Smart Identification Frameworks for Ubiquitous Computing Applications

Kay Römer, Thomas Schoch, Friedemann Mattern, Thomas Dübendorfer
Institute for Pervasive Computing, Department of Computer Science, ETH Zurich
{roemer|schoch|mattern}@inf.ethz.ch, duebendorfer@tik.ee.ethz.ch

Abstract

We present our results in the conceptual design and the implementation of ubiquitous computing applications using smart identification technologies. First, we describe such technologies and their potential application areas, followed by an overview of some applications we have developed. Based on the experiences we gained from the development of these systems, we point out design concepts that we find 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.

1 Introduction

Object tagging is an enabling concept for many interesting ubiquitous computing ("ubicomp") applications [17]. 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 technologies by providing suitable abstractions and concepts and by incorporating these concepts into a middleware 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 to 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 a few tens of cents only. State-of-the-art RFID systems such as the Phillips Icode system [18] allow the simultaneous detection of a few hundreds 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 up to about 100 bytes.

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

Despite their simplicity and current limitations, such smart 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). One example are tagged products ("smart products") which 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. Smart objects thus lead to new supply chain management systems, product life cycle management processes, and customer relationship management processes in the consumer goods industry. However, the use of novel identification technologies is not limited to these processes, but many new applications are possible when real-world objects become "smart" by attaching information to them and linking them directly to backend IT systems or services on the Internet.

During the last two years we have developed a number of smart identification-based applications in areas like smart games, home automation and office automation. These ap-

plications are often based on non-trivial interactions between multiple tagged objects. We found that existing ubicomp infrastructures such as Savant [10], Cooltown [6], one.world [4], Gaia OS [11], or Stanford Interactive Workspaces [5] 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 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 a set of different prototype applications as presented in Section 2 from scratch. 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 have then designed and implemented two application level frameworks to support the development of tag-based ubicomp applications. The realization of two different application level frameworks enables us to evaluate slightly different design decisions, and to compare implementations based on different programming platforms like 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 developed. They will serve as a basis for identifying general design concepts that we present in Section 3. These concepts should be incorporated by 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 on concepts and suitable application frameworks, since this should serve as a basis for future systems of cooperating smart real-world objects.

2 Selected Ubicomp Applications

In the sequel we outline 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 the recent years. Note that all 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) [3]. 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 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.

Smart Medicine This application helps to avoid trouble with medication by monitoring medicine from production to use [14]. For this, medicine bottles are equipped with RFID tags. The environmental temperature of the medicine is constantly monitored in order to avoid it going bad. Within the medicine cabinet, the bottle checks for other pharmaceuticals which are not compliant if taken together. A warning is issued upon detection of such dangerous situations.

Smart Agenda Agendas are equipped with RFID tags. If two or more people want to make an appointment, they place their agendas on the "appointment table", which is equipped with an RFID antenna. A nearby display shows possible dates that are compatible with the schedules of all the participants.

RFID Chef Grocery items are equipped with RFID tags (instead of the barcodes that are commonly used today). When placed on the kitchen counter with an integrated RFID reader, a nearby display suggests dishes that can 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 a form factor of a credit card, which he or she carries in his or her wallet. [8] contains a detailed description of the system.

Smart Playing Cards Ordinary playing cards are equipped with RFID tags. An RFID antenna mounted beneath a table monitors the game moves of the players. A nearby display shows the score, the winner, and a cheat alarm if one of the players does not follow suit, and gives hints to beginners by assessing the players' moves. This is implemented by having each card remember the context in which it has been used and whether the trick in question was won or lost. [12] contains a detailed description of the system.

3 Design Concepts

The above-mentioned applications were first developed from scratch. From these first practical experiences, we identified common issues of 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 for some of these abstractions and design concepts how they 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 Medicine example, where the distance between two distribution centers is relevant for the transportation of the medicine. The symbolic location information is normally determined by the tag reader and its detection range. In the Smart Tool Box application, all the tools in the range of the tool box antenna are supposed to belong to the same tool box.

