Wireless Networks 10, 689–700, 2004

Smart Identification Frameworks for Ubiquitous Computing Applications

KAY RÖMER *, THOMAS SCHOCH and FRIEDEMANN MATTERN
Institute for Pervasive Computing, Department of Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland

THOMAS DÜBENDORFER
Computer Engineering and Networks Lab, Department of Electrical Engineering, ETH Zurich, CH-8092 Zurich, Switzerland

Abstract. We present our results of the conceptual design and the implementation of ubiquitous computing applications using smart identification technologies. First, we describe such technologies and their potential application areas, then give an overview of some of the applications we have developed. Based on the experience we have gained from developing these systems, we point out design concepts that we have found useful for structuring and implementing such applications. Building upon these concepts, we have created two frameworks based on Jini (i.e., distributed Java objects) and Web Services to support the development of ubiquitous computing applications that make use of smart identification technology. We describe our prototype frameworks, discuss the underlying concepts and present some lessons learned.

Keywords: ubiquitous computing, RFID tags, virtual counterparts, Jini, Web Services

1. Introduction

Object tagging is an enabling concept for many interesting ubiquitous computing ("ubicomp") applications [19]. By attaching small electronic tags to physical objects, these objects can be automatically identified and located when brought into the vicinity of a tag detection system. Our goal is to support the development of applications that make use of smart identification technology by providing suitable abstractions and concepts and by incorporating these concepts into a framework. Since identification of real-world objects is the prerequisite for "smart" behavior, the framework should also support basic functionality for smart objects such as associating specific information and functionality with objects and providing an artifact memory. Furthermore, it should support event propagation, location management, and some other basic services for smart objects.

One example of a promising object tagging technology is passive radio frequency identification (RFID), where tags do not need their own power source and cost only a few tens of cents. State-of-the-art RFID systems such as the Phillips Icode system [21] allow the simultaneous detection of a few hundred tags within a space of up to one cubic meter. Typically, such tags not only hold a unique ID, but also provide a small amount of non-volatile read/write memory of up to about 100 bytes.

Besides passive RFID systems, other identification systems also exist. Bar codes are a classical technology for tagging physical objects, but they need line-of-sight to the reader and have other drawbacks that make them less attractive for ubicomp applications. In contrast to passive RFID systems, active RFID systems have built-in batteries enabling them to transmit their data over distances of up to 100 m. Disadvantages are their larger size and the higher price compared with passive systems. In the future, we also envisage small modules based on RF technologies similar to Bluetooth, WLAN or UMTS for tagging physical objects. Their main advantage is that they can cover a larger area and provide additional functionality such as transmitting sensor values. Currently, however, they have disadvantages with respect to size, price, and energy consumption that are similar to those of active RFID systems.

Despite their simplicity and current limitations, such passive RFID-based identification systems enable the implementation of a wide range of novel ubicomp applications by bridging the gap between the physical world (i.e., tagged real-world objects) and the virtual world (i.e., application software or service infrastructure). One example is tagged products ("smart products") that make new services and new cost-saving business processes possible. They bring benefits in the areas of source verification, counterfeit protection, one-to-one marketing, maintenance and repair, theft and shrinkage, recall actions, safety and liability, disposal and recycling as well as mass customizing [3]. Smart objects thus lead to more effective supply chain management systems, product life cycle management processes, and customer relationship management processes in the consumer goods industry [7]. However, the use of novel identification technologies is not limited to these classical business processes – many new and innovative applications are possible when real-world objects become "smart" by having information attached to them and being directly associated to backend IT systems or linked to services on the Internet.

Over the last three years we have developed a number of identification-based applications in areas such as smart games, home automation, and office automation. These appli-

* Corresponding author.
E-mail: roemer@inf.ethz.ch

cations are typically based on non-trivial interactions between multiple tagged objects. We have found that existing ubicomp infrastructures such as Savant [12], Cooltown [8], one.world [5], Gaia OS [13], and Stanford Interactive Workspaces [6] do not provide appropriate application level frameworks to substantially support the implementation of our applications. Although these infrastructures provide useful programming primitives, there is quite a large gap between these primitives and the necessary functionality of ubicomp applications based on the smart identification technologies that we have in mind.

In order to better understand the requirements of smart identification-based ubicomp applications and to proceed towards an application model, we first implemented from scratch a set of different prototype applications as presented in section 2. The only piece of software they had in common was the driver software for the RFID system. Based on our experience with these applications, we identified a number of tasks common to this type of application, which led to an application model and the design of concepts that we found useful for structuring and implementing applications using tagged physical objects. Based on those mechanisms, we then designed and implemented two application-level frameworks to support the development of tag-based ubicomp applications. The implementation of two different application-level frameworks enabled us to evaluate slightly different design decisions, and to compare implementations based on different programming platforms such as Sun's Jini, on the one hand, and Microsoft's .Net Web Services, on the other hand.

In the next section we first present a short overview of some of the applications we have developed. They will serve as a basis for identifying the general design concepts that we present in section 3. These concepts should form the basis of a generic ubicomp framework. After a description of our two prototype frameworks in section 4, we compare and evaluate some of the underlying ideas and draw conclusions for a more elaborate implementation of our concepts in section 5. We conclude by mentioning related work and giving a short outlook. The focus of the paper lies in concepts and suitable application frameworks, since these should serve as a basis for future systems of cooperating smart real-world objects.

2. Selected ubicomp applications

We outline below the type of applications we intend to support with our framework by sketching some of the prototypical smart identification-based ubicomp applications we have developed over recent years. Note that all the applications are based on multiple interacting tagged physical objects.

Smart Tool Box. Tools are equipped with RFID tags, and the tool box contains a mobile RFID system (including a tag reader antenna integrated into the tool box) [9]. The tool box issues a warning for safety reasons if a worker attempts to leave the building site (or a sensitive maintenance area such as an airplane) while any tools are missing from his or her box. The box also monitors how often and for how long tools

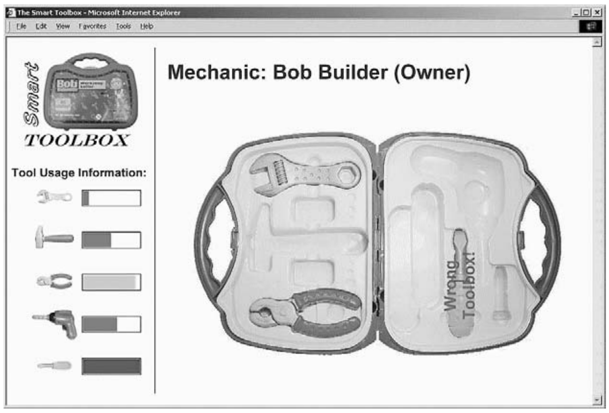

Figure 1. Screenshot of the Smart Tool Box application.

have been in use. Based on this information, tools can be replaced before they wear out. Additionally, the tool owner can charge for tool rental based on actual tool usage. Figure 1 shows a screenshot of this application.

