Electronic Communications of the EASST
Volume 19 (2009)

Proceedings of the Second International DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2009)

Language Abstractions for RFID Technology

Andoni Lombide Carreton, Kevin Pinte, Wolfgang De Meuter

13 pages

Guest Editors: Romain Rouvoy, Michael Wagner
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Language Abstractions for RFID Technology

Andoni Lombide Carreton[1], **Kevin Pinte**[2], **Wolfgang De Meuter**[3]

[1] alombide@vub.ac.be, [2] kpinte@vub.ac.be, [3] wdmeuter@vub.ac.be

Programming Technology Lab
Vrije Universiteit Brussel, Belgium

Abstract: Developing pervasive and context-aware applications that make use of RFID technology is a daunting task given the high degree of failures inherent to communication with RFID tags. The reason is that current programming models do not incorporate these failures into the very heart of their computational model. AmbientTalk, a research language aimed at pervasive applications running in mobile ad hoc networks, does offer such a programming model, but it is aimed at mobile devices interconnected via peer-to-peer network connections such as WiFi or Bluetooth. In this paper we show how we use this programming model for the communication with RFID tags.

Keywords: Pervasive Computing, Distributed Computing, RFID tags, AmbientTalk

1 Introduction

To enable pervasive, context-aware applications that are smoothly integrated in the everyday environment, these applications need contextual information about their environment, such as a representation of the objects in their immediate surroundings. One of the upcoming technologies that makes this possible is Radio Frequency Identification. RFID tags are tiny, disposable chips that can be placed on everyday objects, both enhancing them with digital information and providing a digital representation of these objects to context-aware applications [Ble06]. RFID technology comes in many varieties, but the most common use is passive RFID technology because of the low cost of passive tags. These passive tags are tiny, cheap and disposable chips that use the current generated by an incoming radio wave to power up their circuits and broadcast a reply into the environment using radio waves. Most RFID tags do not only provide some identification, but also some additional EEPROM memory that can be used to write data on them, again by using radio waves. The data on the chips can then be read by devices called RFID readers. These RFID readers typically offer some low level API that allows querying the environment for all nearby tags, get the information from a tag with a certain identification, write data onto a tag with a certain identification etc. These operations frequently fail because of the hardware limitations of RFID technology: tags that are close to each other can interfere with each other, tags can move out of range of the RFID reader while it is still communicating with the tag, the lookup of a tag with a certain identification may fail because the tag is not there anymore etc. These failures may be permanent, but it may very well be that at a later moment the operation would succeed because of minimal changes in the physical environment (e.g. a tag moved back in range or suffers less from interference). Current approaches signal these failures as errors to the client

code, which causes the client code to be polluted with error handling code and to be structured in very awkward ways to deal with these errors. The reason is that current languages and the middleware approaches based on these languages do not incorporate these failures explicitly into the very heart of their computational model. The disconnection with a RFID tag should not be regarded as an exceptional case, but as a phenomenon that can occur with any operation invoked on the tag.

With current approaches, one has to either use middleware that is specifically designed for a certain class of applications that work with RFID tags (e.g. warehouse applications, product tracking etc.) [FRL07, KLS+08]. These types of middleware platforms focus on applications where all the data is stored in a database and is looked up by the electronic product codes on the tags and are clearly not designed for the development of general pervasive applications where it should be possible to store any kind of data on the tags themselves. The alternative is using a lower level library that allows the interaction with a RFID reader. Here the downside is that such a library has to be used in a programming language that has no fault tolerance constructs tailored towards RFID technology.

These characteristics are very reminiscent to the hardware characteristics of mobile ad hoc networks, where the different devices are also interconnected with unreliable connections, which we will name *volatile connections*. In this paper, we apply the ambient-oriented programming paradigm designed for such mobile ad hoc networks to the communication with RFID tags from within a pervasive computing context. In this paradigm, network failures are rather the rule than the exception. This way, the volatile connection metaphor is used for the connection with RFID tags and the programming model geared towards communication over such volatile connections is used for the communication with RFID tags.

In the next section, we describe a programming paradigm geared towards pervasive applications in mobile had hoc networks and a concrete ambient-oriented programming language called AmbientTalk. In Section 3, we explain how we extended AmbientTalk to use its programming model to communicate with RFID tags expressively and indicate some future work. The last section concludes this paper.

