Software abstractions for mobile RFID-enabled applications

Andoni Lombide Carreton*,†, Kevin Pinte and Wolfgang De Meuter

Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium

SUMMARY

Our everyday environments may soon be pervaded with radio frequency identification (RFID) tags integrated in physical objects. These RFID tags can store a digital representation of the physical object and transmit it wirelessly to pervasive, context-aware applications running on mobile devices. However, communicating with RFID tags is prone to many failures inherent to the technology. This hinders the development of such applications, as traditional programming models require the programmer to deal with the RFID hardware characteristics manually. On the other hand, traditional RFID middleware focuses on limited scenarios in an enterprise context and not on general ubiquitous computing scenarios. In this paper, we extend the ambient-oriented programming paradigm to program RFID applications, by considering RFID tags as intermittently connected mutable proxy objects hosted on mobile distributed computing devices, and detail our prototype implementation. Copyright © 2011 John Wiley & Sons, Ltd.

Received 27 July 2010; Revised 31 May 2011; Accepted 1 August 2011

KEY WORDS: RFID; pervasive computing; ambient-oriented programming; mobile RFID-enabled applications

1. INTRODUCTION

Radio frequency identification (RFID) is generally considered as a key technology in developing pervasive, context-aware applications [1, 2]. RFID tags are becoming so cheap that it will soon be possible to tag one’s entire environment, thereby wirelessly dispersing information to nearby context-aware applications. An RFID system typically consists of one or more RFID readers and a set of tags. The RFID reader is used to communicate with the tags, for example, to inventory the tags currently in range or to write data on a specific tag. RFID tags can either be passive or active. Active tags contain an integrated power source (e.g. a battery), which allows them to operate over longer ranges and to have more reliable connections. Some even have limited processing power. Passive tags are more commonly used because they are very inexpensive. Passive tags use the incoming radio frequency signal to power their integrated circuit and reflect a response signal. Most RFID tags possess non-volatile memory on which they can store a limited amount of data. The technologies on which we focus (although not limiting ourselves to them) are cheap, writable passive tags and RFID readers integrated into mobile devices (such as smartphones).

This technology gives rise to distributed applications running on mobile devices that both disperse application-specific data to and process contextual data from tagged physical objects in their environment. They spontaneously interact with physical objects without assuming any additional infrastructure. We will refer to such applications as mobile RFID-enabled applications (see Section 3.1 for an example). These applications use RFID technology in a radically different way than RFID systems deployed today, which only use RFID tags as digital barcodes and almost never

*Correspondence to: Andoni Lombide Carreton, Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium.
†E-mail: alombide@vub.ac.be

Copyright © 2011 John Wiley & Sons, Ltd.
exploit the writable memory on these tags. Furthermore, today’s systems assume infrastructure in the form of a centralized backend database that associates the digital barcode with additional information. Although this is certainly useful, we also want to support entirely ad hoc applications where tagged objects cause applications in communication range to spontaneously react on their presence without the need for such infrastructure. For example: users can tag the books in their home library without setting up a database for this purpose. In this case, the notion of being close to a book or a number of books is essential, as a home library is typically not organized in a way that can be exploited by a centralized approach.

In mobile RFID-enabled applications, communication with RFID tags is prone to many failures. Tags close to each other can cause interference and can move out of the range of the reader while communicating with it. These failures may be permanent, but it may be that at a later moment in time the same operation succeeds because of minimal changes in the physical environment. For example, a tag moves back in range or suddenly suffers less from interference. As a consequence, dealing with these failures and interacting with the low-level abstraction layers offered by RFID vendors from within a general purpose programming language results in complex and brittle code.

In this paper, we propose a natural extension to distributed object-oriented programming [3] by aligning physical objects tagged with writable RFID tags as true mutable software objects. We will model these objects as proxy objects acting as stand-ins for physical objects. For this model to be applicable to mobile RFID-enabled applications, it must adhere to the following requirements:

R1: Addressing physical objects. RFID communication is based on broadcasting a signal. However, to be able to associate a software object with one particular physical object, it is necessary to address a single designated physical object. Object identities of digital objects should always correspond to object identities of physical objects.

R2: Storing application-specific data on RFID tags. Because we want to assume as little infrastructure as possible to implement mobile RFID-enabled applications, we do not rely on a backend database, and hence it should be possible to store the application data in the writable memory of the RFID tags themselves [4].

R3: Reactivity to appearing and disappearing objects. It is necessary to observe the connection, reconnection, and disconnection of RFID tags to keep the proxy objects synchronized with their physical counterparts. Differentiating between connection and reconnection is important to preserve the identity of the proxy object. Furthermore, it should be possible to react upon these events from within the application.

R4: Asynchronous communication. To hide latency and keep applications responsive, communication with proxy objects representing physical objects should happen asynchronously. Blocking communication will freeze the application as soon as one physical object is unreachable.

R5: Fault-tolerant communication. Treating communication failures as the rule instead of the exception allows applications to deal with temporary unavailability of the physical objects and makes them resilient to failures. For example, read/write operations frequently fail because of hardware phenomena.

R6: Data consistency and security. Different mobile applications might concurrently read and – more importantly – write data to a number of tagged objects, all within their proximity. This can lead to data races that have to be prevented. Similarly in some scenarios, data stored on RFID tags may not be read or modified by unauthorized users.

This paper is an extended version of [5] in which we additionally discuss our prototype implementation, hint at how our approach can be integrated with more traditional RFID infrastructure, and extend our implementation with a mechanism to guarantee data consistency. It is organized as follows: Section 2 discusses related work. Section 3 starts by introducing a mobile RFID-enabled application scenario. Thereafter, we use the scenario as a running example to present the language constructs that make up our model. Even though we explicitly focus on mobile RFID-enabled
applications, we demonstrate that our abstractions apply to traditional RFID applications equally well. Section 4 discusses the implementation of the framework and presents the results of performance benchmarks conducted with our prototype implementation. Section 5 concludes this paper by summarizing the limitations and contributions of our approach.

2. RELATED WORK AND MOTIVATION

This section discusses the current state of the art concerning RFID applications and supporting software, and how current approaches do not meet the requirements listed in the previous section.

2.1. Radio frequency identification middleware

Typical application domains for RFID technology are asset management, product tracking, and supply chain management. In these domains, RFID technology is usually deployed using RFID middleware, such as AspireRFID [6] and Oracle’s Java System RFID Software [7]. RFID middleware applies filtering, formatting, or logic to tag data captured by a reader such that the data can be processed by a software application.