Smart Supply Chain. Smart identification technology can significantly improve the efficiency of supply chains and the internal logistics processes of companies [7]. In such scenarios, the automatic identification and localization of goods at instance level can help to prevent faulty deliveries and speed up the whole business process. In our demo application, we simulate a small supply chain. It consists of two companies that bottle mineral water, one retail store, two freight companies, and one mineral water wholesaler. The retail store can send orders to the wholesaler, and the wholesaler can send orders to its two bottlers. The contractor of an order is responsible for having a freight company deliver the goods to the orderer. Every bottle of mineral water, the box containing the bottles, and the container for the boxes are tagged. RFID readers are installed at nine focal points along the supply chain to check whether the correct quantity of goods and the correct product instances have passed a particular focal point. If something goes wrong, a warning message is issued to the warehouse management system, which can decide on further action. Besides the tracking of goods, the bottles also monitor the temperature at every location and issue an alarm message if the current temperature exceeds a predefined temperature range. In addition, locations and objects can be queried for statistical information as a basis for future optimization of the whole supply chain.

RFID Chef. In this application [10], grocery items are equipped with RFID tags (instead of the bar codes that are commonly used today). When placed on a kitchen counter with an integrated RFID reader, a nearby display suggests dishes that could be prepared with the grocery items available, or shows missing ingredients. The suggested dishes not only depend on the available ingredients, but also on the preferences of the cook, who might, for example, prefer vegetarian or Asian dishes. To implement this functionality, the cook is identified by an RFID tag with the form factor of a credit card, carried in his or her wallet.

Smart Playing Cards. Ordinary playing cards are equipped with RFID tags. An RFID antenna mounted beneath a table monitors the players' game moves. A nearby display shows the score and the winner, and it raises a cheat alarm if any of the players do not follow suit. It also gives hints to beginners by assessing the players' moves and sending the hints to the player's PDA. This is implemented by having each card remember the context in which it has been played and whether the trick in question was won or lost. [14] contains a detailed description of the system.

3. Design concepts

The above-mentioned applications were initially developed from scratch. From these initial practical experiences, we identified common issues concerning the applications and came up with some general design concepts. In the following, we introduce the abstractions and design concepts we found. The subsequent section then shows how some of these abstractions and design concepts were incorporated into our application level frameworks.

Location. The notion of location is a central concept for most of the applications. In general, location can be based on geographic information such as coordinates, or on more abstract symbolic information, such as room numbers. A tagging system can provide both kinds of information. If the geographic position of the tag reader is known, the location of the tagged objects can be estimated. This information is useful in the Smart Supply Chain application, for example, where the distance between two distribution centers is relevant for the transportation of goods. The symbolic location information is normally determined by the tag reader and its detection range. In the Smart Tool Box application, all the tools within the range of the tool box antenna are supposed to belong in the same tool box.

Neighborhood. We use symbolic location information to explicitly support the concept of neighborhood. As in the Smart Tool Box example, "cooperating" physical objects are often collocated. Thus, the neighborhood concept is a relation between objects that are close to each other, making them potential candidates for collaboration. Note that we advocate a symbolic meaning of closeness that might differ from the Euclidean distance – two objects in different corners of a room might be closer to each other in a symbolic sense than two objects in two different rooms separated by a wall.

Location management. The management of locations refers to two similar but different issues. On the one hand, physical objects can contain other physical objects (e.g., a box that contains bottles). On the other hand, symbolic locations are usually ordered in a hierarchical way (e.g., a room is part of a building). Both concepts can be combined (e.g., a bottle is in a box, the box is located in a particular warehouse).

Location management should also consider two other aspects. One refers to the dynamic behavior of the containment relationship as in the Smart Tool Box example where tools are frequently put into and taken out of the tool box. The other aspect refers to the evolution of location hierarchies over time. The warehouse in the Smart Supply Chain example may be reorganized so that the location hierarchy needs to be adapted.

Time. Some of the applications require a notion of time. The Smart Tool Box, for example, has to determine the amount of real time that has elapsed between removing a tool from the box and replacing it. The Smart Playing Cards application knows which player played which card by means of the temporal order of the cards played. In general, there is a need to time-stamp such events. In the case of multiple tag readers, the time stamps of events originating from different readers should be comparable, even if some of the readers have been offline during event generation.

Composition. Physical objects are often an aggregation of other physical objects (e.g., a truck that transports bottles consists of thousands of different parts which might all be tagged). Many applications are only interested in manipulating a composite object in order to perform a certain manipulation on all the objects contained within that composite object (e.g., it is highly inefficient to communicate with all tagged parts of a truck if the new location of the whole truck needs to be set). In order to support such situations, it is necessary to explicitly model "part of" relationships between objects. This relationship can also be used to inherit properties. For example, it is not necessary that each part of a truck stores the same location information. If a part needs to know its location, it can ask its parent node in the hierarchy.

Note that composition is different from the neighborhood concept since neighboring objects do not necessarily belong to the same composite object. This concept also differs from the containment relationship. The containment relationship has to consider dynamic aspects in terms of the insertion and removal of objects, whereas the composition concept is more static. Objects in such a relationship depend on each other and cannot easily be inserted or removed without changing the nature or functionality of the objects (e.g., we can take objects out of a cupboard without changing the properties of the cupboard, but if we take the door off the cupboard, the cupboard becomes a shelf).

Linkage of the physical and virtual world. In order to enable a software application to react to actions in the physical world, a link has to be established between tagged physical objects in the real world and the application. Since RFID systems detect the presence and absence of tags in a certain physical space, this link can be established by notifying the applications of tags entering and leaving the space. A natural way to model these notifications is by means of an event notification system. The system has to support at least two basic events, `enter(X)` and `leave(X)`, which are sent to the application when a tag with identity X enters and leaves the detection range of the detection system, respectively. Additionally, applications need a way of expressing their interest in a subset of all possible tags, since a single RFID reader might be used simultaneously by multiple applications. Note that the tag detection system and the application may run on different systems and platforms, as, for example, in the Smart Tool Box application, which consists of a mobile tag detection system in the tool box cooperating with a fixed system located in the workshop, which runs the backend part of the application.