Neighborhood We use the symbolic location information to introduce 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, what makes them potential candidates for a collaboration. Note that we use a symbolic meaning of closeness that might differ from the Euclidian 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. the medicine cabinet contains some medicine bottles). On the other hand, symbolic locations are normally ordered in a hierarchical way (e.g. a room is part of a building). Both concepts can be combined (e.g. the medicine bottle is in a case, the case is stored in shelf row Y, shelf row Y is located in warehouse A).

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 in and out of the tool box. The other aspect refers to the evolution of location hierarchies over time. The warehouse in the Smart Medicine 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 played cards. 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 Often physical objects are an aggregation of other physical objects, e.g. a truck which transports medicine bottles consists of thousands of different parts, which might all be tagged. Many applications are only interested in manipulating composite objects in order to perform a certain manipulation on all the objects contained in a 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 should be set). In order to support such situations, it is necessary to explicitly model "part of" relationships among 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. In case 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 insertion and removal of objects, whereas the composition concept is more static. Objects in such a relationship depend on each other and cannot easily inserted or removed without changing the nature or functionality of the objects (e.g., we can take out objects of a cupboard without changing the properties of the cupboard, if we demount the door of the cupboard, the cupboard becomes a shelf).

Linkage of the Physical and Virtual World In order to enable an 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 presence and absence of tags in a certain physical space, this link can be established by notifying the applications about tags entering and leaving this 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 ID 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 by multiple applications at the same time.

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 stationary system at the workshop.

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 [18], for example, periodically scans (typically at sub-second intervals) for present tags by send-

ing 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 sending 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 value results in fast scans (down to 60ms according to [16]) but many collisions, whereas a large N value results in slow scans (more than one second) but few collisions.

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

Secondly, many applications require that objects be detected as fast as possible. This is necessary if tags stay in the detection range only for a rather short 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 the latter, because the user expects an immediate reaction from the system on placing a card on the table. The optimum detection performance can be achieved by selecting the number of time-slots N to be slightly greater than the actual number M of tags in the range of the reader. However, M is typically unknown. Therefore, nontrivial algorithms are required for selecting N in order to read all present tags in a minimal amount of time [16].

History Some applications do not only react immediately to tagged objects entering and leaving the reading range, but objects are also queried about their history later on. 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 seems appropriate.

The minimal interface to access history information should contain the following methods:

- 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

Context Typically the application's action when a tag enters or leaves the reader's range not only depends 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 with a certain context. Consider the Smart Playing Cards example, where 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, 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 applications are interested in.

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

The applications also differ in the way they assign behavior to physical objects. In the RFID Chef application, for example, all the grocery items and the cook have a "common" behavior – the display of a list of dishes. In the Smart Tool Box application, 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 are able to implement all of the above concepts. Therefore, one needs a digital representation – the virtual counterpart – that can take over this role. An application does not directly interact with the tagged objects themselves, but with their virtual counterparts. In the Smart Agenda example, the actual agenda is stored in the virtual counterpart. The tag is only used as a link to its virtual counterpart. As depicted in Figure 1, virtual counterparts come in various flavors. A counterpart can be associated with a single (1) or a set of (2) physical objects. Moreover, it can represent a single (3) or multiple (4) physical locations.

Name and Address As pointed out above, we use the tag on the object as a pointer to its virtual counterpart. That

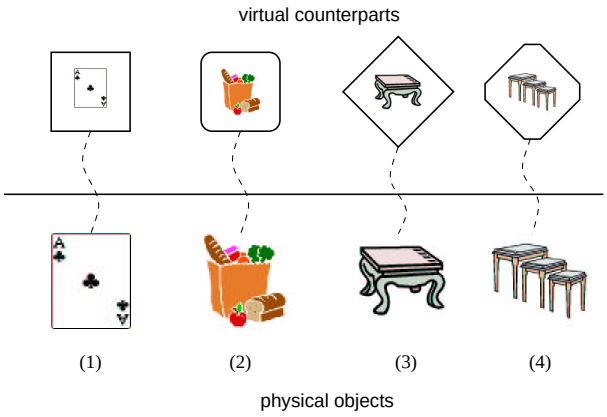

Figure 1. Virtual Counterparts

means that the tag must provide some information how an application can access the virtual counterpart. To identify the corresponding counterpart, a unique name is necessary for each counterpart. An application also has to locate the virtual counterpart that may reside somewhere on the Internet. For this purpose, a structured addressing scheme and an underlying directory service is necessary.