Such traditional middleware uses a setup where several RFID readers are embedded in the environment, controlled by a single application agent. These systems rely on a backend database, which stores the information that can be indexed using the identifier stored on the tags. They use this infrastructure to associate application-specific information with the tags, but do not allow storing this information on the tags directly (requirement R2). Therefore, these systems are not suited to develop mobile RFID-enabled applications.

WinRFID [8] is an RFID middleware that is entirely based on the .NET Framework and Windows services, which are specified in XML. Services can read from and write data onto RFID tags using an object-oriented abstraction. The tag data are also specified in XML and are converted back and forth to a simplified and compressed format when written onto tag memory. The main drawback of WinRFID, however, is that the devices and/or services have to be explicitly registered into a registry component, such that the services can contact this registry to interact with, for example, RFID readers that were a priori registered. This makes WinRFID unsuitable for mobile RFID-enabled applications.

Fosstrak [9] (formerly named Accada) is an open source RFID middleware platform that is based on the Electronic Product Code standards [10]. Fosstrak offers a virtual tag memory service (VTMS) that facilitates writing to a tag by shielding the application from the characteristics of RFID tag memory: limited memory size, different memory organizations, reduced write range. If a write succeeds, the reader module acknowledges this to the host and stores a backup copy of the data in the virtual representation of the tag in the VTMS. If the tag memory gets corrupted at a later stage or the host wants to access the memory of the tag while the tag is outside the range of any reader, the data can be made available via this virtual memory. If the write to the tag fails as a result of insufficient power, the key-value pair will be stored in the VTMS and flagged as “open”. The reader will retry the write command at a later point of time. If there is insufficient memory space, the host will receive an appropriate error message, and the key value will be stored in the virtual tag memory only. The host can also indicate that the virtual memory of a tag can only be accessed once the tag is in the read range of the particular reader. Hence, Fosstrak allows addressing individual tags and storing application-specific data on these tags using asynchronous and fault-tolerant communication. For this, however, the application data have to be converted back and forth to simple key-value pairs of which the values can only be of restricted set of simple data types.

2.2. Radio frequency identification in pervasive computing

In [11], mobile robots carrying an RFID reader guide visually impaired users by reading RFID tags that are associated with a certain location. In [12], users are equipped with mobile readers and RFID tags are exploited to infer information about contextual activity in an environment based on the objects they are using or the sequence of tags read by the reader. Rememberer [13] provides visitors of a museum with an RFID tag. This tag is used as the user’s memory of the visit and stores
detailed information about selected exhibitions. However, none of the preceding systems provide a
generic software framework to develop mobile RFID-enabled applications, but instead use ad hoc
implementations directly on top of the hardware.

In [14], RFID tags are used to store application-specific data. The RFID tags form a distributed
tuple space that is dynamically constructed by all tuples stored on the tags that are in reading range.
Mobile applications can interact with the physical environment (represented by tuples spaces) by
means of tuple space operations. The system not only allows reading data from RFID tags, but at any
time, data in the form of tuples can be added to and removed from the tuple space. However, there
is no way to control on which specific tag the inserted tuples will be stored. RFID tags cannot repre-
sent physical objects as there is no way to address one specific RFID tag as dictated by requirement
R1. Hence, the programmer must constantly convert application data types (e.g. objects) to tuples
and vice versa. Therefore, this approach suffers from the object-relational impedance mismatch [15]
and does not integrate automatically with object-oriented programming.

3. DISTRIBUTED OBJECT-ORIENTED PROGRAMMING WITH RFID-TAGGED OBJECTS

In this section, we discuss our RFID programming model. It is conceived as a set of language con-
structs that satisfy all requirements listed in Section 1, except security, which remains future work.
We do this by means of an example mobile RFID-enabled application that we use as a case study to
motivate our implementation. First, we introduce the general idea of the application.

3.1. A mobile RFID-enabled application scenario

The scenario consists of a library of books that are all tagged with writable passive RFID tags. The
user of the application carries a mobile computing device that is equipped with an RFID reader. On
this device, there is software running that allows the user to see the list of books that are nearby
(i.e. in the reading range of the RFID device) sorted on different properties of the books (e.g. author,
title, ...). This list is updated with the books that enter and leave the range as the user moves about in
the library. Additionally, the user can select a book from the list of nearby books, on which a dialog
box opens. In this dialog box, the user can write a small review about the book. This review is stored
on the tagged book itself. Other users can then select that same book from their list of nearby books
and browse the reviews on the book, or add their review.

In a more generalized setting, the information available on the tagged books could be enriched
with information stored on an external database. For the sake of brevity, we consider the base case
where the example application is an entirely ad hoc implementation.

3.2. Ambient-oriented programming with RFID tags

In the mobile RFID-enabled application introduced in the previous section, mobile devices hosting
the application move throughout an environment of tagged books. These books dynamically enter
and leave the communication range of the mobile devices and interact spontaneously. These proper-
ties are very similar to the ones exhibited by distributed applications in mobile ad hoc networks
[16]. Similar to mobile devices in mobile ad hoc networks, RFID tags and readers should interact
spontaneously when their ranges overlap.

Ambient-oriented programming [17] is a paradigm that integrates the network failures inherent
to mobile ad hoc networks into the heart of its programming model. To this end, ambient-oriented
programming extends traditional object-oriented programming in a number of ways. First, when
objects are transferred over a network connection, it is not desirable having to send the class defi-
nition along with the object. This leads to consistency problems and performance issues. Hence, a
first characteristic of ambient-oriented programming is the usage of a classless object model [18]. A
second characteristic is the use of non-blocking communication primitives. With blocking commu-
nication, a program will wait for the reply to a remote computation causing the application to block
whenever a communication partner is unavailable [19]. The last characteristic is dynamic device
discovery to deal with a constant changing network topology without the need for URLs or other
explicit network addressing. Because we are modeling physical objects in a pervasive computing environment as self-contained software objects, ambient-oriented programming provides a fitting framework to cope with the problems listed in the introduction.