Although from an abstract point of view the tag detection system detects entering and leaving tags, matters are complicated by the actual low-level interface provided by the tag detection system and certain application requirements. The Icode RFID system [21], for example, periodically scans (typically at sub-second intervals) for present tags by sending a short RF pulse and waiting for answers from the tags. When receiving the pulse, a tag waits a random number of discrete time slots before answering in order to avoid time-consuming collisions with other tags transmitting concurrently. The maximum number of time slots N which a tag may wait before answering influences both the time needed for a single scan and the expected number of collisions. A small N results in fast scans (down to 60 ms according to [18]) but many collisions, whereas a large N results in slow scans (more than one second) but few collisions. The best value for N depends on the actual number of tags present. Since this number is typically unknown, non-trivial algorithms are needed to achieve good detection performance [18].

This kind of low-level interface has several implications. Firstly, applications are typically only interested in changes in the detected set of tags, that is, they want to receive enter and leave event notifications. So an appropriate software component has to convert scan results to event notifications. However, this component's task is non-trivial, since the scan results are typically imperfect due to tag collisions, that is, not all tags are detected in every scan. This can result in event flickering – the rapid generation of alternating leave and enter events for a tag that is in fact present all the time. Filters that cancel out spurious leave/enter events are required in the event of such imperfect tag detection.

Secondly, many applications require that objects be detected as quickly as possible. This is necessary if tags stay in the detection range for only a fairly short period of time. Even if the tags stay long enough, long delays in tag detection can cause problems with human–computer interaction. The Smart Playing Cards application exemplifies this, because the user expects an immediate reaction from the system when placing a card on the table.

History. Some applications not only react immediately to tagged objects entering and leaving the reading range, but subsequently also query objects on their history. Consider the Smart Tool Box example, where tools can be queried regarding how long they were used in which tool box on which building site. Therefore, a generic mechanism for logging and querying the history of physical objects would seem appropriate.

Context. Typically the application's action when a tag enters or leaves the reader's range depends not only on the identity of the tag, but also on the context such as the earlier presence or absence of other tags. Consider, for example, the RFID Chef application: the dishes that have to be displayed when a new grocery item is placed on the kitchen counter not only depend on the grocery item itself, but also on the cook. In the Smart Playing Cards application, the action taken when a playing card enters or leaves the antenna's range depends on the other playing cards currently lying on the table.

Often applications are only interested in events within a certain context. Consider again the Smart Playing Cards example where, for a game like whist, the application only wants to be informed when the last of four players has played his or her card in the trick. Such a selection of events can be performed at several levels, for example, in the application. However, the scalability and performance of a system can be increased by performing this selection as close as possible to the source of events. This, however, requires a way of expressing the event contexts in which applications are interested.

State and behavior. Applications typically assign state and behavior to physical objects. In the Smart Tool Box application, for example, the state of a physical object (i.e., a tool) consists of its usage pattern.

The applications also differ in the way they assign behavior to physical objects. In the RFID Chef application, for example, all the grocery items have a "common" behavior – displaying a suitable list of dishes. In the Smart Tool Box application, however, physical objects have a more "individual" behavior – calculating tool usage, for example. Moreover, a single physical object can contribute to the behavior of more than one other physical object. In the Smart Playing Cards application, for example, a single card contributes to the "usage context" of all the other playing cards on the table. A flexible mechanism is therefore needed for assigning state and behavior to physical objects.

Virtual counterparts. Due to resource limitations, neither the physical object nor the tag is able to implement all of the above concepts. Therefore, a digital representation is needed – the virtual counterpart of a tagged object – that can adopt this role. An application does not directly interact with the objects themselves, but with their virtual counterparts. In the Smart Toolbox example, the tool usage pattern is stored in the virtual counterpart. The tag is only used as a link to its virtual counterpart.

Identification and address. As pointed out above, we use the tag attached to the object as a pointer to its virtual counterpart. This means that the tag must provide some information on how an application can access the virtual counterpart. To identify the corresponding counterpart, each counterpart requires a unique identifier. An application also has to locate the virtual counterpart, which may reside somewhere on the Internet. For this purpose, a structured addressing scheme and an underlying directory service is necessary.

The identification or the address of a counterpart can be stored on the tag. The minimum information that is needed is a unique tag ID, which can then be mapped to the identifier or the address of the counterpart by an appropriate service in the infrastructure.

Life-cycle management. Life-cycle management deals with the instantiation, migration, and destruction of virtual counterparts. After a tag has been attached to a physical object, a virtual counterpart has to be created. After a tagged object has been destroyed, its virtual counterpart might also be destroyed to save resources. However, destruction is optional, since the virtual counterpart may exist "forever". For performance reasons, a virtual counterpart might also migrate to a place where communication with its tagged object is more efficient.

Communication infrastructure. All the applications we have developed so far make use of a communication infrastructure to access background services, such as the virtual counterpart of an object or an object history storage service. In environments or in scenarios where a wired Internet infrastructure is not present, we assume a wireless connection, such as IP over Bluetooth, WLAN, or UMTS. However, there may not always be global connectivity, as in the case of the Smart Tool Box application. The tool box contains a mobile RFID system and an associated computing system, which together are able to operate offline. The tool box is only connected to the background communication infrastructure when it is returned to the workshop. Such disconnected operations should also be supported by a general application framework for RFID-based applications.

4. Framework prototype implementations

In order to evaluate the concepts described in section 3, we implemented two prototype systems that build on these concepts. One is based on Jini (i.e., distributed Java objects), while the other uses Web Services as the underlying platform. By using these prototype systems to (re-)implement tag-based ubicomp applications, we wanted to gain experience that would be useful for a more elaborate future implementation of a general platform for smart identification-based applications.

4.1. Jini approach

For our first framework, we implemented the concepts outlined in section 3 in a Jini-based [20] infrastructure for virtual counterparts (VCs). Figure 2 shows an overview of the system architecture. RFID systems are connected to event drivers (EDs) that generate enter and leave events from periodical tag scans. The EDs act as producers for the virtual counterpart event service (VCES). The VCES delivers events to the virtual counterpart manager (VCM), and to specific counterparts. The VCM acts as an execution environment for counterparts. Upon the first sighting of a tagged object or location, it consults the virtual counterpart repository (VCR) to obtain counterpart executables for the tag or the specific location. Counterparts register with the look-up service (LUS) so that cooperating counterparts can find each other. The artifact memory (AM) acts as a place for persistently storing and retrieving counterpart state and event histories. Small amounts of state can also be stored in the tag memory by sending appropriate store events to the VCES.

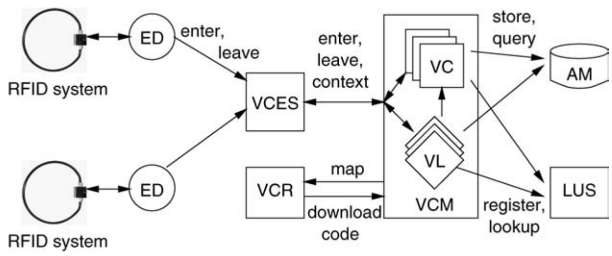

Figure 2. Infrastructure overview of the Jini approach.