The name 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 name or the address of the counterpart by a 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 the physical object, the virtual counterpart has to be created. After a tagged object has been destroyed, the 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 migrate to a place where the 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, e.g. 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 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.

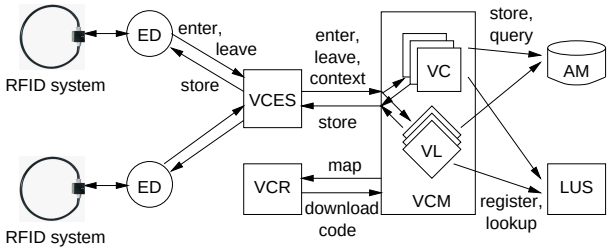

Figure 2. Infrastructure Overview of the Jini Approach

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), the other uses Web Services as the underlying platform. By using these prototype systems to (re-)implement tag-based ubicomp applications, we want to gain experiences that should be useful for a future and more elaborate implementation of a general platform for smart identification-based applications.

4.1 Jini Approach

For our first framework, we have implemented the concepts outlined in Section 3 in a Jini-based infrastructure for virtual counterparts. Figure 2 shows an overview of the system architecture. RFID systems are connected to event drivers (EDs), which generate enter and leave events from periodical tag scans. The EDs act as producers for the counterpart event service (VCES). The VCES delivers events to the counterpart manager (VCM), and specific counterparts. The VCM acts as an execution environment for counterparts. Upon the first sighting of a tagged object or location, it consults the counterpart repository (VCR) to obtain counterpart executables for the tag or 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.

Event Driver Using the RFID driver software, the ED periodically executes tag detection rounds. Each detection round consists of multiple tag scans with a carefully selected maximum time-slots parameter N, which is obtained by using a mechanism we developed in [16]. The latter provides a technique for estimating the actual number of tags present from a scan result, which consists of a list of detected tags as well as the number and type of collisions.

Thus, we start with a small value for N, estimate the number of tags N' from the scan result, and re-scan with N'. This process is repeated until all tags have been detected with a probability p. Note that even if p is very close to 1, due to the probabilistic nature of the anti-collision scheme there is a slight chance of one or more tags not being detected.

By calculating the difference between successive detection rounds, a list E of entering and a list L of leaving tags is determined. In order to avoid event flickering as mentioned in Section 3, we do not post events for tags in E and L to the VCES immediately. Instead, we keep a history of the scan results E_i and L_i at time t_i of the last second. In this window, we look for tags T contained both in L_i and in E_j with $j > i$. Such T are removed from both L_i and E_j. Now for each remaining tag in L_i (E_i) a leave (enter) event is posted to the VCES as soon as t_i is more than one second in the past.

Both enter and leave events contain a tag ID, location ID, and a time stamp. Enter events can carry additional data from the tag memory. The ED also subscribes to store events from the VCES, which contain a tag ID and data that is to be written to the memory of specific tags. Thus the combination of store/enter events can be used to store/retrieve state to/from the tag memory.

Virtual Counterpart Event Service 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 a program. 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 [9]. The action part emits one or more events based on the parameters of the matched event pattern.

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, or store and retrieve state 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 which 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 a meta-counterpart (or meta-location), which corresponds 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. Again, meta-counterparts (locations) are implemented by mapping multiple IDs to the same counterpart (location).