2 Ambient-Oriented Programming

The ambient-oriented programming paradigm [DVM+05] is specifically aimed at pervasive applications. Hence, it seems interesting to integrate support for RFID technology in an ambient-oriented programming language. Ambient-oriented programming languages should explicitly incorporate potential network failures in the very heart of their computational model. Therefore, communication between distributed application components should happen without blocking the execution thread of the different components such that devices may continue doing useful work even when the connection with a communication partner is lost. This implies that there is a natural form of concurrency among the distributed application components. Ambient-oriented languages should also deal with the dynamically changing network topology in mobile ad hoc networks. The fact that in such networks devices spontaneously join with and disjoin from the network means that the services these devices host cannot be discovered using URLs or other extensional service descriptions. Instead, services have to be discovered dynamically in the en-

vironment using intensional service descriptions in order to support roaming.

2.1 AmbientTalk

AmbientTalk [VMG+07, VMD08] is a distributed programming language embedded in Java[1]. The language is designed as a distributed scripting language that can be used to compose Java components which are distributed across a mobile ad hoc network. The language is developed on top of the J2ME platform and runs on handheld devices such as smart phones and PDAs. Even though AmbientTalk is embedded in Java, it is a separate programming language. The embedding ensures that AmbientTalk applications can access Java objects running in the same JVM. These Java objects can also call back on AmbientTalk objects as if these were plain Java objects. AmbientTalk offers direct support for the different characteristics of the ambient-oriented programming paradigm described above.

1. In an ad hoc network, objects must be able to discover one another without any infrastructure (such as a shared naming registry). Therefore, AmbientTalk has a service discovery engine that allows objects to discover one another in a peer-to-peer manner. Java interfaces act as the common pieces of information by means of which objects are advertised and discovered.

2. In an ad hoc network, objects may frequently disconnect and reconnect because of network partitions. Therefore, AmbientTalk provides fault-tolerant asynchronous message passing between objects: if a message is sent to a disconnected object, the message is buffered and resent later, when the object becomes reconnected. Other advantages of asynchronous message passing over standard RPC is that the asynchrony hides latency and that it keeps the application responsive (i.e. the event loop is not blocked during remote communication and is free to process other events).

2.2 Asynchronous Message Passing

The most important difference between AmbientTalk and Java is the way in which they deal with concurrency and network programming. In AmbientTalk, concurrency is not spawned by means of threads but rather by means of actors [AH87]. AmbientTalk actors are not represented as active objects, but rather as *communicating event loops*, as is done in the E programming language [MTS05]. An actor is an event loop encapsulating regular objects which can communicate with one another using either synchronous method invocations (expressed as `o.m()`) or asynchronous message passing (expressed as `o<-m()`). Asynchronous messages are enqueued in an actor's queue of incoming messages, called its *mailbox*. An actor perpetually removes the next message from its mailbox and executes the corresponding method on the receiver of the message. Actors process messages from their message queue serially, i.e. one by one, to avoid race conditions on the state of regular objects.

In AmbientTalk, each object is said to be *owned* by exactly one actor. Only an object's owning actor may directly execute one of its methods. It is possible for objects owned by an actor

[1] The language is available at prog.vub.ac.be/amop

to refer to objects owned by other actors. Such references that span different actors are named *far references* (the terminology stems from E [MTS05]) and only allow asynchronous access to the referenced object. Performing a method invocation via a far reference provokes a runtime exception. Asynchronous messages sent via far references are enqueued in the message queue of the actor that encapsulates the receiver object. Figure 1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the event loop processes of the actors which perpetually take messages from their message queue and synchronously execute the corresponding methods on the actor's owned objects. An event loop process never "escapes" its actor boundary. When communication with an object in another actor is required, a message is sent asynchronously via a far reference to the object. For example, when `A` sends a message to `B`, the message is enqueued in the message queue of `B`'s actor which eventually processes it.

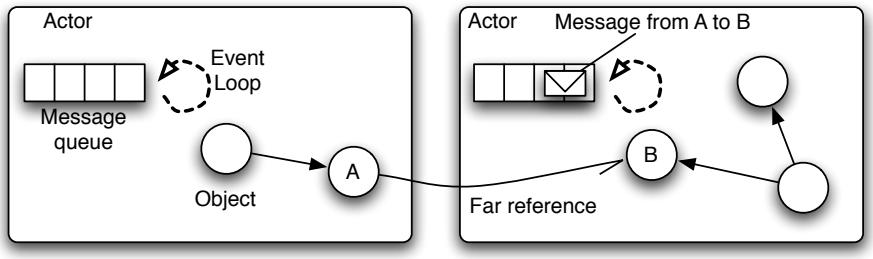

Figure 1: AmbientTalk actors as event loops

In AmbientTalk, asynchronous messages can be sent between objects owned by the same actor (via a local reference) or by different actors (via a far reference). When sending an asynchronous message to an object that is encapsulated within the same actor, the message's parameters are passed *by reference*, exactly as is the case with regular synchronous message sending. When sending a message across a far reference, objects are instead parameter-passed *by far reference*: the parameters of the invoked method are replaced by far references to the original objects.