A promising model epitomizing this paradigm is a concurrency and distribution model based on communicating event loops [20]. In this model, event loops form the unit of distribution and concurrency. Every event loop has a message queue and a single thread of control that perpetually serves messages from the queue. An event loop can host multiple objects that can be published in the network. Other event loops can discover these published objects, obtaining a remote reference to the object. Client objects communicate with a remote object by sending messages over the remote reference, the messages are then placed in the mail queue of the event loop hosting the remote object. The event loop’s thread handles these messages in sequence ensuring that the hosted objects are protected against race conditions. A remote reference operates asynchronously, the client object will not wait for the message to be delivered, but immediately continues with other computations. Within the same event loop, local object references are accessed using regular, synchronous message sending. Figure 1 illustrates the communicating event loops model. The difference with actor programming languages [21] is that actors are in essence functional abstractions of which the entire behavior has to be replaced (become) instead of hosting mutable objects as the unit of modularity. When mobile devices move out of each others range, the event loops that are hosted on the different devices are disconnected from each other. However, upon such a disconnection, all remote references become disconnected and buffer incoming messages, as illustrated by Figure 2. When the communication is reestablished, the remote references are automatically restored, and all buffered messages are automatically flushed to the message queue of the destination event loop.

AmbientTalk is an ambient-oriented programming language that uses the communicating event loop model as its model for concurrency and distribution [22]. It is conceived as a scripting language that eases the composition of distributed Java components in mobile ad hoc networks, and it inherits...
most of its standard language features from Self, Scheme, Smalltalk, and E. From Scheme, it inherits the notion of true lexically scoped closures. From Self and Smalltalk, it inherits an expressive block closure syntax, the representation of closures as objects and the use of block closures for the definition of control structures. Inspired by Self, AmbientTalk features classless slot-based objects. The concurrency model of the language was adopted from the E programming language.

We implemented our RFID system in AmbientTalk, and in the next sections, we introduce the concrete language abstractions that allow us to program with RFID-tagged objects as mutable software objects. Each of the next sections corresponds to a requirement formulated in Section 1 and is numerated accordingly.

**R1. RFID-tagged objects as proxy objects**

As discussed earlier, we model RFID-tagged objects as proxy objects. An example of a book proxy object is given next. It contains slots for the ISBN, title, and reviews, and provides two mutator methods to update the book’s title and add reviews:

```ruby
deftype Book;
def aBook := object: {
def ISBN := 123;
def title := "My Book";
def reviews := Vector.new();

def setTitle(newTitle)@Mutator {
title := newTitle;
};

def addReview(review)@Mutator {
reviews.add(review);
};
taggedAs: Book;
```

The hardware limitations of RFID tags render it impossible to deploy a full fledged virtual machine hosting objects on the tags themselves. We thus store a serialized data representation of a proxy object on its corresponding tag. Because we use a classless object model, objects are self-contained; there is no class that defines their behavior. Upon deserialization, the object’s behavior (its methods) is preserved and used to reconstruct the proxy object (see Section R2). Because we cannot rely on classes to categorize objects, we use type tags. These are “mini-ontologies” that are attached to an object to identify its “type”. In the preceding example, we define a type Book on line 1 and attach that type to the aBook object in line 14. In Section R3, we use the type tag to discover objects of a certain kind.

Of course, the data stored on the tags have to be synchronized with the state of these proxy objects. Methods that change the state of the book objects are annotated by the programmer with the Mutator annotation‡. These annotations are used by the implementation to detect when objects change and have to be written to the corresponding tag. For example, calling the addReview mutator method on a book object first updates the reviews field by adding the new review. Subsequently, the system serializes the modified book object and stores it on the correct RFID tag.

The proxy objects are managed by what we will henceforth denote as the RFID event loop, as shown in Figure 3. It controls an RFID reader to detect appearing and disappearing tags (a and b), and it associates proxy objects with them (A and B). These proxy objects can then be used by other event loops to interact with the tags as if they were mutable software objects. They do this by obtaining remote references to the proxy objects. Remote references (X and Y) reflect the state of

‡AmbientTalk is a highly dynamic programming language, which makes it hard to determine from the source code if mutating operations are going to be invoked. Additionally, allowing the programmer to specify which methods are mutators and which are not leaves room for optimizations or other reasons not to (immediately) write through the changes to the object.
Figure 3. Overview of the RFID event loop.

the corresponding RFID tags (a and b). When a tag moves out of range of the reader, the remote reference is signaled of this disconnection; conversely, when a tag moves back in range, the remote reference is signaled of the reconnection.

R1.1. Providing a uniform interface for different RFID hardware. Embedding RFID technology in the computational model described in Section 3.2 requires a uniform interface to shield application programmers from hardware-specific protocols. In our approach, the hardware-specific glue code from the device level up to the application interface level is encapsulated in the RFID event loop (see Section 4 for an overview of the architecture). This allows multiple RFID readers to be used in one or more applications using the same interface. Figure 4 depicts a mobile device that hosts three different applications that interface with RFID hardware by means of two RFID event loops steering different types of RFID hardware. This not only allows abstracting over hardware protocols but also over where the data associated with the RFID tag is physically stored, as we will discuss in the following section.

R1.2. Object storage transparency. Although we are focusing on pervasive applications that do not assume any infrastructure, our abstractions do not limit programmers to this kind of application. In fact, when using the same programming model, it can be made transparent whether the data associated with an RFID tag is stored onto the tag itself or is stored in a (potentially distributed) external database (see Section 4). The key point here is that although the data itself may be permanently accessible (using an internet connection for example), the application interacting with the physical world might still be interested whether the physical object denoted by the tag is in communication range (see requirement R3), and in addition the same interface to program RFID applications is offered to the application programmer.

Figure 4 depicts a mobile device that hosts three different applications. The library application looks up information associated with tagged books in a centralized database by means of the serial number on the tag, which is scanned by a stationary reader to which the mobile device is connected.
(for example, using a local Wi-Fi network). The other two applications directly use the RFID reader built into the mobile device and do not assume any infrastructure by storing all necessary data on the tags themselves. The first approach can utilize well-understood database technology, and the latter approach is the topic of the next section, in which we will discuss an infrastructure-less version of the library application. This architecture allows different hardware abstraction layers and/or persistence layers to be encapsulated by the same interface towards applications: the RFID event loop. All applications are encapsulated in their own event loop. The RFID event loop notifies client applications of the appearance and disappearance of RFID tags and takes care of (sequentially) scheduling messages sent to RFID proxy objects hosted by the event loop and of writing the necessary data on the concrete storage infrastructure associated with the proxy object.

