A Context-Aware Communication Platform for Smart Objects

Frank Siegemund

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

Abstract. When smart objects participate in context-aware applications, changes in their real-world environment can have a significant impact on underlying networking structures. This paper presents a communication platform for smart objects that takes an object's current real-world context into account and adapts networking structures accordingly. The platform provides (1) mechanisms for specifying and implementing context-aware communication services, (2) a tuplespace-based communication abstraction for inter-object collaboration, and (3) support for linking communication and context-recognition layers. For specifying context-aware communication services, a high-level description language, called SICL, and a compiler that generates corresponding code for smart objects were realized. The tuplespace-based infrastructure layer for inter-object collaboration hides low-level communication issues from higher layers and facilitates collaborative context recognition between cooperating objects. The paper also presents examples of context-aware communication services and evaluates the platform on the basis of a concrete implementation on an embedded device platform.

1 Introduction

As pointed out by Weiser and Brown[12], Pervasive Computing "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 everyday items with information technology by augmenting ordinary objects with small sensor-based computing platforms (cf. Fig. 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.

But networks of collaborating smart objects are extremely difficult to manage: some objects are mobile, often with distinct resource restrictions, communication is dynamic, takes place in a highly heterogeneous environment, and

To appear: Proc. 2nd International Conference on Pervasive Computing (Pervasive 2004)

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 even aggravates the problem of keeping communication efficient.

Our approach to addressing this problem is 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.

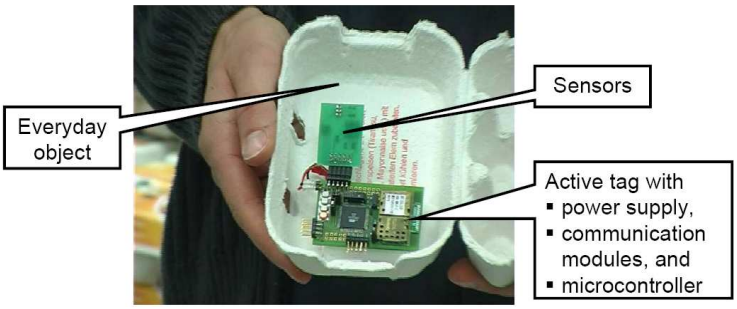

Fig. 1. A smart object: an everyday item augmented with a sensor-based computing platform.

Location-based routing protocols [1] already consider sensory information to improve the effectiveness of networking services. Also, Katz [6] 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.

In this paper, we present a platform for building context-aware communication services. It consists of (1) components for the description and generation of context-aware communication services, (2) a tuplespace-based communication abstraction for inter-object collaboration, and (3) mechanisms that enable communication services to access context information. Regarding service generation, we developed a description language, called SICL, for the design of context-aware services, and a compiler that, given an SICL program, automatically generates code for smart objects. The tuplespace-based infrastructure layer for inter-object collaboration is the basis for deriving context information from multiple sensor

sources. It also hides low-level communication issues from context-recognition and application layers. Furthermore, we present concrete examples of context-aware communication services that show how to link communication and context layers.

The rest of this paper is structured as follows: In the next section we review related work. Section 3 gives an overview of our platform for building context-aware communication services. Section 4 presents a description language for context-aware services. In section 5, we describe the tuplespace-based infrastructure layer for inter-object collaboration. Section 6 gives concrete examples of context-aware communication services, and section 7 concludes the paper.

2 Related Work

In [8] and [10], we proposed to consider the real-world context of smart objects in the design of specific networking protocols. In this paper, we build on our previous work, generalize our approach, and describe necessary infrastructural components that facilitate context-aware communication services.

Beigl et. al [3] reports on a networking protocol stack for context communication, and suggests to consider situational context in order to improve its efficiency. Thereby, 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 kind of adaptiveness.

The Stanford Interactive Workspaces [5] project 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 the tuplespace among autonomous smart objects and do not assume that there is always a server in wireless transmission range.

Want et al. [11] augments physical objects with passive RFID tags to provide them with a representation in the virtual world. A similar approach is taken in the Cooltown project [14]. Our approach to building smart objects, however, is that of the Smart-Its [15] and MediaCup [2] projects, where active instead of passive tags are used to augment everyday objects. This allows them to implement more sophisticated applications and to autonomously process information. Consequently, smart objects do not depend on always available background infrastructure services that implement applications on their behalf.

