Distributed Object-Oriented Programming with RFID Technology

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

Software Languages Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussels, Belgium
{alombide,kpinte,wdmeuter}@vub.ac.be

Abstract. Our everyday environments will soon be pervaded with 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. In this paper, we propose extending 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.

Key words: RFID, pervasive computing, ambient-oriented programming, mobile RFID-enabled applications

1 Introduction

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 are cheap, writable passive tags and RFID readers integrated into mobile devices (such as smartphones).

* Funded by a doctoral scholarship of the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

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 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.

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 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.

R2: Storing application-specific data on RFID tags. Since mobile RFID-enabled applications do not rely on a backend database, the data on the RFID tags should be self-contained and stored on the writable memory of the tags [3].

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 due hardware phenomena.

The remainder of this paper 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. Section 4 discusses the limitations of our system. Finally, section 5 concludes this paper.

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.

RFID 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 Accada [4] and Aspire RFID [5]. 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.

RFID 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, current RFID middleware is not suited to develop mobile RFID-enabled applications.

RFID in Pervasive Computing In [6], mobile robots carrying an RFID reader guide visually impaired users by reading RFID tags that are associated with a certain location. In [7] 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 [8] 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 above 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 [9] 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 represent physical objects as there is no way 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* [10] 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 constructs that satisfy all requirements listed in section 1. 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 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.

3.2 Ambient-Oriented Programming with RFID Tags

In the mobile RFID-enabled application introduced in the previous section, mobile devices hosting different instances of 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 properties are very similar to the the ones exhibited by distributed applications in mobile ad hoc networks [11]. Similar to mobile devices in mobile ad hoc networks RFID tags and readers should interact spontaneously when their ranges overlap.

Ambient-oriented programming [12] 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 definition along with the object. This leads to consistency problems and performance issues [13], [14]. Hence, a first characteristic of ambient-oriented programming is the usage of a *classless object model*. A second characteristic is the use of *non-blocking communication primitives*. With blocking communication a

program will wait for the reply to a remote computation causing the the application to block whenever a communication partner is unavailable [15]. 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. Since 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 [16]. 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 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. When mobile devices move out

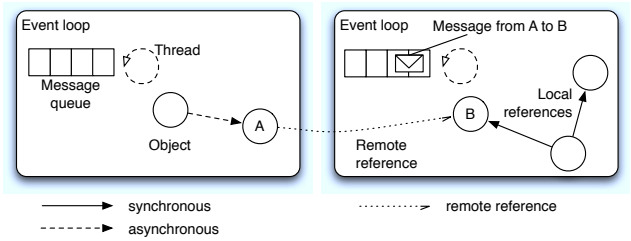

Fig. 1. Overview of the communicating event loops model.

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 [17]. It is conceived as a scripting language that eases the composition of distributed Java components in mobile ad hoc networks. 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 mu-

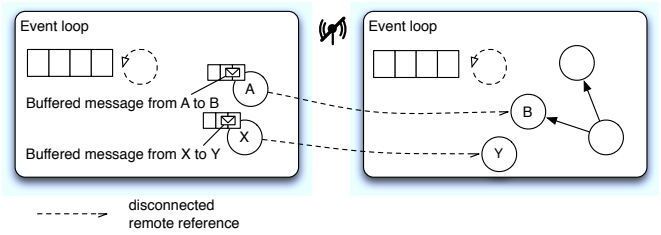

Fig. 2. Messages to disconnected objects are buffered until reconnection.