Event driver. The event driver maps the output of the RFID reader to enter and leave events. As mentioned in section 3, typical RFID readers perform scans and return a possibly incomplete list of currently present tags. By calculating the difference between successive detection rounds, a list of entering tags and a list of leaving tags is determined. By removing from these lists tags that leave and reenter in rapid succession, we can avoid the event flickering mentioned above. From the resulting tag lists, enter and leave event notifications are generated. Both enter and leave events contain a tag ID, a location ID, and a time stamp.

Our current event driver implementation uses a so-called RFID framework [4], which provides an abstract tag reader interface and already implements the mechanisms outlined above. Additionally, this framework supports a wide variety of RFID hardware.

Virtual counterpart event service. Event producers and consumers advertise and subscribe to the VCES by specifying the types of events they want to generate or receive. Based on this information, the VCES forwards events to interested subscribers only. The VCES can tell producers not to produce events if nobody is interested in them.

Subscriptions can optionally contain a rule for specifying context events. Such a rule consists of event declarations and an executable. The program consists of a list of condition–action specifications. Each condition specifies an event pattern using a composite event language similar to the Cambridge Composite Event Language [11]. The action part emits one or more events based on the parameters of the matched event pattern.

Virtual counterparts. Counterparts are digital representations of real-world objects. We differentiate between two classes of counterparts: counterparts that represent physical objects (so-called Virtual Counterparts or VCs) and counterparts that represent physical locations (so-called Virtual Loca-

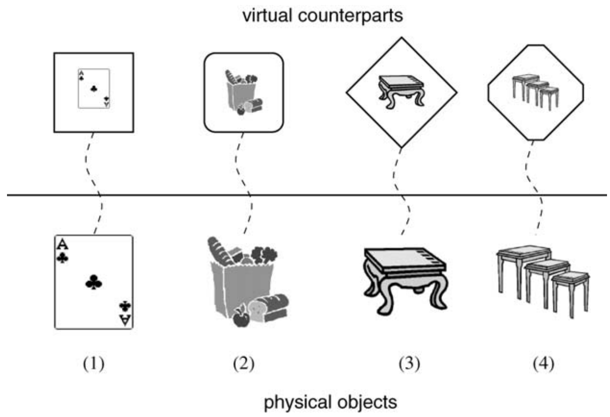

Figure 3. Virtual counterparts: (1) VC, (2) VMC, (3) VL, (4) VML.

tions or VLs). Mainly for performance reasons we have introduced so-called Meta Counterparts, which represent a whole set of physical objects (so-called Virtual Meta Counterparts or VMCs) or physical locations (so-called Virtual Meta Locations or VMLs). Since a meta counterpart manages a whole set of physical entities, the resource overhead per physical entity is much lower when using a meta-counterpart than when using lots of ordinary counterparts. These concepts are also illustrated in figure 3.

A VC is implemented as a Jini service with its own execution thread. A VC can communicate with other virtual counterparts by sending events or by invoking remote methods using Java RMI. The only difference between a VC and a VMC is that there is a one-to-one mapping between tag IDs and VCs, whereas there is a many-to-one mapping of tag IDs to VMCs. A VL is the digital representation of a location that is monitored by an RFID reader. The implementation is the same as for the VC except for the fact that the VL also maintains a list of VCs currently present at that location. Additionally, a VL forwards all received events to the VCs in that list. Similar to VMCs, VMLs represent a set of locations (or RFID readers), such as a set of shelves in a retail store.

Virtual counterpart manager. A VCM acts as an execution environment for the various types of virtual counterparts. It is also responsible for counterpart instantiation, migration, and destruction. For this purpose, the VCM monitors tagged objects by subscribing to enter and leave events.

If the VCM receives an enter event, it first consults the look-up service for matching counterpart instances. If no counterpart exists, the VCM consults the counterpart repository, which maps tag and location IDs to URLs. The URLs point to Java archive (JAR) files which contain code, resources, and arbitrary additional data for the respective virtual counterparts. The VCM downloads this code, executes it in a separate thread, and registers the counterpart with the look-up service. If on the other hand the look-up service already contains matching counterpart instances executing in a different VCM instance, the VCM asks the counterpart to migrate to the new location. However, the counterpart may choose to disregard this request. If the VCM receives a leave event, it asks the respective counterpart to clean up and exit. As with migration, the counterpart may choose to disregard this request.

Once a counterpart is up and running, it can subscribe to events, program the VCES for context events, use the LUS to look-up cooperating counterparts, and store and retrieve state information using the artifact memory. Counterparts are Java objects that provide an event API and a set of interface methods to the VCM. Counterparts cooperate by using events or Java RMI.

Note that it is possible to implement abstract virtual counterparts that have no physical equivalent by selecting an unused tag ID and manually sending enter/leave events with this ID to the VCM.

Virtual counterpart repository. The VCR consists of two components – a mapping facility that maps tag and location IDs to URLs, and an HTTP server for downloading the counterpart executables. By mapping multiple IDs to the same URL, we can implement meta-counterparts (or meta-locations) that correspond to multiple physical objects (or locations). Managing a whole set of similar objects (such as playing cards) by a single meta-counterpart is more efficient than having a distributed implementation with many communicating counterparts.

Look-up service. The LUS is somewhat similar to the VCR in that it maps location and tag IDs to virtual counterparts. However, in contrast to the VCR it returns pointers to executing counterpart objects, whereas the VCR returns pointers to static Java code which is used to instantiate the virtual counterparts. Again, meta-counterparts (locations) are implemented by mapping multiple IDs to the same counterpart (location). Normally, the LUS and the other Jini infrastructure components are totally transparent to the application developers since these components are only used internally by our framework to manage the registration of new VCs. However, it is possible for the application developers to contact the LUS and query it for VCs, infrastructure components such as the VCM, or 3rd party services.

Artifact memory. The AM stores state information in the form of attribute/value pairs and event histories. It is implemented as an abstract virtual counterpart. Other virtual counterparts can send predefined events (store state, retrieve state, store event, query events) to the AM. The query event can be used to issue queries to the AM regarding multiple events, such as "which objects were at location X at time T". The AM internally uses JDBC to open a connection to an SQL relational database. The AM creates one table for persistent state and one table for each event type in the database. The persistent state table has two columns; an attribute column and a value column. The table for a particular event type has one column for each parameter of this event type. The entry event table, for example, has three columns for its three attributes (tagID, locationID, time stamp). The AM query language is plain SQL, which is passed through to the database unmodified. However, to simplify frequently used requests, some powerful new query commands have been added:

- find(TAG, TIME): location of TAG at TIME;
- with(TAG, TIME): returns the set of tags at the same location as TAG at TIME;
- look(LOC, TIME): set of tags at location LOC at TIME;
- history(TAG): list of recent locations visited by TAG.