3 Architecture Overview

Smart objects usually implement context-aware applications. Thereby, they derive the context of nearby users in collaboration with other smart objects and other sensor sources distributed throughout their environment. A context-aware

application then adapts its behavior on the basis of the derived context. A typical software architecture for smart objects therefore consists of a communication layer, a context and perception layer, and an application layer. Fig. 2 shows how the main tasks of these separate layers have been addressed in our platform. In the following, we give a more detailed overview of these layers and show how they influence the design of context-aware communication services.

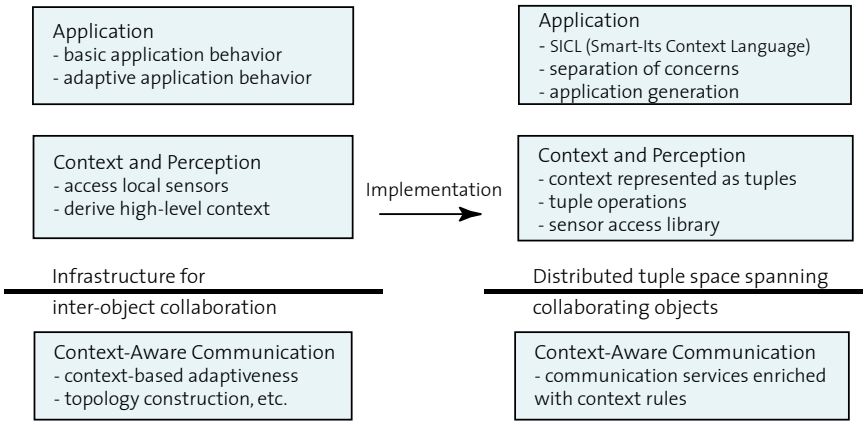

Fig. 2. A software architecture for smart objects and its implementation.

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) that separates these concerns and helps programmers to write more legible and structured code (cf. Fig. 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. Our goal was to implement context-aware communication services using the same description language, and to link communication and application layer by exporting communication modules as sensors (cf. Sect. 6).

Context and Perception Layer. The context and perception layer derives the current situational context of a nearby person or of a smart object itself. Sensor access and context recognition is specified in SICL as part of an application. However, the SICL-compiler generates the actual code for the context and perception subsystem, which operates on the smart objects. The subset of the generated code responsible for querying sensors and fusing data makes up the context and perception layer. It provides information about an object's context

to the application layer, which is then able to adapt according to this information (cf. Fig. 2). In our current implementation, context information is represented as tuples and the derivation of higher-level context information as tuple transformations. Thereby, SICL statements are translated into corresponding rules for merging and finding context tuples. Such tuples might originate from other smart objects and the results of a tuple operation might also be of potential interest to other smart objects in vicinity. 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. 2). This data space allows objects to share data as well as resources, and to fuse sensory data originating from different smart objects. This data space has been implemented as a distributed tuplespace. The distributed tuplespace hides low-level communication issues from higher layers, and it is also a means to group sensor nodes. As the accuracy of sensory data sharply decreases with increasing distance – 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. Such objects can therefore be grouped into a tuplespace that operates on a dedicated wireless channel, which makes tuplespace operations very efficient (cf. Sect. 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 [4] is used in our experiments, the communication layer consists in its basic form of a Bluetooth protocol stack. The context-aware communication platform considerably extends this protocol stack with mechanisms that adapt according to results from the context layer. Thereby, communication services contain rules for the context-layer that describe how to derive relevant context and how to change networking parameters in certain situations. 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 such that communication between these devices is more efficient (cf. Sect. 6.3). We call communication services that take input from the context layer into account *context-aware*.

4 A Description Language for Context-Aware Services

Developing context-aware communication services in a language that natively supports embedded platforms (usually C) often leads to unnecessarily complex and error-prone code. Consequently, we have designed a high-level description language – the Smart-Its Context Language (SICL) – which facilitates the development of context-aware services and applications.