table 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 below. It contains slots for the ISBN, title and reviews and provides two *mutator* methods to update the book's title and add reviews:

```
1  deftype Book;
2  def aBook := object: {
3    def ISBN := 123;
4    def title := "My Book";
5    def reviews := Vector.new();
6
7    def setTitle(newTitle)@Mutator {
8      title := newTitle;
9    };
10
11   def addReview(review)@Mutator {
12     reviews.add(review);
13   };
14 } 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. Since 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 above 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 has 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[1]. 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*. 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 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. Figure 3 shows a general overview of the RFID system.

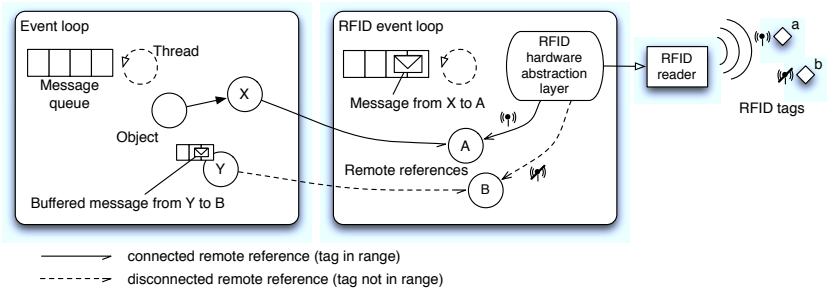

Fig. 3. Overview of the RFID event loop.

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 below shows how a blank tag is initialized as a book object:

```
when: tag<-initialize(aBook) becomes: {|book| ...};
```

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. 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

[1] AmbientTalk is a highly dynamic programming language which makes it impossible to determine from the source code if mutating operations are going to be invoked. Annotations and type tags are the same data type and are programmer-defined.

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.

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 below, 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).

```
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.

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 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 cause 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:

```
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 `"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 send with a duration that controls how long a message is buffered before timing out. For example, we can add short reviews to a book:

```
def myReview := "not suitable for beginners";
when: book<-addReview(myReview)@Due(10.seconds) becomes: {|ack|
  // 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 seconds. 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 second timeframe, the exception is signaled to client applications and their registered `catch`-blocks are invoked.

3.3 Addressing Specific Groups of RFID-tagged Objects

As mentioned in section 2, RFID tags are typically used in large quantities, e.g. in warehouse applications. In mobile RFID-enabled applications it is often necessary to address a specific group of objects. E.g. 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 due to 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 [18]. 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, e.g. the set of nearby books. This set is updated behind the scenes when books move in and out of range. The example below 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 below where books are selected based on their category field:

```
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 one book about this topic in range for its shelf:

```
def shelfFuture := computerScienceBooks<-getShelf()@Any;
when: shelfFuture becomes: { |shelf|
  system.println("The book should be on shelf: " + shelf);
};
computerScienceBooks<-setShelf("5D")@Sustain;
```

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. First of all, while the user moves about in the library, the list of nearby books has to be updated. The following code snippet shows this:

```
1 deftype Book;
2 def books := ambient: Book;
3
4 whenEach: books<-getBookInfo()@Sustain becomes: {|infoAndRef|
5   GUI.addBookInfoAndReferenceToList(infoAndRef);
6 };
7 whenever: Book discovered: {|book|
8   whenever: book disconnected: { GUI.removeBookFromList(book) };
9 };
```

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*, i.e. 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 7, 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. This is illustrated by the code snippet below. 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 above:

```
1 def addReviewToBook(book, text) {
2   when: book<-addReview(text)@Due(5.seconds) becomes: {|ack|
3     showOkDialog("Review added succesfully!");
4   } catch: TimeoutException using: {|exc|
5     showWarningDialog("Failed to add review!");
6   }};
```

The dialog object passes the reference to the book and the user's text as arguments to the function shown above. 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.seconds)` to indicate that if the message is not successfully processed after 5 seconds, a `TimeoutException` should be raised. The `when:becomes:catch:` observer installed on the future returned by the message send 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 was successfully written to the physical RFID tag (since the `addReview` method is a mutator). As mentioned earlier, within the 5 second 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. This block simply shows a dialog box that notifies the user that adding the review failed. In response, the user can try again, maybe after repositioning himself closer to the book.

4 Limitations and Future Work

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. When these users concurrently update the same tag from different devices, distributed race conditions on that tag may occur. In our experiments we have employed passive RFID tags that can only be powered upon communication. This means that here is no way of locking the RFID tag for a limited amount of time.

Another limitation of using this type of tags is that currently, they offer only a very limited amounts of writable memory. 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. As a way to circumvent these limitations we are currently experimenting with active RFID tags. These tags are battery-powered and can keep on running independently from the readers. This means that they can store more data and can execute code, which opens up opportunities for solving the problems mentioned above when more expensive tags can be used, and in addition may lead us in new research directions.

5 Conclusion

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 are 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 is 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 raise the appropriate exceptions.

References

1. V. Waller and R. B. Johnston, "Making ubiquitous computing available," *Commun. ACM*, vol. 52, no. 10, pp. 127–130, 2009.
2. J. Bleecker, "A manifesto for networked objects — cohabiting with pigeons, arphids and aibos in the internet of things," 2006.

3. G. Roussos and V. Kostakos, "RFID in pervasive computing: State-of-the-art and outlook," *Pervasive Mob. Comput.*, vol. 5, no. 1, pp. 110–131, 2009.
4. C. Floerkemeier, C. Roduner, and M. Lampe, "RFID Application Development with the Accada Middleware Platform," *IEEE Systems Journal, Special Issue on RFID Technology*, vol. 1, pp. 82–94, Dec. 2007.
5. N. Kefalakis, N. Leontiadis, J. Soldatos, K. Gama, and D. Donsez, "Supply chain management and NFC picking demonstrations using the AspireRfid middleware platform," in *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, (New York, NY, USA), pp. 66–69, ACM, 2008.
6. V. Kulyukin, C. Gharpure, J. Nicholson, and S. Pavithran, "RFID in robot-assisted indoor navigation for the visually impaired," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 2, pp. 1979–1984 vol.2, 2004.
7. M. Philipose, K. P. Fishkin, M. Perkowitz, D. J. Patterson, D. Fox, H. Kautz, and D. Hahnel, "Inferring activities from interactions with objects," *IEEE Pervasive Computing*, vol. 3, no. 4, pp. 50–57, 2004.
8. M. Fleck, M. Frid, E. O'Brien-Strain, T. Kindberg, R. Rajani, and M. Spasojevic, "From informing to remembering: Deploying a ubiquitous system in an interactive science museum," in *Interactive Science Museum, IEEE Pervasive Computing Magazine, April-June*, pp. 13–21, 2002.
9. M. Mamei, R. Quaglieri, and F. Zambonelli, "Making tuple spaces physical with rfid tags," in *Symposium on Applied computing*, (New York, NY, USA), pp. 434–439, ACM, 2006.
10. M. J. Carey and D. J. DeWitt, "Of objects and databases: A decade of turmoil," in *22th International Conference on Very Large Data Bases*, (San Francisco, CA, USA), pp. 3–14, Morgan Kaufmann Publishers Inc., 1996.
11. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-oriented programming," in *20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, (New York, NY, USA), pp. 31–40, ACM, 2005.
12. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-oriented programming in ambienttalk," in *20th European Conference on Object-Oriented Programming*, pp. 230–254, 2006.
13. D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle, "Organizing programs without classes," *Lisp Symb. Comput.*, vol. 4, no. 3, pp. 223–242, 1991.
14. J. Dedecker and W. D. Meuter, "Using the prototype-based programming paradigm for structuring mobile applications," 2002.
15. A. L. Murphy, G. P. Picco, and G.-C. Roman, "Lime: A middleware for physical and logical mobility," in *21st International Conference on Distributed Computing Systems*, (Washington, DC, USA), p. 524, IEEE Computer Society, 2001.
16. M. Miller, E. D. Tribble, and J. Shapiro, "Concurrency among strangers: Programming in E as plan coordination," in *Symposium on Trustworthy Global Computing*, vol. 3705 of *LNCS*, pp. 195–229, Springer, April 2005.
17. T. V. Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. D. Meuter, "Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks," in *XXVI International Conference of the Chilean Society of Computer Science*, (Washington, DC, USA), pp. 3–12, IEEE Computer Society, 2007.
18. T. Van Cutsem, J. Dedecker, S. Mostinckx, E. Gonzalez, T. D'Hondt, and W. De Meuter, "Ambient references: addressing objects in mobile networks," in *21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, (New York, NY, USA), pp. 986–997, ACM, 2006.