More information on the concepts and implementation of the Jini approach can be found in [2].

4.2. Web Services approach

Another approach to implementing the concepts described in section 3 is the use of Web Services. Web Services seem to be appropriate for several reasons. Firstly, the client/server paradigm is useful for modeling virtual counterparts – on the one hand, the virtual counterpart can provide its functionality as a service, and, on the other hand, tag-based applications can act as clients. Secondly, Web Services also provide a service description and discovery framework, which can be used to describe and locate virtual counterparts. Thirdly, Web Services build on open standards such as the Simple Object Access Protocol (SOAP), making them universally applicable. Fourthly, the framework can then easily communicate with third-party Web Services on the Internet.

Figure 4 shows the main components of the infrastructure. The tag detection system scans for tagged objects within its reading range. If a tagged object is detected, the system reads a URI from the memory of the tag. The URI consists of the identifier and the DNS-like address of the virtual counterpart. This URI is used by the tag detection system to contact a hierarchy of Universal Service Discovery and Description Interface (UDDI) servers. These UDDI servers use the DNS-like address to retrieve the Web server on which the virtual counterpart is running as a regular Web Service. In the next step, the tag detection system sets the new location of the tagged object (i.e., the location determined by the tag reader) in its virtual counterpart. The virtual counterpart uses this location information to register itself with a hierarchy of location managers. Since all virtual counterparts have to register themselves with this hierarchy, a virtual counterpart can ask the hierarchy who its neighbors are.

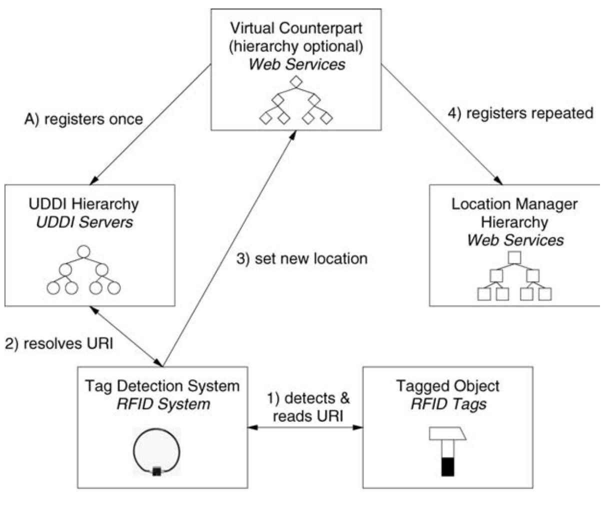

Figure 4. Infrastructure overview of the Web Service approach.

In the following, each system component mentioned above will be explained in more detail, with a focus on those issues that are different from our Jini-based application framework described in section 4.1.

Tagged object. The framework is designed to support various tagging technologies. Up to now, however, we have only implemented support for passive RFID technology. A tag only needs to store a Universal Resource Identifier (URI), which is used as a pointer to the virtual counterpart. As in our other approach, we have made use of the RFID framework [4]. Only a simple bridge had to be developed to couple the RFID framework with our system.

Tag detection system. The tag detection system is the actual component that bridges the gap between the physical world and the digital world. On the one hand, the system communicates with the tag, which resides on a real-world object. On the other hand, it also contacts the virtual counterpart of the tagged object to report the new location of the tagged object. A tag detection system is initialized with its physical or symbolic location and uses this information as the new location for all the tagged objects within its range. More sophisticated tag detection systems may calculate the position of a tagged object within the detection range more precisely (e.g., by measuring signal strength).

After a tagged object has entered the reading range of an antenna, the tag detection system reads the tag's memory, which contains the URI of the virtual counterpart. In order to set the new location of the tagged object in its virtual counterpart, the tag detection system first has to contact the UDDI hierarchy to resolve the URI. The UDDI hierarchy returns the Web server on which the virtual counterpart resides. Using this information, the tag detection system can set the new location in the virtual counterpart.

UDDI hierarchy. Within the Web Services framework, the UDDI defines how information about services can be stored and retrieved. A UDDI server acts as a database for service information, and implements the UDDI. The most important information that a UDDI server stores is the service description and the location of the service. Web Service Description Language (WSDL) is used to describe the service interface, so that clients can access the service. A Web Service that is up and running has to register itself on a UDDI server with its Web server address, so that clients can locate the service.

Originally, the UDDI servers were intended to establish a service cloud. This means that all UDDI servers belonging to a service cloud have to store information about all services worldwide, that is, a change on one UDDI server is propagated to all others within the cloud. However, we believe this will lead to scalability issues if a large number of objects are tagged. We have therefore extended the UDDI service cloud structure with a DNS-like partitioning that distributes all service information across the UDDI servers without redundancies (with the exception of some backup servers for reliability reasons).

The UDDI server generates a universal and unique identifier (UUID) if a service is registered for the first time. We also use this UUID as the unique name for a virtual counterpart. This UUID is a random number and has no structure. A structured identifier is necessary if we want to structure UDDI servers in a DNS-like style. We have therefore introduced addresses for the UDDI servers, also in a DNS-like style. The URI of a virtual counterpart consists of this DNS-like address and the UUID (e.g., uri:pharma.foopharma:40a96d21-ee00-0000-0080-e698e3243f5a). In this example "pharma.foopharma" denotes the UDDI server on which the virtual counterpart is registered. The DNS-like structure is used to find the UDDI server within the UDDI hierarchy. "40a96d21-ee00-0000-0080-e698e3243f5a" denotes the virtual counterpart. It is unique for each tagged object and independent of the UDDI server, allowing the virtual counterpart to migrate within the UDDI hierarchy. Each tag detection system possesses a UDDI client. This client uses the DNS-like address to find the appropriate UDDI server to retrieve the Web server on which the service (i.e., the virtual counterpart) is running.

Virtual counterpart. As mentioned above, every virtual counterpart is implemented as a Web Service that runs on a Web server somewhere on the Internet. The interface of such a virtual counterpart is different for different types of tagged objects. A minimal set of functions is common to all virtual counterparts and therefore supported by all counterpart implementations. Besides some auxiliary methods, a virtual counterpart provides methods to set and get the current location and retrieve the location history, as well as some methods for adding and removing parent and child nodes depending on its position in a composition tree (if this object is part of a composite object). All other methods that are specific to a tagged object have to extend this minimal interface.