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 enter event table, for example, has four columns, since enter events have four attributes (tag ID, location ID, time stamp, tag memory contents). The AM query language is plain SQL, which is passed through to the database unmodified. However, to simplify often used requests, some powerful new query commands were added (such as "Where is object X?" or "Who is at location L?").

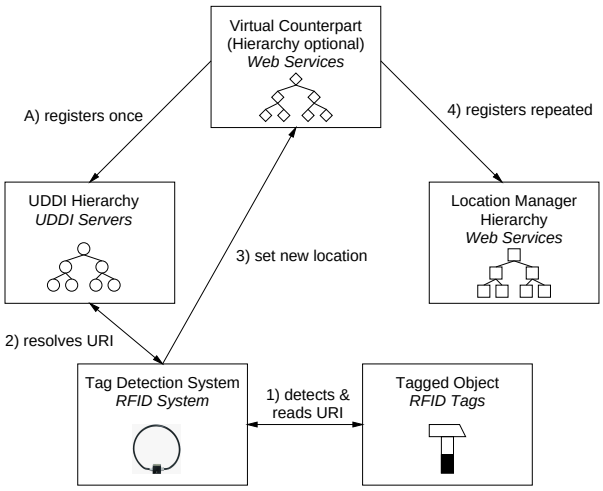

Figure 3. Infrastructure Overview of the Web Service Approach

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

4.2 Web Services Approach

Another approach to implement the concepts of Section 3 is the use of Web Services. Web Services seem to be appropriate for several reasons. First, the client/server paradigm is useful to model 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 client. Second, Web Services also provide a service description and discovery framework, which can be used to describe and locate virtual counterparts. Third, Web Services build on open standards like the Simple Object Access Protocol (SOAP) which makes them universally applicable. Fourth, the framework can then easily communicate with other third parties' Web Services on the Internet.

Figure 3 shows the main components of the infrastructure. The tag detection system scans for tagged objects in its read range. If a tagged object is detected, the system reads out an URI from the memory of the tag. The URI consists of the name 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 where 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) on its virtual counterpart. The virtual counterpart uses this location information to register itself at a hierarchy of location managers. Since all virtual counterparts have to register themselves at this hierarchy, a virtual counterpart can ask the hierarchy for its neighbors.

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 the 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.

Tag Detection System The tag detection system is the actual component that bridges the gap between the physical 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 in 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 strengths).

After a tagged object has come into the read range of an antenna, the tag detection system reads out the memory of the tag, which contains the URI of the virtual counterpart. In order to set the new location of the tagged object on 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 where the virtual counterpart resides. Using this information, the tag detection system can set the new location on 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 are the service description and the location of the service. The Web Service Description Language (WSDL) is used to describe the service interface, so that a client can access the service. A Web Service that is up and running has to register itself at a UDDI server with its Web server address, so that a client can locate the service.

Originally, the UDDI servers were intended to establish a service cloud. That means that all UDDI server that belong to a service cloud have to store the information about all services worldwide, i.e., a change at one UDDI server is propagated to all others within the cloud. We think that this does not scale well if every object is tagged. Therefore we extended the UDDI service cloud structure with a DNS-like partitioning which distributes all service information over the UDDI servers without redundancies (except some backup servers for reliability reasons).

The UDDI server generates a universal and unique identifier (UUID) if a service is registered the first time. We use this UUID also as the unique name for a virtual counterpart. This UUID is a random number and provides no structure. A structured identifier is necessary if we structure UDDI servers in a DNS-like style. Thus, we introduce 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.pervartis:40a96d21-ee00-0000-0080-e698e3243f5a. In this example "pharma.pervartis" denominates the UDDI server where 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" denominates the virtual counterpart. It is unique for each tagged object and independent of the UDDI server, which allows 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 where the service (i.e. the virtual counterpart) is running.

Virtual Counterpart As mentioned above, every virtual counterpart is implemented as a Web Service that is running 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, to retrieve the location history, and some methods to add and remove parent and child nodes depending on its position in a composition tree. All other methods that are specific for 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 at 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 as a tree. The root location manager is responsible for the whole world. The child nodes build a partition of the world. When a virtual counterpart has to register itself at 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 the cabinet that contains the medicine bottles, a location manager, which is responsible for the space of the cabinet, has a link to a so-called host service. This host service is the virtual counterpart of the tagged object that can contain other tagged objects.