2.3 Far References and Partial Failures

In AmbientTalk, two objects are said to be *local* when they are owned by the same actor. Objects are considered *remote* when they are owned by different actors, even if those actors are hosted by the same device. By design, AmbientTalk abstracts from the physical location of actors and considers actors as the unit of distribution. Because objects residing on different devices are necessarily owned by different actors, the only kinds of object references that can span across different devices are far references. This ensures that all distributed communication is asynchronous.

By allowing far references to cross virtual machine boundaries, we must specify their semantics in the face of partial failures. AmbientTalk's far references are by default resilient to network disconnections. When a network failure occurs, a far reference to a disconnected object starts buffering all messages sent to it. When the network partition is restored at a later point in time, the far reference flushes all accumulated messages to the remote object in the same order as they

were originally sent. Hence, messages sent to far references are never lost, regardless of the internal connection state of the reference. Making far references resilient to network failures by default is one of the key design decisions that make AmbientTalk's distribution model suitable for mobile ad hoc networks, because temporary network failures have no immediate impact on the application's control flow.

However, not all network partitions are transient. Some of these failures will be permanent (e.g. a device moving out of wireless communication range that never returns) and require application-level failure handling. To preserve the resilience of far references to transient failures while still being able to deal with permanent failures, AmbientTalk employs *leasing* [GC89]. A *lease* denotes the right to access a resource (e.g. an object) for a finite amount of time. At the discretion of the owner of the resource a lease can be renewed, prolonging access to the resource. In AmbientTalk, we chose to represent leases as a special kind of far references which we name *leased* far references. A leased far reference behaves like a far reference, except that it grants access to its service object only for a limited period of time. Because we chose to model leasing by means of a special kind of far reference, the client can use the leased reference as if it were the remote object itself, making the use of leasing transparent to the client. AmbientTalk incoporates a leased far reference variant which transparently adapt their lease period under certain circumstances. As long as the leased far reference is actively being used (i.e. messages are sent via the reference to the remote object) *and* there is a network connection, its lease is transparently renewed, meaning that the leased far reference remains usable for an extended period of time. With this variant tedious boilerplate renewal code is avoided at client as well as at server-side.

2.4 Exporting Objects as Services

Objects can acquire far references to objects by means of parameter-passing or return values from inter-actor message sends. However, it remains to be explained how objects can acquire an *initial* far reference to an object owned by a remote actor. In order to make some objects available to remote actors and their objects, an actor can explicitly *export* objects that represent certain services. In most distributed systems, exported objects are identified by means of a simple name or UUID in a name server or by a URL. However, in a mobile ad hoc network, name servers are impractical due to the limited infrastructure and the URL of a service may not be known to other actors.

In AmbientTalk, service objects are exported by means of a *type tag*. Type tags are a lightweight classification mechanism, used to categorise objects explicitly by means of a nominal type. One use of type tags in AmbientTalk is to provide an intensional description of what kinds of services an object provides to remote objects. In AmbientTalk, a type tag can be a subtype of one or more other type tags, and one object may be tagged with multiple type tags[2].

[2] Although type tags are not used for static type checking, they are best compared with empty Java interface types, like the typical "marker" interfaces used to merely tag objects (e.g. `java.io.Serializable` and `java.lang.Cloneable`.). One assumption we make is that all devices in the network attribute the same *meaning* to each type tag, i.e. we assume they have a common ontology to classify their services.

2.5 Ambient References

When writing AmbientTalk code to query nearby services for data (e.g. all nearby temperature sensors in a wireless sensor network) using the language features discussed above, one often writes a recurring pattern of code to deal with the discovery and loss of nearby services while a query is executing, and to deal with gathering the replies from all respondent services. To ease the writing of multicast queries in AmbientTalk, AmbientTalk offers a language construct named ambient references [Van08].