Location manager. While the UDDI hierarchy tracks the whereabouts of virtual counterparts, the location manager hierarchy tracks the whereabouts of the tagged objects. Since every virtual counterpart has to register itself with the appropriate location manager for its tagged object, the location manager is able to determine the neighbors of a tagged object. Hence a virtual counterpart can ask the location manager for all other virtual counterparts of tagged objects that are close to its own tagged object.

Location information is modeled as coordinates. Besides the geographic information, the location information also contains hierarchically classified symbolic names. The location managers are arranged in a tree structure. The root location manager is responsible for the whole world. Child nodes constitute a partition of their parent node. When a virtual counterpart has to register itself with the root location manager, the root location manager delegates this registration to the node that covers the smallest space in which the tagged object is contained.

Besides implementing the neighborhood concept, the location manager also implements the containment concept. When an object may contain other objects (such as a warehouse that contains boxes of bottles), the location manager in charge of the warehouse space maintains a link to the virtual counterpart of the containing object.

5. Experiences with the frameworks

In this section we present our experiences with the two frameworks based on Jini and Web Services. Firstly, we describe the implementation of a complex application with both frameworks. Secondly, we present a performance analysis of various aspects of the two framework implementations. We conclude this section by discussing some of the design decisions.

5.1. Sample application

We used the prototype frameworks described in section 4 to implement a number of applications. Here, we want to describe our experiences with the implementation of the Smart Supply Chain mentioned earlier in the paper. We will first describe the implementation of this application using the Jini-based framework. This will then be followed by a discussion of the differences between that and an implementation using the framework based on Web Services.

Jini approach. You will remember from section 2 that our supply chain consists of bottlers, retail stores, wholesalers, and freight companies. Each supply chain element is equipped with one or more tag reading systems in order to monitor in- and outgoing tagged goods (i.e., bottles and boxes). Each such tag reading system is represented by a virtual location (VL). Additionally, there is a software component called the warehouse management system associated with each supply chain element. This component is closely linked to the VLs of the corresponding supply chain element, implements most of the application logic, and provides a graphical user interface to monitor and control various aspects of the supply chain.

Both the bottles and the boxes are individually tagged, such that their presence can be detected by one of the tag reading systems of the supply chain elements. Each bottle and each box is represented by a separate virtual counterpart (VC). Bottle VCs register for temperature events in order to check that storage temperatures are complied with. If bottles or boxes move along the supply chain, their VCs migrate to the VLs of the corresponding supply chain element.

The warehouse management system provides the following functionality (see also figure 5):

- It displays the VLs associated with the managed supply chain element. For each VL, a list of associated VCs of bottles and boxes is displayed. Additional detailed information can be displayed for each VC and the physical object associated with it.

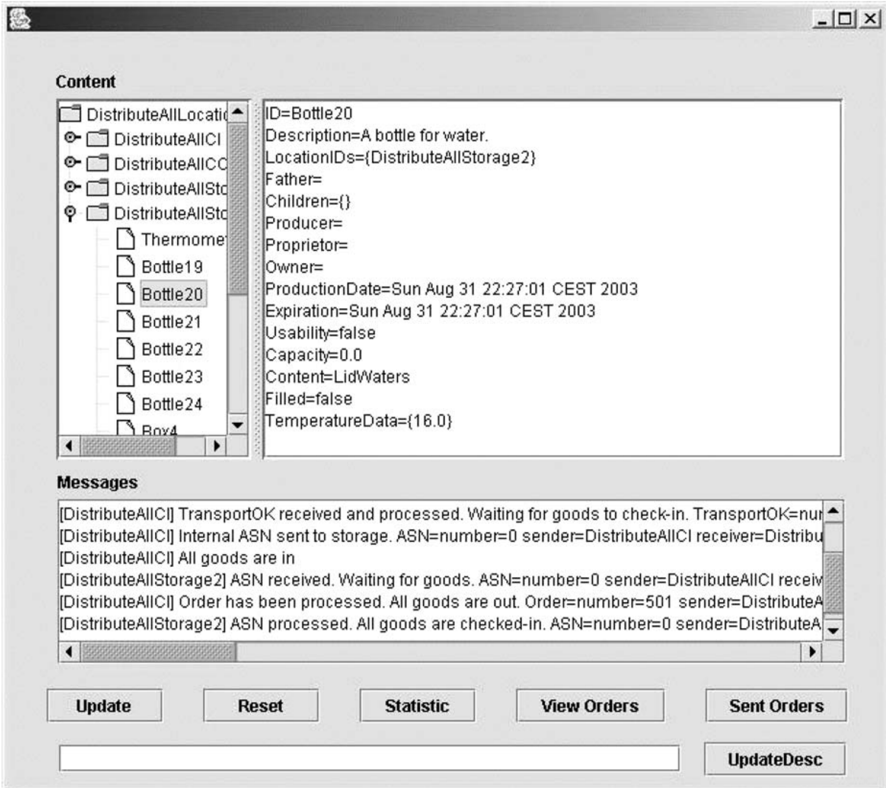

Figure 5. User interface of the warehouse management system.

- Both VLs and VCs can send status events (e.g., non-compliant temperature detected, incorrect delivery), which the management system subscribes to and displays.
- Orders can be sent to upstream elements in the supply chain.
- Both VLs and VCs can be queried on their transit times through the supply chain for statistical reporting purposes.

The last item in the above list is implemented using the artifact memory (AM). Each VL and VC stores entry events, exit events, and temperature events in the AM, and these can subsequently be retrieved for statistical analysis. Additionally, the AM is used to store the state information of the VCs in order to support their migration.

Overall, the framework provided useful abstractions for implementing this application in a structured way. Once the developer is familiar with these abstractions, the development effort is well below that for an implementation from scratch. However, we also encountered a number of problems during the development process, which we will now describe.

The implementation of a VC as a Jini service with its own execution thread provides a good decoupling of individual VCs, but suffers from a rather large resource requirement. Hence, overall resource consumption may be a problem where there are large quantities of goods (i.e., bottles and boxes). We tried to reduce resource consumption by implementing a box as a meta counterpart (VMC) that managed the bottles contained in the box. However, this turned out to be impossible due to the static mapping of tags to VMCs in the framework, which precluded moving a bottle from one box to another. This static mapping of tag IDs to counterparts also complicates the introduction of new goods, since this requires a new entry to be manually added to the mapping list.

The missing option of grouping VCs and VLs in a hierarchical way according to their location complicated the implementation of queries across sets of VCs (e.g., a query to obtain the number of goods on all shelves of a wholesaler). Such queries involved looking up all the affected counterparts and issuing the query individually to each counterpart. A mechanism to issue a query to a whole set of VCs as a single operation would be handy.