5 Experiences with the Frameworks

We used the prototype frameworks described in Section 4 to realize a number of applications. Based on the experiences we evaluate the prototype frameworks by pointing out some of their their strengths and weaknesses.

In particular, we re-implemented several of the earlier applications using our frameworks. In general the development time dropped significantly since much fewer code had to be written. Applications written from scratch were hard to understand and maintain since they tended to mix up different functionalities (e.g., reading tags, program logic, user interface) in a monolithic program. The framework-based re-implementations gave a much better separation of concerns, which typically resulted in fewer errors and better maintenance capabilities. Understanding and using a framework, on the other hand, may present a significant initial overhead to a developer.

According to [7], the boundary of a ubicomp application supported by our frameworks is defined by a room-scale installation of some tag detection systems, which we call a cell. The size of such a cell and the scalability of a framework required to support a cell is quite limited. Larger systems can be built using multiple cooperating cells, which support a hand-over procedure when a tagged mobile object crosses the boundary between two cells. However, the frequency of such hand-overs is expected to be small. One example would be a tool moving from the workshop cell to the tool box cell in the Smart Tool Box application. Nevertheless, the framework must scale to a large number of interacting virtual counterparts (e.g., a whole truck filled up with medicine bottles as in the Smart Medicine application).

Besides the conceptual issues, we also considered performance aspects such as the memory usage per service, and the time a service lookup and service invocation takes. Both criteria are critical, the memory use determines how many virtual counterparts can be hosted on a server, and the time for a service lookup and invocation determines whether an application can react in real-time to changes in the real-world.

We conducted performance measurements for two platforms: Sun's Jini implementation and Microsoft's .NET Web Services, which is used in the second framework. The tests were performed on 451 MHz Intel Pentium III PCs with 256 MB of RAM running Windows XP. The computers were connected by a 100 Mbps Ethernet network. For

the tests we used a simple application which implements counterparts as described in Section 4 using Jini and .NET, respectively. We examined the memory footprint of the Jini/.NET runtime environments and the memory footprint of each virtual counterpart in the test application. Additionally, we measured the amount of time required for performing a counterpart lookup followed by the invocation of a simple method on the counterpart. We performed 20 runs and calculated averages.

The tests have shown that Jini is currently much more performant than .NET Web Services. The memory footprint of the Jini runtime environment is 9564 kB, 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 shows a better performance with respect to time measurements. A service lookup and an invocation of a simple method takes 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.

In the approach based on computational objects described in Section 4.1, scalability to large numbers of counterparts highly depends on the actual implementation of virtual counterparts. We examined three implementations: spawn a new process for each virtual counterpart, spawn a new thread for each virtual counterpart, and execute all counterparts in the same thread. While the first two approaches offer more flexibility due to decoupling the control flows of the counterparts, they are both too heavy-weight for large numbers of counterparts.

Both 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 and virtual world. The latter allows to decouple infrastructure services 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, which makes it more difficult to make use of the concepts in an application. Another difference is the support of migration and history – the first framework provides dedicated entities, which incorporate these concepts, whereas the second framework does not possess such entities. Also, the first framework introduces meta-counterparts and meta-location that are not part of the second framework. On the one hand, these concepts render the first framework somewhat more complex, but on the other hand enables more efficient applications. The challenge now consists in combining the proven concepts of both prototypes into an encompassing framework.

6 Related Work

There exist several other efforts to provide support for applications based on smart objects. The work of the MIT Auto-ID Center [13] comes closest to our intention to provide a framework for smart objects. The goal of the Auto-ID center and its sponsoring companies is to replace the traditional barcode with passive RFID tags. For this, they investigate 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 to exchange information about products. The middleware that controls the readers and processes the tag IDs is called Savant [10]. The Savants form a tree. The edge Savants directly control the RFID readers and store the tags IDs. Internal savants aggregate the data received from their child nodes. Savants also provide means to notify external programs with tag information or they run tasks that have to be registered at a Savant. One issue we miss in this approach is the virtual counterpart that actively reacts to changes in the real-world, in contrast to Savants that only manage passive database entries.