**R1.3. Maintaining item identity using multiway references.** We represent RFID tags as remote proxy objects. These proxy objects are remote to client applications, where the connectivity to a physical RFID tag is reflected by the connectivity state of the remote reference. In the case the mobile device has discovered a tag using an internal reader, the corresponding proxy object is hosted locally on the device. The RFID event loop of the device then not only offers the proxy object as a remote reference to applications locally hosted on the device but also to any other application running on another device that is interconnected with the device hosting the proxy object. For (mobile) devices that rely on one or more external RFID readers, the connectivity state of a remote reference reflects both the physical presence of the respective RFID tag in range of the reader as well as the connectivity to the device hosting the proxy object to the RFID tag. Figure 5 illustrates two scenarios where a mobile device acquires remote references to proxy objects for tags discovered by a fixed reader and another mobile device with an internal reader.

Our approach installs an RFID event loop per RFID reader. This entails that per tag discovered, per RFID reader, a proxy object for this tag is constructed. Therefore, an application that discovers RFID tags from more than one reader might end up with two separate references to the same physical object, through different proxy objects. This is illustrated by Figure 6. However, to satisfy...
requirement R1, object identity must be maintained. We addressed this issue by introducing the concept of multiway references [23].

Multiway references are a generic abstraction that collapse remote references considered “equal” based on an equivalence relation associated with every single reference. For RFID proxy objects, remote references are considered “equal” if they point to proxy objects that represent an RFID tag with the same unique serial number. To applications that acquire references, a multiway reference appears as a single remote reference. However, the multiway reference incorporates the different ways a physical object can be reached. Figure 6 shows a multiway reference to a book object that can be reached in two ways. The client application however discovers the book only once: when new proxy objects for the same physical book are discovered, the multiway reference is extended with the references these new proxy objects. A multiway reference is thus able to reach a single RFID tag via multiple paths and does the bookkeeping of the connection status of each of the paths. A multiway reference is only disconnected if each single reference it incorporates is disconnected; this is illustrated in Figure 7. As with remote references described in Section 3.2, all messages sent to a disconnected multiway reference are buffered in the reference. As soon as one of the encapsulated remote references reconnects or a new path is discovered, the stored messages are flushed over the connected path. Each multiway reference keeps one of its references as the active path. When sending messages to the multiway reference, the messages will flow over this active path to the destination. Selecting the reference that becomes the active path happens by ranking the encapsulated remote references based on the priority carried by each of them. This priority can be anything from a simple number indicating, for example, the number of hops a reference has to make to reach the device hosting the target proxy object, to a full fledged object that can embody the number of hops together with more detailed information as the range of the reader, the signal strength, and so on.

Messages to proxy objects may cause side-effects and thus alter the RFID tag’s state. The different RFID event loops hosting proxy objects for the same RFID tag are responsible for synchronizing the state of the proxy objects. If one proxy object is updated and persists its state to the RFID tag (or database), the other RFID event loops will reflect the changes upon the next readout of the tag (see Section 4 for implementation details). Multiway references behave exactly as ordinary remote references do, and both are interchangeable. For the remainder of the text, we will employ the term “remote reference”; note that they are actually multiway references.

R2. Storing objects on RFID tags

When the RFID event loop detects a blank RFID tag, the tag is represented by a generic proxy object, which responds to only one message: initialize. The code next shows how a blank tag is initialized as a book object:

```plaintext
when: tag<-initialize(aBook) becomes: ([book] ...);
```

![Figure 7. A multiway reference is connected up until all of its composing remote references are disconnected.](image-url)
The RFID event loop generates a data representation of the aBook object by serializing it and stores this data on the RFID tag that corresponds with the tag proxy object. For storing objects on RFID tags, we currently employ a custom representation that requires a low memory footprint while still preserving method implementations of objects. The serialization strategy is not interleaved with the rest of the implementation of the system to allow more standardized object representations to be used such as XML. Note that these representations probably have to be extended with support for prototype-based objects (by including method implementations into the serialized representation) to obtain the same functionality that we are proposing here.

The reference tag to the generic proxy object is obtained using the discovery constructs we explain in Section R3. From this point on, the RFID tag is no longer “blank” as it contains application specific data. When storing the object on the tag succeeds, the call to initialize returns with a new remote reference book that points to a newly constructed proxy object (the when:becomes:-construct is explained in Section R4) representing the book. The RFID event loop keeps track of the unique link between a proxy object and a tag by means of the serial number that each tag carries. Note that the concrete behavior of performing side effects on such a proxy object depends on the underlying hardware implementation. Using an entire ad hoc implementation where all the data are stored on the tag memory itself behaves differently than when the data are stored in a relational database to which the mobile device has access. However, the interface provided to the application programmer is the same.

R3. Reactivity to appearing and disappearing objects

As explained in Section R1, the RFID event loop notifies other event loops of the appearance and disappearance of the objects they have remote references to. In the code example shown next, an event handler that will execute a block of code each time an object of type Book is discovered is installed using the whenever:discovered: construct. The registered code block is parametrized by the remote reference to the book object (which is also used to send it asynchronous messages).

```plaintext
whenever: Book discovered: (|book|
    whenever: book disconnected: { // react on disappearance ;
    whenever: book reconnected: { // react on reappearance ;
    );
```

Once a remote reference to a book is obtained, within the whenever:discovered callback, two more event handlers can be registered on the book remote reference using the whenever:disconnected: and whenever:reconnected: constructs. These allow one to install a block of code, which is executed as soon as the object denoted by the book remote reference moves in or out of range of the reader♯. Notice that upon reconnection, the proxy object maintains its identity through the book reference. For each whenever-handler, there exists a when-variant that executes only once.

Note that the semantics of a tag being “connected” depends on the underlying hardware implementation. Using a built-in close-range RFID reader leads to different results than that of using a wireless connection to a stationary high-performance RFID reader.

R4. Asynchronous communication

Applications that acquire a remote reference to a proxy object can communicate with it via asynchronous message sending. Messages sent to proxy objects are handled sequentially by the thread encapsulated in the RFID event loop. This ensures that all proxy objects hosted by the RFID event loop are protected against race conditions. When the remote reference to a proxy object is disconnected, all messages sent to it are locally buffered in the remote reference. When the connection is restored, the messages are flushed to the RFID event loop’s message queue. This means that a

♯These constructs return an object, which implements a method that cancels the callback.
message sent to a proxy object of which the RFID tag temporarily suffers from interference or is temporarily unavailable will eventually be processed.