One further problem was caused by the implementation of the event notification system, which resulted in a significant programming overhead. Since the event arguments are implemented as Java Objects, the programmer has to provide marshalling code for them. Also, the receiver of an event has to check event arguments for type conformity as well as having to cast Java Objects to the expected argument types. Remote method invocation systems such as Java RMI do a much better job of supporting the programmer in this respect.

One potential problem relates to the lookup service, which we used as a repository for all VCs. The use of a single service instance might become a bottleneck if a large number of counterparts were used.

Web Services approach. Most of the problems encountered with the Jini-based framework can be resolved when using the framework based on Web Services. As with the first approach, bottles and boxes are implemented as virtual counterparts. However, the box is a composite counterpart that contains the counterparts of six bottles. Hence, all the bottles located in a box can be manipulated by manipulating their box. Also, the location hierarchy supports queries relating to all counterparts at a specified location (e.g., a query to obtain the quantity of goods on all the shelves of a wholesaler). Last but not least, the UDDI hierarchy supports a more flexible mapping of tag IDs to counterparts and removes the limitations of a single repository as used in the Jini framework.

What remains unsolved also with this second implementation are the performance and resource consumption issues. Each counterpart is implemented as a Web Service, which implies a significant resource overhead. Additionally, method invocation performance is rather poor, since this involves constructing and parsing complex SOAP messages. The following section contains a more detailed analysis of these performance overheads.

5.2. Performance

We conducted performance measurements for two concrete platforms: Sun's Jini implementation and Microsoft's .NET Web Services. The tests were performed on 451 MHz Intel Pentium III PCs with 256 MB of RAM running Windows XP, connected by a 100 Mbps Ethernet network. For the tests, we used a simple application that implements counterparts as described in section 4 using Jini and .NET, respectively. We examined the memory footprint of the two runtime environments and the memory footprint of each virtual counterpart in the test application. Additionally, we measured the amount of time required to perform a counterpart lookup followed by the invocation of a simple method on the counterpart. We performed 20 runs and calculated averages.

The tests show that Jini currently performs much better than .NET Web Services. The memory footprint of the Jini runtime environment is 9564 kB, whereas the .NET runtime consumes 17332 kB. Each additional virtual counterpart consumes at least 1.84 kB with Jini, and at least 1640.97 kB with the .NET implementation. Jini also performs better with respect to the execution time of method invocations. One service lookup and the invocation of a simple method take on average 198.8 ms with a deviation of 7.2 ms for Jini. The .NET Web Services needs 814.8 ms on average with a deviation of 121.8 ms. Note that this refers to a best case scenario, where the first UDDI server contains the registered service and only a single simple method is called.

5.3. Discussion

The two frameworks differ with respect to some architectural concepts. One difference is the usage of the tag ID. In the first framework, the tag ID is used by several background entities, whereas in the second framework, the tag is only used to establish the link between the physical world and the virtual world. The latter allows infrastructure services to be decoupled by hiding low-level details. The frameworks also differ in how they manage the addressing hierarchy, the structuring hierarchy of tagged objects, and the location hierarchy. The second framework makes these hierarchies and their underlying models explicit, whereas the first framework does not explicitly consider them, making it more difficult to use the concepts in an application. Another difference is the support of migration and history – the first framework provides dedicated entities that incorporate these concepts, whereas the second framework does not possess such entities. Also, the first framework introduces meta-counterparts and meta-locations, but these do not figure in the second framework. These concepts were intended to reduce resource consumption and improve performance. However, during our experiments it turned out that they are currently too inflexible to be of practical use in many applications. The challenge now consists of fixing these problems and combining the proven concepts of both prototypes into an encompassing framework. The experiments also revealed that current Web Service implementations are rather inefficient – at least for the type of applications we have in mind.

6. Related work

Several other trials exist that aim to provide support for applications based on smart objects. The work of the MIT Auto-ID Center [15] comes closest to our intention of providing a framework for smart objects. The goal of the Auto-ID center and its sponsoring companies is to replace the traditional bar code with passive RFID tags. For this, the whole spectrum of components for such a solution, ranging from low-level protocols for the communication between tag and reader to an XML-based language for exchanging information about products, is investigated. The middleware that controls the readers and processes the tag IDs is called Savant [12]. The Savants form a tree, with the edge Savants directly controlling the RFID readers and storing the tags IDs, and the internal savants aggregating the data received from their child nodes. Savants also provide a means of notifying external programs of tag information, and they run tasks that have to be registered with a Savant. One aspect that is lacking in this approach is the virtual counterpart that actively reacts to changes in the real-world – Savants only manage passive database entries.

Cooltown [8] and the associated CoolBase infrastructure aim to give people, places, and things a Web presence. This Web presence has a similar function to that of our virtual counterparts. The use of well-known Web technology is both an advantage and a disadvantage. On the one hand, this technology is proven and widely available, but on the other hand we think there is an important difference between Web applications and ubicomp applications. Due to its origin, the Web is document-centric. Although it has been augmented with ways of including dynamic distributed applications (e.g., SOAP), it still retains its inherent hypertext nature. On the other hand, ubicomp applications are more akin to dynamic distributed applications. The emerging XML-based Web infrastructure (i.e., Web Services) might support the needs of tag-based ubicomp applications in the future. However, our experience indicates that the performance of current Web Services implementations is not adequate to support such large-scale applications.

The Stanford Interactive Workspaces project [6] aims to provide a support infrastructure for interactive rooms equipped with large displays and other wireless devices for interaction. The main focus of the project, however, is to support user interaction and group work in augmented rooms. The i-Land and Roomware [16] projects have a similar focus. Although these infrastructures provide good and useful concepts for modeling and implementing ubiquitous computing applications, they focus mainly on HCI issues and how to support users with many portable and stationary electronic devices. In our systems, we focus on everyday items that do not have any additional electronics except an almost invisible tag. With such enhanced everyday items we want to enable new ubiquitous computing applications – where the items do not necessarily have to interact with users.

Projects such as Gaia [13], One.World [5], Microsoft Easy Living [1], and CORTEX [17] aim to develop an infrastructure to support augmented environments in a fairly broad sense. They provide basic abstractions and mechanisms for coping with the dynamics and device heterogeneity of pervasive computing environments. On top of these mechanisms they provide application models that are still rather generic. There is quite a large gap between the abstractions provided by these projects and frameworks such as our own, which support a rather specific application model (that of multi-object, tag-based applications in our case). Although those infrastructures try to achieve different goals, some of their underlying concepts and components are similar. Gaia, for example, provides a concept called *digital entity* which is similar to our virtual counterpart. Like many other systems, Gaia also uses events as a basic communication abstraction. However, Gaia is intended to support the rather broad application domain of so-called active spaces. In contrast, our frameworks are specifically tailored for tag-based applications. That is, our frameworks support a rather narrow application domain, but provide a number of specialized mechanisms to substantially support the development of applications based on smart identification technology.