While developing a range of such context-aware applications [4, 9], we found that they all have a similar recurrent structure. This structure consists of four parts that deal with (1) basic application behavior, (2) the context-aware adaptiveness of an application, (3) access to local sensors, and (4) context recognition. Unfortunately, the C programming language does not encourage application programmers to clearly separate these different concerns, which often leads to illegible, unstructured and error-prone code. For these reasons, we decided to automatically generate C code from a more concise higher-level description of the specific application parts. Thereby, SICL aims at simplifying 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.
- The adaptive part of an application, that is, how the application reacts to results from the context recognition process, can be clearly formulated.
- SICL enables a programmer to specify how to access sensors.

```
1:  smart object name;

    %{
4:  C declarations
    %}

    /* sensor statements */
8:  define [remote|local] sensor name {
9:      tuple type declaration;
10:     sensor access function;
11:     sensor access policy;
12: };

    /* tuple transformations */
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 >;

    /* adaptation rules */
20: tup < field_1, ..., field_p > -> func(arg_1, ..., argn);

    %%

24: C code
```

Fig. 3. The basic structure of an SICL program.

As a result, it becomes easier, faster, and less error-prone to implement context-aware applications – and context-aware communication services in particular – on an embedded sensor node platform. Fig. 3 shows the basic structure of an SICL program and Fig. 4 presents a concrete example. An SICL program consists of the following main parts that reflect the previously identified tasks of a typical context-aware application:

Sensor access. Sensor statements (cf. lines 8-12 in Fig. 3) describe how to deal with local and remote sensors. In our program model, sensor readings are embedded into tuples and used in transformations to derive new tuples that finally represent higher-level context information. As sensor tuples are not generated in tuple transformations, their type – i.e., the type of every field in a sensor tuple – must be specified (cf. line 9 in Fig. 3). Also, if a sensor is local, a function must be given that reads out a sensor, creates a tuple with the specified type, and writes it into the distributed tuplespace. It is also possible to determine when to read out a sensor by either providing the amount of time between consecutive sensor readings or by leaving it to the system to access the sensor as often as possible.

```
smart object egg_box;

%{
#include "sensors.h";
#include "gsm.h";
%}

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

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

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

%%
```

Fig. 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.

Context recognition. Tuple transformations (cf. line 15 in Fig. 3) describe how tuples are transformed and new tuples are created. This reflects the process of deriving high-level context information by fusing sensor readings from different nodes. Thereby, it is unimportant at which specific smart object a tuple has been created, because an infrastructure for inter-object collaboration makes it available to all collaborating nodes.

The basic structure of a transformation rule is as follows. The left-hand side of a rule lists the tuples that must be available for the rule to be executed, and the right-hand side specifies the tuple that is to be generated. A set of functions can be specified that operate on the fields of left-hand side tuples. The tuple transformation is then only carried out when all these functions return true. When more than one rule can be executed, the rule first specified in the SICL description is executed first.

Adaptive application behavior. Adaptation rules (cf. line 20 in Fig. 3) represent the adaptive part of a context-aware application in that they describe how an application needs to react when a certain context is derived. An adaptation rule simply consists of a tuple 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.

Basic application behavior. The basic application behavior of a smart object is provided in form of ordinary C code. SICL allows 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.

The SICL compiler was implemented using lex and yacc and is exceptionally slim, consisting of only around 2'500 lines of code.

5 Tuplespace-Based Inter-Object Collaboration

As mentioned before, context-aware services – and context-aware communication services in particular – need a platform to disseminate information to other nodes, to access sensors at various smart objects, and to process these sensory data. This section describes the infrastructure component that facilitates this kind of cooperation among smart objects: a distributed tuplespace on context-based broadcast groups.

5.1 Basic Approach

The distributed tuplespace handles all basic communication-related issues required for inter-object collaboration. It hides low-level communication issues from higher layers and serves as a shared data structure for collaborating smart objects. Thereby, every object provides a certain amount of memory for storing tuples, and the tuplespace implementation handles the distribution of data among nodes. The context and perception layers access local sensors and embed the corresponding data into tuples, which are written into the space and are therefore accessible from all smart objects participating in the distributed tuplespace.