Messages sent to proxy objects can either retrieve data (read operations) or trigger behavior that causes side effects (write operations). Both kinds of operations aim to keep the tag synchronized with the proxy object. Performing a read operation on a proxy object causes the proxy object to be updated with the data on the corresponding tag. Performing write operations first causes a side effect on the proxy object, thereafter the corresponding RFID tag is updated to contain the modified proxy object. Reading and writing tags is thus caused by sending messages to the proxy objects; this also means that access to the RFID reader is managed by the RFID event loop’s message queue and protected against concurrent access.

Asynchronous messages are sent using the `<-` operator. The following example asks a book for its title and displays it:

```kotlin
when: book<-getTitle() becomes: { |title| system.println(title));

system.println("here first");
```

The asynchronous call to `getTitle` immediately returns with a `future` object. Such a future object can be used to notify callbacks that the return value of the asynchronous call was received. This happens by means of the `when: becomes:`-construct. Using this construct, a block of code can be registered on the future that is executed once the future signals that the return value of the message was received, taking the return value as an argument. This example thus immediately prints the string "here first" and only after the `title` future signals the reply, it prints the title of the book. If the RFID tag corresponding to the book object has disappeared upon sending the message, the remote reference buffers the message until the tag reappears. This message will only be sent when the RFID tag represented by the remote reference is back in range.

### R5. Fault-tolerant communication

Buffering an asynchronous message to a proxy object ensures that the message will eventually be sent if the tag moves in range. This makes the communication fault-tolerant as no exception is raised when the object is unavailable for a short period of time. However, failures may not be temporary; a tag may move out of range and never return again. Using the `Due` annotation, we can annotate the message sent with a duration that controls how long a message is buffered before timing out. For example, we can add short reviews to a book:

```kotlin
def myReview := "not suitable for beginners";
  // message processed successfully
  ) catch: TimeoutException using: { |e|
    // message timed out
  };
```

Suppose the RFID tag corresponding with `book` would leave the reader’s range before the `addReview` message is received by the book’s proxy object. Then the message is buffered for at most 10 s. If the tag does not respond in time, a timeout exception is raised. If the tag reappears in range within this time frame, the message to add the review `myReview` is delivered to the RFID event loop, and the corresponding book object is updated and stored on the RFID tag. Remember from Section R1 that `addReview` was annotated as a mutator method. This means that first the `reviews` field of the proxy object is updated by adding the new review. Subsequently, the RFID event loop serializes the changed book object and stores it entirely on the correct RFID tag. Only after both of these operations complete successfully, the future object triggers all its registered `when`-observers. If this did not happen within the 10-s timeframe, the exception is signaled to client applications, and their registered `catch`-blocks are invoked.
R6. Data consistency and security

The difficulty of ensuring data consistency and security and the effectiveness of the applicable techniques greatly depend on the RFID hardware used and the assumptions that an RFID-enabled application can make. In this section, we describe a number of possible approaches and their trade-offs, and we detail our approach to ensuring data consistency in mobile RFID-enabled applications.

R6.1. Ensuring data consistency. A first issue here to consider is the type of RFID hardware used. Passive tags, as the ones we have used in our experiments, offer no opportunity to programmatically coordinate read and write operations. They do offer low-level protocols (e.g. different variants of the ALOHA protocol [24]) for preventing read and write collisions, but these are automatically exploited by the RFID reader when reading the data from or writing it on tag memory. For our implementation, this means that objects will always be consistently written by the hardware each time a mutator method returns (i.e., data cannot be garbled because of concurrent writes). However, on a smaller granularity, successive read and write operations within a single method invocation, such as incrementing a counter based on a previously read value, can lead to data loss (as the proxied counter might have been concurrently updated by another RFID-enabled device before the method returns).

Hence, ensuring data consistency with passive RFID tags requires coordination among writers before physically writing the data on the tags. This is no problem in a setting where RFID-enabled applications can coordinate by means of a shared centralized entity, for example, by all connecting to an internet server that manages a certain type of tagged objects. The location of such servers could be stored on the tags to minimize configuration.

In the base case that we have considered in this paper where no additional infrastructure such as a centralized server is assumed, the only way of coordinating write operations is by means of the tag itself. Our current solution is to allow RFID event loops to function in two modes. In the first mode, the RFID event loop only signals the RFID reader to power up nearby tags when they have to be detected, read, or written. Because requests from different readers might interleave, distributed race conditions can occur in this mode. In the second mode, the RFID event loop instructs the RFID reader to keep its radio frequency (RF) field active, permanently powering all reachable tags, unless a client application explicitly signals the RFID event loop to shortly power down its RF field. The result is that RFID tags that are powered by the RF field of a single reader, grant exclusive access to that reader and ignore commands from all other readers. When the tag moves out of range of the reader or when the reader temporarily switches off its RF field, it loses its exclusive access to the tag and, by doing so, grants exclusive access to other nearby RFID devices willing to interact with the tagged object. The following code snippet shows how losing exclusive access to a tagged object raises an exception in the client application such that defensive code can be triggered:

```
when: book<getRating{} becomes: (|rating| // obtain rating
    book<updateRating(rating + 1);       // increment with exclusive access
) catch: NoExclusiveAccessException using: (|e|
    // handle lost exclusive access
);
```

We have found this approach satisfactory in small scale scenarios, such as the tagged library used as a case study in this paper. In larger scale applications, keeping a number of tags powered to obtain exclusive access to them may cause performance issues, because potentially a large number of tags can cease responding and become invisible to other RFID readers.

---

When an RFID reader powers up its antenna, it generates an “RF field” in which the circuits of tags are powered. The Google Android smartphone OS for example keeps the RF field of phones active when the screen is lit, assuming that only then the user is interacting with nearby tags. The RF antenna in such a phone is specifically designed to consume little power in such an active state.
When looking at more advanced hardware such as active RFID tags, there are more possibilities. We have just started experimenting with active RFID tags, but their autonomous nature allows to, for example, assign time slots (i.e., leases [25]) in which applications are allowed exclusive access to its memory before either aborting all operations or committing them before giving another application write access.

R6.2. Security. Security deals with unauthorized access to RFID tag memory and the reception of data transmitted in RF signals among RFID readers and tags (sniffing). Some passive RFID tags provide password protection to their memory. Although in our approach serialized objects are simply stored as clear text compressed using gzip. This data could be encrypted before storing. The question here is how to safely and conveniently exchange encryption keys. Again, different trade-offs can be made, which are out of the scope of this paper.