Cooltown [6] and the associated CoolBase infrastructure aim to give people, places, and things a Web presence. This Web presence has a similar function like 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 to include 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 sufficient to support large-scale applications.

The Stanford Interactive Workspaces project [5] 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 [15] projects have a similar focus.

Projects such as Gaia [11], One.World [4], and Microsoft Easy Living [1] aim to provide an infrastructure to support augmented environments in a rather 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, which are still rather generic. There is quite a large gap between the abstractions provided by these

projects and frameworks like ours, which support a rather specific application model (that of multi-object, tag-based applications in our case).

7 Conclusion and Outlook

Based on our experiences with several prototype applications, we came up with a set of basic functionalities 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 many 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 has the necessary performance to guarantee real-time requirements. To achieve this, we have defined three work packages. The first package focuses on different tagging and sensing systems in order to come up with an abstract interface for tag detection, independent of the actual tagging system. Secondly, we want to provide a service infrastructure for smart objects, including, for example, security mechanisms. Thirdly, we want to extend the concepts and implementations of virtual counterparts. All this together should eventually lead to a general and performant application framework for smart objects.

8 Acknowledgments

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 RFID Chef, Matthias Lampe for his work on Smart Tool Box, and Philip Graf, Martin Hinz, Svetlana Domnitcheva, and Vlad Coroama for their help with 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, Sept. 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] C. Floerkemeier, M. Lampe, and T. Schoch. The Smart Box Concept for Ubiquitous Computing Environments. Submitted for publication.

[4] R. Grimm et al. Programming for Pervasive Computing Environments. Technical Report UW-CSE-01-06-01, University of Washington, Department of Computer Science and Engineering, June 2001.

[5] B. Johanson, A. Fox, and T. Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing*, 1(2):71–78, Apr. 2002.

[6] T. Kindberg et al. People, Places, Things: Web Presence for the Real World. In *WMCSA 2000*, Monterey, USA, Dec. 2000.

[7] T. Kindberg and A. Fox. System Software for Ubiquitous Computing. *IEEE Pervasive Computing*, 1(1):70–81, Jan. 2002.

[8] 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, Oct. 2000.

[9] G. J. Nelson. *Context-Aware and Location Systems*. PhD thesis, University of Cambridge, 1998.

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

[11] M. Roman, C. Hess, R. Cerqueira, A. Raganat, R. Campbell, and K. Nahrstedt. Gaia: A Middleware Infrastructure to Enable Active Spaces. Submitted to IEEE Pervasive Computing Magazine.

[12] K. Römer and S. Domnitcheva. Smart Playing Cards - A Ubiquitous Computing Game. *Journal for Personal and Ubiquitous Computing*, 6(6), Nov. 2002.

[13] D. B. S. Sarma and K. Ashton. The Networked Physical World - Proposals for Engineering the Next Generation of Computing, Commerce & Automatic Identification. Technical Report MIT-AUTOID-WH-001, MIT Auto-ID Center, Oct. 2000.

[14] F. Siegemund and C. Floerkemeier. Interaction in Pervasive Computing Settings using Bluetooth-enabled Active Tags and Passive RFID Technology together with Mobile Phones. In *PerCom 2003*, Fort Worth, USA, Mar. 2003.

[15] P. Tandler. Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices. In *Ubicomp 2001*, Atlanta, USA, Sept. 2001.

[16] H. Vogt. Efficient Object Identification with Passive RFID Tags. In *Pervasive 2002*, pages 98–113, Zurich, Switzerland, Aug. 2002.

[17] 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.

[18] The Philips I–Code System. www-us2.semiconductors.philips.com/identification/-products/icode.