We extend the basic concept of distributed tuple spaces by grouping nodes into a space according to their current context, and by assigning such groups a dedicated broadcast channel (cf. Fig. 5). For example, nodes that are in the same office room 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.

Because Pervasive Computing networks are likely to be dense, nodes in the same distributed tuplespace communicate on a dedicated wireless channel. This minimizes the amount of collisions with other nodes in the same area and considerably improves the effectiveness of tuplespace operations (cf. Sect. 5.4). In

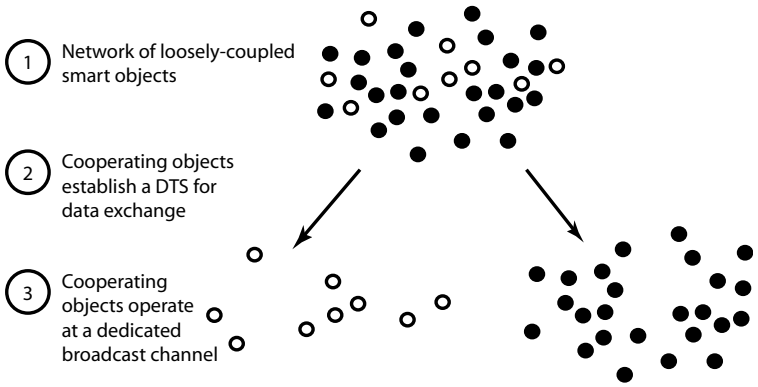

Fig. 5. Grouping of collaborating smart objects into distributed tuple spaces that operate on different communication channels.

case of frequency-hopping spread spectrum techniques 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 [13], for example, support multiple frequencies but do not implement frequency hopping patterns.

5.2 Efficiency Discussion

An infrastructure for inter-object collaboration must consider the energy consumption of nodes and the effectiveness of data exchange (i.e., how fast data can be transferred between objects). We argue that our approach is effective with respect to both aspects in typical Pervasive Computing networks, which share the following characteristics:

1. **Tight collaboration between adjacent nodes.** Sensor readings originating from smart objects that are in close proximity of a user are likely to provide the most accurate information for characterizing the user's situation. Therefore, in order to realize context-aware applications, primarily objects that are in close vicinity need to exchange data. Consequently, cooperating objects are often in range of each other.
2. **Short-range communication.** Because energy-consumption is a major concern in the envisioned settings, smart objects exchange data using short-range wireless communication technologies. A major property of corresponding communication modules is that their energy consumption for sending data is approximately as high as that for receiving data.

Criterion (1) suggests that collaborating smart objects are often in range of each other. Consequently, collaborating nodes that are grouped in a distributed tuplespace can reach each other by broadcasting data. Because nodes in a tuplespace transmit on dedicated channels they are also the only nodes that can

receive such data. Tuplespace operations can therefore be implemented by broadcast protocols, which are more efficient than unicast-based solutions (cf. Sect. 5.4).

Because sending data is approximately as energy-consuming as receiving data (cf. criterion (2)), nodes can directly transmit data to other nodes in range without wasting energy. That means, objects would not save energy by reducing their transmission range and routing data over intermediate nodes, because receiving is so expensive. Consequently, when grouping nodes on broadcast channels their actual distance is not important as long as they can reach each other. Grouping of nodes into broadcast channels does therefore not have to consider node distance but can take other parameters – such as context – into account. As a result, our approach to grouping nodes according to their context rather than distance is energy-efficient. Please note, that this is not true for general wireless networks because for long-range communication technologies it is usually more energy-efficient to reduce the transmission range and to transmit data over multiple short distance hops instead of over one long-distance hop [7].

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 [4].

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. Thereby, Bluetooth distinguishes *master* and *slave* roles. The master of a piconet is the node in the centre of the star topology; slaves cannot communicate directly with each other but only over 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, which results in very few collisions with other piconets in the same area. Multiple piconets can form so called scatternets. Thereby, bridge nodes participate in more than one piconet and switch between piconet channels in a time division multiplex (TDM) scheme.