In general, there currently exist no security measures for passive RFID technology that can entirely avoid unauthorized memory access or sniffing [4, 26]. Just like any other approach, unless workable RFID security models are established, we suggest the use of RFID for applications dealing with non-critical data. Similarly to the problem of consistency described earlier, using active tags instead of passive tags provides much more possibilities.

3.3. Addressing specific groups of RFID-tagged objects

As mentioned in Section 2, RFID tags are typically used in large quantities, for example, in warehouse applications. In mobile RFID-enabled applications, it is often necessary to address a specific group of objects. For example, for all tags that represent a certain product, the price stored on the tag should be updated. However, such a collection of RFID tag objects has a highly dynamic nature as a result of the volatile connections with the RFID tags. At any point in time, tags move out of range, and new tags move in range. Instead of forcing the programmer to manually manage collections of nearby objects, AmbientTalk has a dedicated abstraction to discover and address a group of objects: ambient references [27]. At any point in time, an ambient reference designates the set of proximate objects of a certain type. This abstraction is applicable because we represent physical objects as remote proxy objects. An ambient reference represents a variable collection of proxy objects, for example, the set of nearby books. This set is updated behind the scenes when books move in and out of range. The example next shows an ambient reference to all books in the proximity, denoted by the `Book` type:

```def books := ambient: Book;
```

Ambient references allow to specify various predicates to refine the set of objects designated. This is shown in the example next where books are selected based on their category attribute:

```def computerScienceBooks := ambient: Book where: [|b| b.category == "Computer Science";
];
```

A last example shows how we can address a single object out of the group of nearby objects encapsulated in the ambient reference. For example, if all books about computer science are placed in the same shelf in the library, it is sufficient to query any book about this topic in range for its shelf:

This happens by annotating the `getShelf` message with `@Any`. We can also reach all objects in range using one-to-many communication. The last line of the example updates the shelf where computer science books should be located (e.g., because they have to be moved). The `Sustain` annotation causes the `setShelf` message to be perpetually sent to newly discovered computer science books.
3.4. Putting it all together

Finally in this section, we bring together the language constructs presented throughout this paper to implement the example application introduced in Section 3.1\(^1\). First of all, while the user moves about in the library, the list of nearby books has to be updated, as shown next:

```plaintext
def shelfFuture := computerScienceBooks<-getShelf()@Any;
when: shelfFuture becomes: { |shelf|
    system.println("The book should be on shelf: " + shelf);
};

computerScienceBooks<-setShelf("5D")@Sustain;
```

The first line declares the `Book` type, and the second line creates an ambient reference that refers to all books in range. On line 4, the asynchronous message `getBookInfo` to the `books` ambient reference is annotated with `@Sustain`, which causes the ambient reference to perpetually send this message to newly appearing books. This returns a `multifuture`, that is, a special future object that can trigger the same callback block multiple times with a new value. This callback is registered on the multifuture with a special `when`-construct (`whenEach:becomes:`). The code block is triggered each time the multifuture is resolved with a new return value from the message invocation on the ambient reference. The return value of this message is the info about the book (i.e., ISBN number, title and authors) and a reference to the book object. These return values are bound to the `infoAndRef` parameter of the observer block, which is added to the list in the user interface object. This causes the user interface to show a new entry in the list of nearby books, and to associate a reference to the book entry in this list.

On line 9, for every book discovered, a `whenever:disconnected:` observer is installed that, when triggered because a book went out of range, removes the book from the list in the user interface by means of the `book` remote reference. Notice that although the remote reference points to an unreachable book, it can still be used to look up the book in the list and remove it. This is an example of the system being tailored towards scenarios where disconnections are the default rather than the exception.

As mentioned earlier, the references to the books are being associated with the list entries. This way, when a user double clicks on a list entry, a dialog box is shown in which the user can type a small review or some comments about the book. When accepting the input data of the dialog box, the application attempts to add the text the user just entered to the list of reviews associated on the book itself. As we showed earlier in Section R4, invoking the `addReview` method on a book is a mutating operation (i.e., the method is tagged as a `Mutator`), which causes the book proxy object to be synchronized with its physical representation on the RFID tag. Notice that this write operation might not happen instantaneously because the RFID tag might be out of range for some time.

The following code snippet shows the function that is called after the user wrote a comment in the dialog box we described earlier:

```plaintext
1 deftype Book;
2 def books := ambient: Book;
3
4 whenEach: books<-getBookInfo()@Sustain becomes: { |infoAndRef|
5    GUI.addBookInfoAndReferenceToList(infoAndRef);
6 };
7
8 whenever: Book discovered: { |book|
10 }
```

\(^1\)A video demonstration of the application can be found at soft.vub.ac.be/amop/research/RFID
The dialog object passes the reference to the book and the user’s text as arguments to the function shown earlier. This addReviewToBook function asynchronously sends the addReview message to the book via the remote reference passed as an argument. The message is annotated with @Due(5 s) to indicate that if the message is not successfully processed after 5 s, a TimeoutException should be raised. The when:becomes:catch: observer installed on the future returned by the message sent can trigger two blocks. The becomes: block is triggered when the message was successfully processed by the proxy object, and in addition, the mutated data were successfully written to the physical RFID tag (because the addReview method is a mutator). As mentioned earlier, within the 5-s timeout period, the RFID tag might have moved in and out of range for several times, but the underlying implementation of the language constructs keeps attempting to write the data until this timeout period has passed. If the timeout period passed without that the review has been successfully written on the tag, the catch: block of the observer is invoked. In response, the user can try again, maybe after repositioning himself closer to the book.

When we look at the lines of code in this application, 291 lines are about dealing with specifying and managing proxy objects, managing communication with RFID tags, and reacting to different events (acknowledgments, tags moving in and out of range...), whereas the remaining 1161 lines of code deal with graphical user interfaces and application logic or are simple utility code. Even though this is a simple application, it shows that the complicated part is (in terms of code size) reasonably small compared with the total application code, thanks to a number of tailored language constructs.

4. IMPLEMENTATION

In this section, we detail the implementation of our RFID system. For our experiments, we used FEIG** ID ISC.PR101-USB 13.56 MHz proximity readers, high frequency desktop readers connected via USB. We used a variety of high frequency passive RFID tags, such as Philips I-Code1 and I-Code SLI tags (sticker-shaped), Philips MiFare Classic tags (contact cards), and Texas Instruments ISO tags (durable tags for harsh environments). Read range, the number of simultaneously detected tags, and storage size vary greatly among these types of tags.

