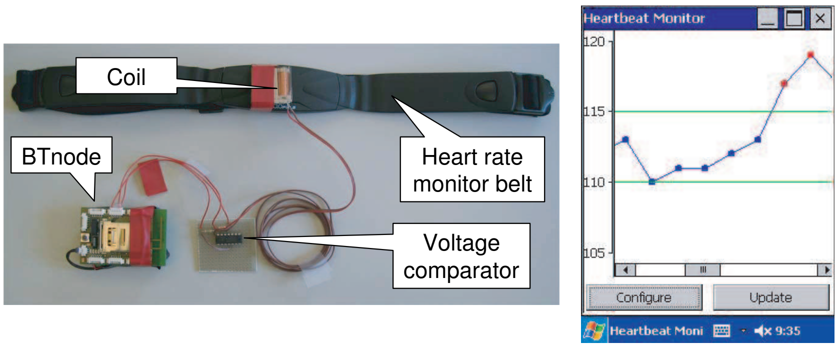


Fig. 8. (a) A BTnode with an attached heart rate sensor; (b) the user interface downloaded from the BTnode to visualize the pulse data collected during the last training.

possess keys or displays. To show the applicability of our concepts in this area, we have augmented a heart rate monitor belt with a BTnode that records the electromagnetic pulses generated by the belt during a training (cf. Fig. 8). The BTnode also carries a Smoblet that can be downloaded to a handheld device. Thereby, a user interface is transmitted to the mobile user device that allows a sportsman to evaluate the last training.

8 Conclusion

In this paper, we presented an approach that enables smart everyday objects to spontaneously access the capabilities of nearby mobile user devices. Thereby, smart objects outsource computations to nearby handhelds and hence can dynamically exploit their resources. A distributed data structure facilitates the cooperation with multiple smart items and enables handhelds to access remotely-generated sensor values. We have evaluated our concepts based on a concrete implementation, and identified two application domains – collaborative context recognition and graphical user interaction with smart objects – in which they prove to be valuable.

References

1. M. Beigl and H. W. Gellersen. Smart-Its: An Embedded Platform for Smart Objects. In *Smart Objects Conference (SOC) 2003*, Grenoble, France, May 2003.
2. 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)*, Berlin, Germany, January 2004.
3. S. Hartwig, J.-P. Strömann, and P. Resch. Wireless Microservers. *IEEE Pervasive Computing*, 2(1):58–66, 2002.
4. B. Johanson and A. Fox. Tuplespaces as Coordination Infrastructure for Interactive Workspaces. In *UbiTools '01 Workshop at Ubicomp 2001*, Atlanta, USA, 2001.
5. T. Kindberg et al. People, Places, Things: Web Presence for the Real World. In *WMCSA 2000*, Monterey, USA, December 2000.
6. D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.
7. 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.
8. F. Siegemund. A Context-Aware Communication Platform for Smart Objects. In *Proceedings Second International Conference on Pervasive Computing, Pervasive 2004*, pages 69–86, Linz/Vienna, Austria, April 2004.
9. F. Siegemund and M. Rohs. Rendezvous Layer Protocols for Bluetooth-Enabled Smart Devices. *Personal Ubiquitous Computing*, 2003(7):91–101, 2003.
10. 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.
11. M. D. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):66–75, September 1991.