7. Conclusion and outlook

Based on our experiences with several prototype applications, we came up with a set of basic functions and services, and an application model for smart identification-based ubicomp applications. We built two prototype frameworks based on different underlying platforms to support the development of such applications. Initial experience shows that application development and maintenance can be significantly simplified by using such application-level frameworks, which are tailored to the specific needs of tag-based applications. In the future we not only intend to support other tagging systems, but also sensing devices giving rise to another class of interesting applications.

Our two prototypical frameworks have covered various aspects, but only to a certain depth. In the future we want to investigate some concepts in more detail in order to come up with a single framework that is based on well-suited concepts and also possesses the necessary level of performance.

Acknowledgements

We would like to acknowledge Daniel Schädler for his work on the Web Services framework, Tobias Schwägli for his work on the performance test, Marc Langheinrich and Harald Vogt for their work on the RFID Chef, Matthias Lampe for his work on the Smart Tool Box, and Philip Graf, Martin Hinz, Svetlana Domnitcheva, and Vlad Coroama for their help with the Smart Playing Cards.

References

[1] B. Brummit, B. Meyers, J. Krumm, A. Kern and S. Shafer, Easy living: technologies for intelligent environments, in: *HUC 2000*, Bristol, UK (September 2000).

[2] T. Dübendorfer, An extensible infrastructure and a representation scheme for distributed smart proxies of real world objects, Master's Thesis, ETH Zurich (2001). Also available as Technical Report TR-359.

[3] E. Fleisch and M. Dierkes, Betriebswirtschaftliche Anwendungen des Ubiquitous Computing – Beispiele, Auswirkungen und Visionen, in: *Total Vernetzt*, ed. F. Mattern (Springer, Heidelberg, 2003).

[4] C. Floerkemeier, M. Lampe and T. Schoch, The Smart Box concept for ubiquitous computing environments, in: *Smart Objects Conference (sOc) 2003*, Grenoble, France (May 2003).

[5] R. Grimm, System support for pervasive applications, Ph.D. Thesis, University of Washington, Department of Computer Science and Engineering (December 2002).

[6] B. Johanson, A. Fox and T. Winograd, The interactive workspaces project: Experiences with ubiquitous computing rooms, IEEE Pervasive Computing 1(2) (2002) 71–78.

[7] A. Kambil and D. Brooks, Auto-ID across the value chain: from dramatic potential to greater efficiency and profit, Technical Report ACN-AUTOID-BC-001, MIT Auto-ID Center Cambridge (2002).

[8] T. Kindberg et al., People, places, things: Web presence for the real world, in: *WMCSA 2000*, Monterey, USA (December 2000).

[9] M. Lampe and M. Strassner, The potential of RFID for moveable asset management, in: *Workshop on Ubiquitous Commerce, Ubicomp'03*, Seattle, USA (October 2003).

[10] 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).

[11] G.J. Nelson, Context-aware and location systems, Ph.D. Thesis, University of Cambridge (1998).

[12] Oat Systems and MIT Auto-ID Center, The Savant, Technical Report MIT-AUTOID-TM-003, MIT Auto-ID Center (May 2002).

[13] M. Roman, C. Hess and R. Campbell, Gaia: An OO middleware infrastructure for ubiquitous computing environments, in: *ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS) 2002*, Malaga, Spain (June 2002).

[14] K. Römer and S. Domnitcheva, Smart Playing Cards – A ubiquitous computing game, Journal for Personal and Ubiquitous Computing 6(6) (2002) 371–378.

[15] S. Sarma, D. Brock and K. Ashton, The networked physical world – Proposals for engineering the next generation of computing, commerce and automatic identification, Technical Report MIT-AUTOID-WH-001, MIT Auto-ID Center (October 2000).

[16] P. Tandler, Software infrastructure for ubiquitous computing environments: supporting synchronous collaboration with heterogeneous devices, in: *Ubicomp 2001*, Atlanta, USA (September 2001).

[17] P. Verissimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday and J. Kaiser, CORTEX: Towards supporting autonomous and cooperating sentient entities, in: *European Wireless 2002*, Florence, Italy (February 2002).
[18] H. Vogt, Efficient object identification with passive RFID tags, in: *Pervasive 2002*, Zurich, Switzerland (August 2002) pp. 98–113.
[19] 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).
[20] Jini Architecture Specification, `www.sun.com/jini/specs/`
[21] The Philips i-code system, `www-us2.semiconductors.philips.com/identification/products/icode`

Kay Römer received his Masters degree with honor from the University of Frankfurt/Main, Germany in 1999. Since 1999, he is a research assistant and Ph.D. student at ETH Zurich (the Swiss Federal Institute of Technology), Switzerland. His research interests encompass sensor networks, software infrastructures for ubiquitous computing, and middleware for distributed systems.
E-mail: roemer@inf.ethz.ch

Thomas Schoch studied computer science and business administration at Darmstadt University of Technology, Germany from 1996 to 2000. In 2000 he wrote his Master Thesis at the International Computer Science Institute at Berkeley. Since 2001, he is a Research Assistant at the Distributed Systems Group at ETH Zurich. His research focus lies on aspects related to infrastructures for smart things. Currently, he finishes his Ph.D. Thesis and started to work at Intellion – The Smart Things Company.
E-mail: schoch@ethz.ch

Friedemann Mattern has been a Professor of Computer Science at ETH Zurich since 1999. He studied computer science and communications at the University of Bonn and received his Ph.D. from the University of Kaiserslautern in 1989. He served as Professor of Computer Science at the University of Saarbrucken from 1991–1994, and at Darmstadt University from 1994–1999. At ETH Zurich, Mattern founded the Institute for Pervasive Computing and heads the Computer Science Department's Distributed Systems Group. His research interests include concepts of distributed computing, ubiquitous computing, and infrastructure mechanisms for dynamic networking of small and mobile devices.
E-mail: mattern@inf.ethz.ch

Thomas Dübendorfer received a Master of computer science with first class honors from ETH Zurich, Switzerland in 2001. His Master Thesis in the field of distributed computing was awarded the ETH medal. In the same year he joined the Cooltown team at Hewlett-Packard Research labs in Palo Alto for a research internship. In late 2001 he started his Ph.D. at ETH. His main research interest is in the field of network security.
E-mail: duebendorfer@tik.ee.ethz.ch