In the remainder of this section, we explain the different layers of the implementation and then point out changes to the AmbientTalk language and interpreter and additional language constructs implemented as a library that are used in the implementation.

**RFID device driver**
At the lowest layer shown in Figure 8, we have a driver that allows to communicate with the RFID reader over a USB port. The driver provides only minimal functionality and provides operations to search available USB ports for the correct device, to open and close a given port and to read and write data (bytes) via the USB port.

**Java RFID library**
The Java RFID library implements vendor specifications for communicating with the RFID reader. The library interfaces with the driver using Java Native Interface. It provides classes such as Transponder and Device, which represent the RFID tags and reader device. The methods of these classes implement classic RFID operations such as inventory all tags in range, read data from a

```java

def addReviewToBook(book, text) {
    when: book<-addReview{text}@Due{5.seconds} becomes: {lack
        showKDialog("Review added successfully!");
    ) catch: TimeoutException using: {exc:
        showKDialog("Failed to add review!");
    });
}
```

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe
tag, or write data to a tag, and powering on and off the RF field of the reader. This is done by sending byte combinations via the driver to the RFID reader device. Note that the library (and driver) can be substituted by any custom built or vendor provided RFID library to support other RFID hardware.

**RFID event loop**

The RFID event loop keeps track of which tags are in range of the reader by continuously polling the RFID reader for an inventory. The RFID event loop does so by putting a request for an inventory in its own message queue by performing a message sent to itself. The effect is that polling requests are interleaved within the single thread of the RFID event loop with other operations the RFID reader needs to perform as messages from client applications enter the message queue. This way, race conditions in the execution of the event loop are prevented and operations are queued until the loop is ready, without requiring the overhead of an additional polling thread\(^{11}\). As per new inventory, the RFID event loop classifies the RFID tags based on their current and previous connectivity state. For tags that newly appear, a new proxy object is created using the data on the tag (or from another data source as detailed in Section R1.2). These new proxy objects are published to other connected event loops. All remote references to tags that were previously connected but are currently out of range are disconnected, as detailed in Section 4.1 next. For tags that reappear (tags that were seen before and for which a proxy object already exists) the corresponding proxy object is updated, and the remote references are reconnected as described in Section 4.1. As per connected RFID reader, one RFID event loop is needed. The RFID event loop requires a minimal API from the Java library, which is accessed using language symbiosis between AmbientTalk and Java. First, it relies on a class representing the RFID reader, which should implement a method to perform an inventory of the tags in range. This class should also implement a method that allows the RFID event loop to instruct the RFID reader to keep its RF field permanently active (to obtain exclusive access to RFID tag memory during interaction) or switch it off after each new request, as explained earlier in Section R6.1. Second, the presence of a class representing the RFID tags is assumed, which has a read and write method to retrieve and store data associated with a tag. The implementation of this interface in the Java library layer determines how the data are actually stored, as described in R1.2. As mentioned earlier, we have experimented both with serializing AmbientTalk objects on tag memory itself and using an external database on which the read and write operations are performed.

**Thing/application level**

The top level consists of applications such as the presented library application. Their implementation is agnostic to the way different RFID event loops are implemented. The logic of these applications is solely expressed in terms of the appearance/disappearance of and messages to physical things.

---

\(^{11}\)We assume that polling operations or keeping the RF field active are cheap in terms of power consumption, an assumption that is valid when, for example, using RFID-enabled smartphones running the Android OS, which optimizes the power consumption of the RFID antenna of the phone for such purposes. For example, for the Google Nexus S and Samsung Galaxy S the maximal power consumption is 0.55 W, which is about the same as the GSM modem, but less than the Wi-Fi module on similar phones [28].

4.1. Changes to the AmbientTalk language and virtual machine

In order to support all the constructs presented in this paper, the AmbientTalk language was slightly extended, and the existing virtual machine had to be adapted. In the communicating event loops model, an event loop forms the unit of distribution. Because of that, AmbientTalk only provides a primitive to take online and offline an event loop as a whole, which entails that either all of the published objects hosted by the event loop are made available or all of them are made unavailable. All remote references to these objects are marked connected and disconnected accordingly in their respective event loops. In the setting of mobile RFID-enabled applications, however, the RFID event loop hosting the RFID proxy objects has to be able to take online and offline proxy objects individually, because the connectivity of RFID tags varies individually. The alternative is spawning an event loop per scanned tag, which is clearly not scalable. To allow this, we extended the event loop model to enable a more fine-grained control over the connection status of objects. Event loops remain the unit of distribution, but in addition, it is possible to programmatically disconnect or reconnect a single published object. To this end, we introduced a new operation disconnect: that logically disconnects the published object it receives as an argument.

```plaintext
proxy.disconnection := disconnect: proxy;  // RFID tag leaves range
proxy.disconnection.reconnect();          // RFID tag reenters range
```

The preceding code shows the use of a disconnection object returned by disconnecting the proxy. Calling this object’s only method reconnect reestablishes the published object’s connection.

4.2. New language constructs on top of AmbientTalk

As described in Section 1.3, a multiway reference is a higher order reference that clusters references to different objects that are equivalent. The references therefore carry an equivalence relation to check whether two references are equivalent, and a priority that is used within the multiway reference to order the references and select the default path. To extend references with this meta information, we introduced property references [23]. A property reference is a construct to extend a reference to a published object with additional information. Publishing an object in AmbientTalk is performed using the export:as: operation, which takes the object to publish as a first argument and a type tag as its second argument. We extended this construct as such that a third argument (property) can be specified that will be locally available when a remote reference to the object is obtained. In the example next, we publish a proxy object with a type tag Book together with a property that contains two functions that implement the equivalence relation and the priority function (greater than) to support multiway references:

```plaintext
deftype Book;
def serialNr := proxy.serialNr;
def equivalent(proxyRef) { serialNr == proxyRef.serialNr };
def > (anyRemoteReference) { ... };
}
```

A multiway reference internally holds a message queue that is used to buffer messages in case all of the internal remote references it contains are disconnected. When a new remote reference is obtained that is deemed equivalent by the multiway reference, the remote reference is added to that multiway reference. If the reference matches no known equivalence relationships, a new multiway reference is constructed containing the remote reference as its only path. Upon acquiring a new remote reference or upon changes of the connection status of one of the remote references that belong to the multiway reference, the multiway reference will recompute the default path by means of its priority function.
4.3. Performance evaluation

In this section, we present the results of simulations that measure the computational overhead that comes with using our constructs for developing RFID-enabled applications. These simulations were ran on an Apple Macbook Pro laptop with an Intel Core i7 2.66-Ghz processor and 4 Gb of RAM running OSX version 10.6.7 and Java SE version 1.6.0. The version of AmbientTalk that we used is build 2.19.1. Our simulation is based on a simulated RFID reader device, that is, a Java object able to generate “dummy” transponder objects. These dummy transponder objects allow reading and writing their single data field that represents the data stored on the simulated tag as a byte sequence.