Our implementation tries to group collaborating smart objects into one piconet. This corresponds to the aim, according to which tightly cooperating objects shall be grouped such that they operate 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. 6). As embedded systems often do not pro-

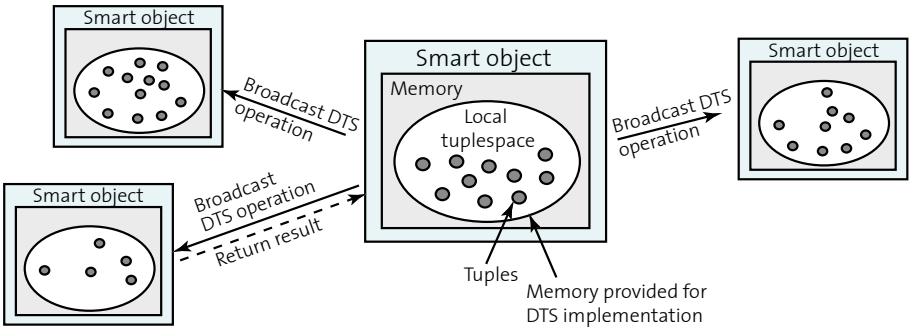

Fig. 6. Overview of the distributed tuplespace implementation: multiple local tuple spaces are connected to form a distributed tuplespace.

vide 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 the storage of tuples. Because of the memory restrictions of embedded systems, for example, tuple fields have always 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 are restricted in size. The memory restrictions 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 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.

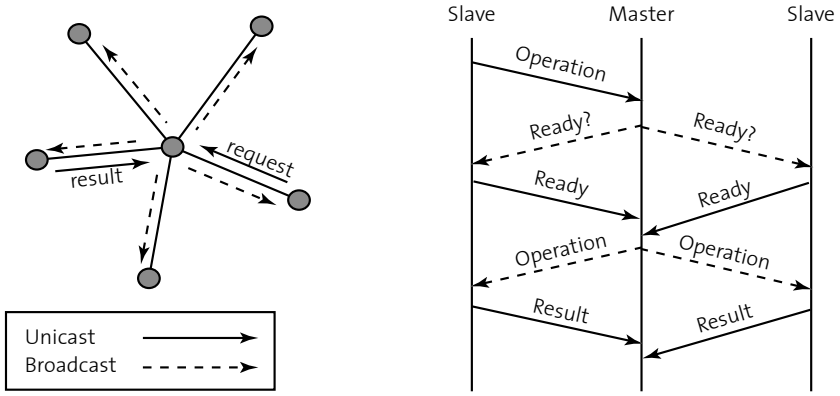

Fig. 7. Basic protocol for distributed tuplespace operations.

The DTS 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, however, we focus on functions that operate on all nodes in a distributed tuplespace and describe the underlying protocol based on an example for a *read* operation (cf. Fig. 7). When a slave wants to execute a *read* operation on the distributed tuplespace, it must send the request first to the master of its piconet. Next, the master broadcasts a *ready?* message to all nodes on the broadcast channel and waits for their response. The explicit response is necessary because broadcast traffic in wireless networks is usually unacknowledged. When all nodes acknowledged the request, the master broadcasts the actual operation together with all necessary parameters to its slave and awaits the results (cf. Fig. 7).

We would like to emphasize that the concepts of our implementation are in no means restricted to a specific communication standard. In fact, porting our code to other device platforms, as for example the Berkeley motes [13], would be possible because they do also support multiple communication channels.

5.4 Evaluation

Based on measurements, this section tries to evaluate the performance of the presented implementation.

A central question is whether the broadcast-based implementation for data sharing is superior to a more simpler, unicast-based solution. The *read* operation presented in the previous section, for example, queries all nodes for tuples matching a given template. However, *read* only requires one matching tuple. In Fig. 8 we compared the broadcast-based solution with a unicast-based read