Ambient references represent a collection of nearby services of the same type. This collection is constantly kept up-to-date with the proximate physical environment: newly discovered services are added to the collection, while unresponsive services are removed from it. This synchronisation with the environment must no longer be done manually by the programmer, but is instead done by the ambient reference itself.

Sending a message to an ambient reference causes this message to be multicast to all services in the collection. A message can also be annotated with an expiration period (in milliseconds). If a message has an expiration period, it will not only be multicast to all services in the ambient reference's collection at the time the message is sent, but also to any services discovered at a later point in time, until its expiration period has elapsed. Consider the following example query:

```
def sensors := ambient: TemperatureSensor;
whenAll: sensors<-getTemp()@Expires(5*1000) resolved: {
  |temperatures|
  // process the sensed temperature values
}
```

The keyword `ambient:` allows one to create an ambient reference given a Java interface. The variable `sensors` contains an ambient reference that refers to all nearby `TemperatureSensor` services. The message `getTemp()` is asynchronously multicast to these services with an expiration period of 5 seconds. This implies that the message may be received by all proximate sensors at the time it is sent, as well as to all additional sensors discovered within the next 5 seconds.

The `whenAll:becomes:` control structure allows the programmer to install an event handler that can be used to gather the results of the query. Within this event handler, `temperatures` refers to an array containing the readouts of the sensors that replied. The event handler is triggered when the message's expiration period has elapsed. Ambient references support only weak delivery guarantees: some sensors may not have received the `getTemp()` message, and some replies to the message may have gotten lost or may arrive too late, in which case they are discarded.

3 Language Abstractions for RFID Technology

In the previous section we have shown a number of polished language constructs geared towards communicating with remote objects via volatile network connections. RFID tags can be regarded as an extreme case of remote objects that may become unreachable at any point in time. The motivation for using the language level to provide the necessary abstractions is that by doing this on the language level, expressive constructs can be designed that do not require the programmer to write recurring patterns or boiler plate code to use the abstractions. Furthermore, by in our

case integrating the abstractions in an event loop concurrency model, the programmer does not have to be aware of issues such as thread synchronization, data races, etc. In this section, we show how we apply the AmbientTalk language constructs discussed in the previous section to RFID tags.

3.1 Asynchronous Communication with RFID Tags

As explained in Subsection 2.2, communication between remote objects can only happen asynchronously in AmbientTalk. We want to do the same thing with RFID tags, e.g. model them as remote objects of which the interface can only be invoked asynchronously. This way, client code that sent a message to some tag will not remain blocked until the message is processed. Furthermore, by using this communication model, we provide fault tolerant message passing to RFID tags.

We have created a special AmbientTalk actor which acts as an abstraction over a RFID reader. The underlying implementation makes sure that whenever new tags are detected, they are added as RFID objects to the RFID reader actor. RFID objects have a well-defined interface that allows remote parties to interact with the tags (e.g. read the data on them, write data on them, protocol-specific operations of the tag etc.). These are low-level RFID reader-specific operations and are only triggered when the corresponding message has arrived on their host actor, meaning there is a connection with the tag on which the operation has to be invoked.

Of course, AmbientTalk provides some additional constructs to deal with disconnections, such as leasing, which we explained in Subsection 2.3. The point is that by encapsulating RFID tags into their own actor, we can use AmbientTalk's fault tolerant communication model to communicate with these tags. Furthermore, because we rely entirely on this communication model, constructs that are built on this communication model also work with RFID tags. One example are synchronization techniques such as *futures*. When an asynchronous message is sent, a place holder object for the result that is being computed asynchronously on another device is being returned. Such a place holder is called a future. Futures can be used to register event handlers on them that get executed as soon as the result of the asynchronous invocation is available. An example is shown below of such a **`when:becomes:`** event handler registered on a future.

```
when: rfidTag<-getSerialNr() becomes: { |serialNr|
  // Execute code with the serialNr of the tag
}
```

3.2 Discovering RFID Tags

In the previous subsection we used far references to RFID tags to communicate with the tags in a fault tolerant way. But how does one obtain such a far reference to a tag? For this, we rely entirely on the service discovery mechanism of AmbientTalk that we discussed in Subsection 2.4. The actor that encapsulates the RFID tags makes sure that when new tags are discovered, they are immediately exported into the network. Other actors can then obtain a reference to these tags as follows:

```
when: RfidTagType discovered: { |tag|
```

```
  // Do something with tag
}
```