We conducted two simulations: one for read operations and one for write operations, which we implemented by using both our constructs as by using a plain Java implementation without taking into account the requirements postulated in the beginning of this paper. For the read operations, we let the simulated RFID reader perform 100 inventory operations, producing a collection of tag objects, and we read the data of every single tag object after each inventory operation. We vary the number of tags from 10 to 100 per inventory operation and measure how much time the complete simulation takes to finish. For the write operations, we again start from a collection of tag objects, but this time, we perform a write operation on every single tag and measure how long this simulation takes. For both operations, we also varied the amount of data read from and written to the tag objects, but this had very little effect on the results; so we omit these results. The results comparing the Java version and the version using our constructs are depicted by the two graphs shown next in Figure 9.

Without comparing both implementations, it is clear that both scale linearly when increasing the number of tags. When comparing both implementations, we can see that there is an overhead of a factor 10 to 20 depending on the operations performed. Although this seems excessively high, one must take into account that the version relying on our AmbientTalk constructs is generating, garbage collecting, and updating the connectivity status of proxy objects, (de)serializing complete AmbientTalk objects, enqueuing, and processing asynchronous message sends (sequentially) to provide fault tolerance, and so on. The basic Java implementation assumes that not a single fault occurs in these simulation runs. We measured where the biggest overhead in our implementation occurs and found that the creation of proxy objects, their serialization and deserialization, and the exporting and asynchronously discovering of references to these proxy objects contribute the most to the observed slowdowns.

A more interesting comparison is with the read rates (in tags per second) that modern RFID hardware can provide. Large industrial ultra-high frequency RFID readers offer in ideal conditions a read rate of 200 tags per second if 100% reliable reads are necessary. If faulty reads are tolerable, about 450 tags per second can be read [29]. As can be be seen from the two graphs shown next in Figure 10 (the left graph shows read rates whereas the right graph shows write rates), read and write rates do not decrease while increasing the number of tags. Additionally, our implementation is able to read (see left graph) slightly more than 200 tags per second. This means that in practice,
the performance gap between the two implementations will be barely noticed because of the delay caused by physical RFID hardware.

We conclude from these figures that our implementation can keep up with such a high-performance RFID reader and hence be applied to realistic systems, although we are targeting a different class of applications that consist of mobile, less performant devices. Finally, both the implementation of the AmbientTalk language as our RFID language constructs are unoptimized research artifacts that we expect to be further optimizable.

5. CONCLUSION

5.1. Limitations

The thread associated with each event loop consumes the incoming messages sequentially. This means that no objects are shared between different threads and race conditions cannot occur. However, when we consider RFID tags as an ambient environmental memory, it may very well be that a set of RFID tags is in the range of multiple users at the same time. Currently, we offer a way to guarantee that RFID tags are made invisible to other RFID devices when interacting with them (as discussed in Section R6.1), but this is recent work, and we are investigating more satisfactory solutions. Similarly, although not a critical problem, we observe that our implementation in AmbientTalk comes with a considerable computational overhead. Optimizing the implementation of AmbientTalk is ongoing work as more research in this field is performed.

A problem that we have not considered yet is security. Hence, our future work mainly lies in fulfilling the last requirement for mobile RFID-enabled applications. However, it must be noted that the severity of these problems highly depends on the used hardware and the setting. Because this landscape is quickly evolving, we are on the lookout for low-level protocols that can offer more security guarantees.

Another limitation is the limited amount of writable memory on passive RFID tags. We have tested our implementation using RFID tags with up to 8 kbits of writable memory. This means that we can only store very small serialized objects on the tags. On the other hand, the technology is progressing, and we can expect the storage on passive tags to steadily increase while the costs drop. This opens the door to use more standardized serialization formats as well, which we have not considered at the moment to cater for our prototype-based object model.

5.2. Contributions

Today, developing mobile RFID-enabled applications remains complicated because application developers have to deal manually with the hardware characteristics on a very low level in a general-purpose programming language. Current middleware is not suited to develop such applications (which require writing application-specific data on tags). On the other hand, lower level approaches
do not integrate the hardware characteristics into the heart of their programming model, introducing the complexity that we are trying to tackle. The abstractions presented in this paper integrate closely with the object-oriented message-passing paradigm, thereby aligning physical objects tagged with writable RFID tags with true mutable software objects. By implementing an example mobile RFID-enabled application, we have observed that the requirements that we set forward for programming mobile RFID-enabled applications are met in the following ways:

**Addressing physical objects.** The implementation of the application shows that mobile RFID-enabled applications can be written in an object-oriented fashion, where application-level proxy objects uniquely represent physical objects in one’s physical environment.

**Storing application-specific data on RFID tags.** The data needed to construct these proxy objects are stored on the RFID tags themselves.

**Reactivity to appearing and disappearing objects.** Application logic is expressed in terms of reactions to changes in the physical environment by relying on a number of expressive abstractions that are integrated into a communicating event loops framework.

**Asynchronous communication.** Interacting with physical objects is achieved by using the message-passing metaphor on the proxy objects, by means of asynchronous message passing and asynchronous signaling of return values.

**Fault-tolerant communication.** Communication failures are considered the rule rather than the exception. Failures that must be considered permanent are detected, and the appropriate exceptions are raised.

**Data consistency.** By allowing the mobile device equipped with an RFID reader to temporarily make its reachable tags invisible to other mobile devices (by keeping them powered in the RF field), exclusive access for that mobile device can be granted and data consistency guaranteed, in combination with the consistency guarantees of the event loop model among local applications for the device. For the moment, we have not considered security, and this remains future work.

### ACKNOWLEDGEMENTS

Andoni Lombide Carreton is funded by a doctoral scholarship of the “Institute for the Promotion of Innovation through Science and Technology in Flanders” (IWT Vlaanderen).

### REFERENCES

7. Oracle (Sun Developer Network). Developing auto-ID solutions using Sun Java system RFID software.