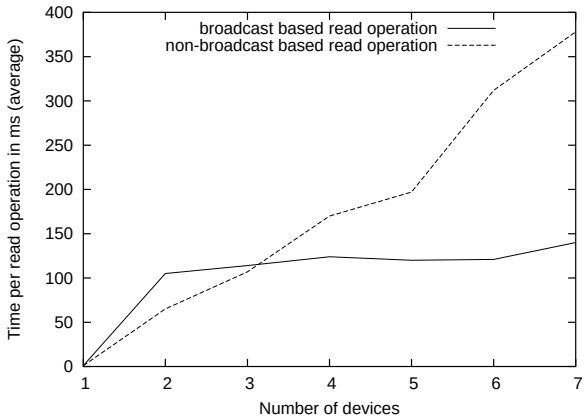

Fig. 8. Performance of a broadcast-based *read* operation compared to a unicast-based implementation.

operation, which queries all nodes after another and stops as soon as it receives a result from one node. In the experiment, there was always a matching tuple on one of the nodes. This node was picked with equal probability from all available nodes, and the *read* operation was invoked directly at the master.

As can be seen, the broadcast-based implementation clearly outperforms the unicast-based version, especially with increasing number of nodes. A reason for this is that because of the broadcast, fewer packets are transmitted over the medium when there are many nodes on the broadcast channel. However, this fact alone does not fully explain the graphs in Fig. 8. We also noticed that the whole process of sending a unicast packet and receiving a response from one node always takes about 35 ms. On the other hand, sending a broadcast packet and receiving the results takes between 50 and 70 ms for two to seven nodes. We think that this is due to the Bluetooth modules used on the BTnodes, which seem to delay packets before delivering them to the microcontroller when packets are not streamed. Also, the time division duplex scheme implemented in Bluetooth decreases the efficiency of the unicast-based protocol.

We have also made measurements regarding the performance of tuplespace operations executed during the context recognition process. As mentioned before, SICL rules (cf. Fig. 3 in Sec. 4) are translated by the SICL compiler into tuplespace operations for obtaining relevant tuples, merging them, and writing the corresponding results back to the space. Given a typical SICL program, usually all global tuplespace operations except *consumingScan* are executed during this process. The reason for this is that the context recognition layer cannot decide by the time it is scanning for tuples whether to remove them from the space or not. It can only do this when it has evaluated these tuples, and then only deletes selected tuples by calling the *take* operation instead of *consumingScan*. Please note that SICL offers a special operator for deleting tuples after a rule has been successfully evaluated. Fig. 9 shows the number of tuplespace operations

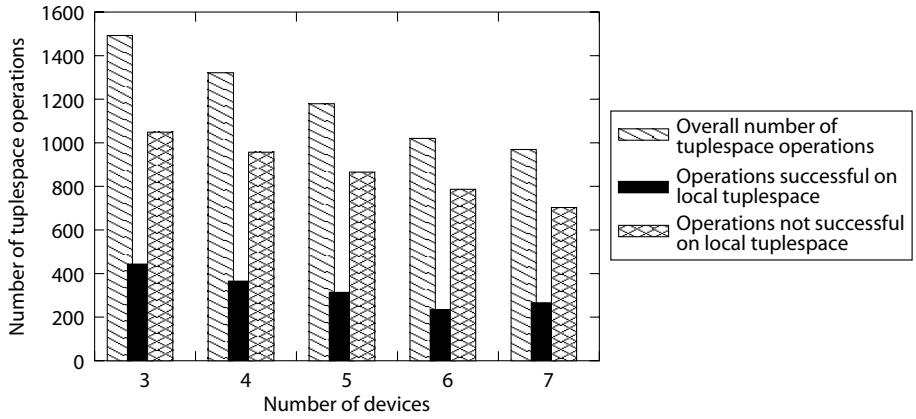

Fig. 9. Number of operations executed on the distributed tuplespace during the context-recognition process under heavy load in t = 3 min.

executed during the context-recognition process with respect to the number of collaborating nodes. To test the performance of the tuplespace operations under heavy load, the underlying context-recognition code was generated from an SICL program such that global tuplespace operations are continuously issued by all participating nodes without periods of low activity. The generated code was uploaded to and executed on all collaborating nodes, not only on the master. The measurements in Fig. 9 show that the number of executed tuplespace operations that require communication with other nodes only decreases from around 1000 to 750 when the number of slaves increases from two to six. Because the presented grouping procedure reduces the number of nodes in a DTS, this is a very promising result. Also, the time needed for executing tuplespace operations on the local space is negligible because of the memory-restrictions of smart objects. The number of local operations only depends on the structure of the SICL program.