In this example, `tag` denotes a far reference which can be used to communicate asynchronously with the RFID tag. Additionally, one can also be notified of a tag to which the connection is lost:

```
when: tag disconnected: {
  // Do something in response to the disappearance of tag
}
```

This event is signalled by the RFID reader actor when it detects that a tag has become unreachable. As mentioned in Subsection 2.4, discovery of remote objects happens via Java interfaces, which are represented as types in AmbientTalk. With a simple extension however, more fine grained service discovery can be achieved. One could store the type of a tag on the tag itself. When this tag is scanned, the RFID reader actor can then export the tag as of the type stored on the tag. This way, the programmer is able to provide service discovery that only triggers on certain types of tags. In the next section, we show an even more advanced communication construct.

3.3 Communicating with Groups of RFID Tags

RFID tags typically are used in large quantities, e.g. in warehouse applications. In such situations, it is often important to communicate with a group of related tags, 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 is highly volatile because of the volatile connections with the RFID tags. At any point in time, tags move out of range and new tags move in range. Simply iterating over such a collection would require lots of manual error handling. Furthermore, one should manually encode the updates to this collection when new tags move in range, possibly causing synchronization errors.

Designating a group of objects is done in AmbientTalk using ambient references, which are explained in Subsection 2.5. In the example below, we assume that the RFID reader actor broadcasts property objects for each scanned tag that can be used to check certain properties of the tags, in this case the product it represents. We also assume that the tag objects understand a `atKey:putValue:` message that stores or updates a key-value pair on the tag.

```
def tagsInRange := ambient: RfidTagType
  where: { |tag| tag.product == productToUpdate };

tagsInRange<-atKey: "price" putValue: newPrice @ [ All, Sustain ];
```

The `All` and `Sustain` annotations on the message shown above are used to tell the ambient reference how and to which objects the message has to be sent. `All` means sending the message to all objects in range that match the intensional service description of the ambient reference. `Sustain` means that the message will be perpetually sent to newly discovered objects in the network that also match this service description. The ambient reference designates a group of tags that is behind the scenes constantly kept up to date with the tags that are physically in range.

3.4 Implementation

The implementation is conceived as a layered architecture, where the lower layers are hardware-dependent and the higher layers provide an interface that the lower layers can use to interact with the AmbientTalk layer. At the lowest level, there is a driver that is used to handle the low-level communication with the RFID device. On top of that there is a reader-specific layer written in Java which implements all the RFID reader device commands (e.g. scanning for tags, obtaining a connection to a tag etc.) and provides a representation of RFID tags with a well-defined interface (e.g. reading and writing to tags).

The layer on top uses this layer to scan for tags and transform the information gathered from scans into the appropriate events, e.g. a new tag has been discovered, a tag has disconnected, a known tag has reconnected etc. AmbientTalk is conceived as a scripting language to make Java components work together in mobile ad hoc networks. Therefore, AmbientTalk supports symbiosis with its host language Java: AmbientTalk methods can be called from within Java and Java methods can be called from within AmbientTalk. Furthermore, both AmbientTalk and Java objects are automatically coerced such that they can be used in both languages. This allows this layer to call the necessary callbacks on the AmbientTalk actor level when RFID events are detected and allows the RFID tag objects generated by the reader-specific layer described above to be passed to the AmbientTalk layer.

Finally, the AmbientTalk layer is an AmbientTalk actor that provides some callbacks that should be called by the layer below, as described above. These callbacks will export (or retract) the RFID tag objects that were passed by the lower layer. This way, client actors can easily be notified of tags moving in and out of range and obtain a far reference to these tags using the language constructs already offered by AmbientTalk.