6 Context-Aware Communication Services

Context-aware communication services are communication services that take an object's current situational context into account. They can either adapt their behavior according to changes in the real-world environment or parameterize conventional communication services with context information. For example, a service for context-aware device discovery could return not only the devices currently in range but could also determine which of them share the same symbolic location. Context-aware communication services can also completely substitute conventional communication mechanisms (cf. Sect. 6.2). In this section, we describe how we linked the communication and context layers in order to allow communication services to access and process context information. We also present two examples of context-aware communication services.

6.1 Linking Communication and Context Layer

We designed the context-aware communication platform 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 the device discovery functionality of a communication module as a sensor statement in SICL (cf. Fig. 10). Thereby, 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. 10, 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 module discovered (lapd and uapd).

```
define sensor communication_module {
    <lapi, uapi, lapd, uapd>;
    void start_inquiry();
    every 60000 ms;
};
```

Fig. 10. Linking communication and context layer by treating communication modules as sensors: sensor statement for a communication module in SICL.

By linking communication and context layer in this way, communication services can take advantage of all the services and infrastructural components presented in the previous sections. Thus, they can access derived context information and sensor readings from collaborating objects.

6.2 Context-Aware Device Discovery

In some communication technologies, device discovery can be extremely slow. Especially for frequency-hopping technologies – such as Bluetooth – where there is no dedicated frequency at which devices 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 discovery mechanism [10], 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 sensor rfid {
    <lap, uap>;
    void read_device_address_from_rfid();
    best effort;
};

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

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

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

Fig. 11. The code for the smart door in SICL.

In the *smart door* application, a door is augmented with a BTnode and an attached RFID reader, which serves as sensor for the BTnode. Other smart objects have an RFID tag attached that contains their actual device address, such as their Bluetooth address. When they enter a room through the smart door, the device address is read out of the tag by the RFID reader and made available to other objects in the room through the distributed tuplespace shared by all objects inside the room. The smart door also tries to include the arriving object into the distributed tuplespace shared by objects in the room. The corresponding code for the smart door in SICL is depicted in Fig. 11.

The technology break of using another technology for device discovery is of course an overhead because it requires that every smart object is additionally equipped with an RFID tag, but it is faster than conventional Bluetooth device discovery. It is also possible to combine this RFID-based approach with the conventional device discovery mechanism. Thereby, discovery results are enriched with context information by not only identifying the objects in range, but also those that share a certain symbolic location.

6.3 Context-Aware Topology Construction

As already discussed in section 5, it is a goal of the platform for inter-object collaboration to group smart objects according to their current context. Thereby, a network topology is created such that nodes that are likely to cooperate are grouped on one channel (in Bluetooth, this corresponds to grouping nodes in different piconets). Because 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 is better suited for characterizing the situation of a nearby person. In contrast, collaboration across room borders is less likely because sensor readings from another room are often less accurate. The context-aware service for topology construction presented in this section therefore groups nodes in clusters that share the same symbolic location. The goal is to show how the presented platform components support the implementation of such a service.

At first, the communication service assumes that relevant objects participate in one distributed tuplespace. The location of smart objects is then determined by audio input from a small microphone attached to smart objects. The microphone is read out once every minute on every object in the tuplespace. Thereby, a sampling rate of about 5 kHz for the duration of one second was used (cf. Fig. 12). As the access to the microphone has to be synchronized – i.e., it has to take place at the same time – a node triggers the sampling by writing a special tuple in the tuplespace. As a result, the microphone is then simultaneously read out at every node in the tuplespace.

The corresponding results are again made available to all participating nodes via the infrastructure for inter-object collaboration. Because transmitting the whole data stream is impossible, a small number of features are extracted from the audio stream that are embedded into a tuple and written into the space.

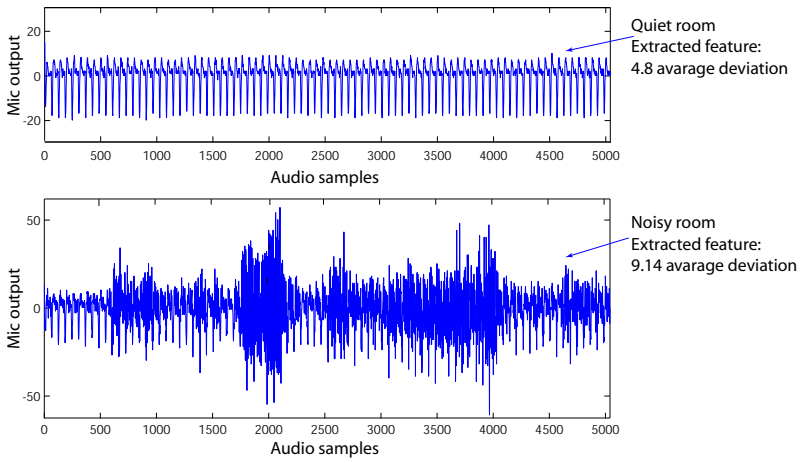

Fig. 12. Audio output of a microphone sensor and the extracted features shared via the infrastructure for inter-object collaboration.

In our implementation, we simply use the average volume as the main feature. Because this feature provides only limited information and synchronization problems can lead to inaccurate information, one sensor reading is not sufficient to really deduce which objects are in the same room. Instead, only after several consecutive sensor readings is it possible to draw a conclusion about the location of a node. The cluster head (in Bluetooth the master node) accesses this sort of history information by querying the distributed tuplespace and, if necessary, issues a command for organizing nodes into different clusters.

7 Conclusion

In this paper, we presented a context-aware communication platform for resource-restricted, wirelessly-communicating smart objects. The described platform consists of three parts: (1) a high-level description language, called SICL, for the development of context-aware communication services, (2) a tuple-based communication abstraction that hides low-level communication issues from higher layers, and (3) examples of context-aware communication services.

The description language SICL helps application programmers to clearly separate the main aspects of a context-aware application: basic and adaptive application behavior, context-recognition, and sensor access. So far, our experiences with the language are very promising, and the examples given in this paper show that it is practical for the development of context-aware communication services. However, the main advantage of SICL is that it relieves programmers of implementing the low-level communication issues necessary to fuse sensory data. Instead, it provides a simple tuple-based approach for data handling. This approach is realized by a distributed tuplespace that groups collaborating objects

based on their current context. The distributed tuplespace is also the main platform for inter-object cooperation and hides low-level communication issues from higher layers. According to the measurements presented in Sect. 5, it supports efficient collaboration among autonomous smart objects.

The concepts described in this paper have been implemented on a resource-restricted device platform, which has been used in several applications to augment everyday objects with computation. Experiments have shown that our implementation is practical and works well in the envisioned settings.

References

1. 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.
2. M. Beigl, H.W. Gellersen, and A. Schmidt. MediaCups: Experience with Design and Use of Computer-Augmented Everyday Objects. *Computer Networks, Special Issue on Pervasive Computing*, 25(4):401–409, March 2001.
3. 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)*, Seattle, USA, October 2003.
4. 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.
5. B. Johanson and A. Fox. Tuplespaces as Coordination Infrastructure for Interactive Workspaces. In *UbiTools '01 Workshop at Ubicomp 2001*, Atlanta, USA, 2001.
6. R. H. Katz. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1(1):6–17, 1994.
7. 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.
8. F. Siegemund. Kontextbasierte Bluetooth-Scatternetz-Formierung in ubiquitaeren Systemen. In *Michael Weber; Frank Kargl (Eds): Proc. First German Workshop on Mobile Ad Hoc Networks*, pages 79–90, Ulm, Germany, March 2002.
9. 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.
10. F. Siegemund and M. Rohs. Rendezvous layer protocols for Bluetooth-enabled smart devices. *Personal Ubiquitous Computing*, 2003(7):91–101, 2003.
11. 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.
12. M. Weiser and J. S. Brown. The Coming Age of Calm Technology, October 1996.
13. Berkeley Motes. www.xbow.com/Products/Wireless_Sensor_Networks.htm.
14. The Cooltown project. www.cooltown.com.
15. The Smart-Its Project: Unobtrusive, deeply interconnected smart devices. www.smart-its.org.