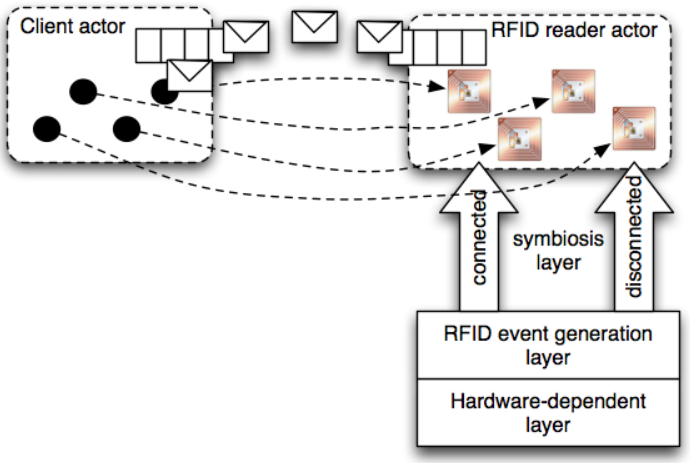

Figure 2: Overview of the architecture of the implementation

3.5 Future Work

The work presented in this paper mainly focuses on the communication between RFID tags and pervasive, context-aware applications. We currently employ ad hoc methods to represent the data on the RFID tags (e.g. raw strings, key-value pairs...). In the next step, we will investigate more advanced techniques for storing data on the tags, such as storing serialized AmbientTalk objects. One important benefit is that this not only allows the programmer to interact with the stored data via the API provided by the representation of RFID tags in the language, but also directly with the objects stored on the tags. This requires deserializing the objects on the RFID tags by the RFID device actor and exporting and retracting from the network the deserialized objects individually. There is some related work in this field, in [MQZ06] tuples are used to store information on the tags and the information of all tags in range is organized into a tuple space.

In actor-based languages such as AmbientTalk, incoming messages are serialized in a queue and executed sequentially by the actor's single thread. This prevents race conditions since no data is shared between different actors. However, it may also introduce a bottleneck in implementations such as the one presented in this paper. Messages sent to RFID tag objects in a single RFID reader actor are forwarded one by one to the tag objects. If lots of messages are waiting to be processed, the connection with some of the destination RFID tags may be lost while some of the messages may still have to be forwarded to these tags. Currently, we signal an exception that is raised on the sender of the message that could not be forwarded, but we feel that there is room for optimization here.

To remain responsive, pervasive applications running on mobile ad hoc networks have to be conceived as event-driven architectures because of the dynamic nature of both the underlying network and the applications running on top of them [KB02, MC02, MPR01, Gri04]. Clearly this is also the case for AmbientTalk applications: the reception of replies from asynchronous messages and the connection and disconnection of devices are modeled as events. By adopting such an event-driven architecture, the application logic becomes scattered over different event handlers or callbacks which may be triggered independently [CM06]. The control of the application is no longer driven by an explicit control flow determined by the programmer, but by external events. This is a phenomenon known as *inversion of control* [HO06]. Control flow among event handlers has to be expressed implicitly through manipulation of shared state. This is why in complex systems such an event-driven architecture can become hard to develop, understand and maintain [LC02, KR05]. Since RFID tags typically are used in large quantities, these problems become substantial. For this reason we are going to integrate the work presented in this paper with a special variant of the AmbientTalk interpreter that supports programming techniques that prevent an inversion of control and on top of which we implemented distributed event processing constructs geared towards mobile ad hoc networks.

4 Conclusion

The development of pervasive, context-aware applications that make use of RFID technology is impeded by the complexity of interacting in a fault tolerant way with RFID tags. The set of RFID tags currently in range of an application is a very volatile set. The inherent complexity of

interacting with such a set is reflected in code written in a programming language that does not incorporate network failures into the heart of its computational model. We have extended the ambient-oriented language AmbientTalk with support for RFID technology in such a way that the original programming model of the language, which is designed for interacting with volatile sets of remote objects, is applied for the communication with RFID tags. RFID tags are in this case nothing more than a special kind of remote objects that provide an interface that allows client code to interact with the tags. Concretely, the programming model we have described here offers:

- Fault tolerant message sending to RFID tags by adopting an asynchronous communication model,
- An event-driven architecture to notify AmbientTalk applications of the appearance and disappearance of tags in the environment,
- Communication with groups of tags which are defined by means of intensional descriptions and are dynamically kept up to date with the tags present in the environment.

We believe that dealing with some of the hardware characteristics of RFID technology on the language level (by offering polished language constructs as the ones presented in this paper) can significantly reduce the complexity of developing pervasive applications that interact with RFID tags.

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

Bibliography

[AH87] G. Agha, C. Hewitt. Concurrent programming using actors. *Object-oriented concurrent programming*, pp. 37–53, 1987.

[Ble06] J. Bleecker. A Manifesto for Networked Objects — Cohabiting with Pigeons, Arphids and Aibos in the Internet of Things. 2006. http://research.techkwondo.com/blog/julian/185

[CM06] B. Chin, T. Millstein. Responders: Language support for interactive applications. In *ECOOP*. Nantes, France, July 2006.

[DVM+05] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter. Ambient-Oriented Programming. In *OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, 2005.

[FRL07] C. Floerkemeier, C. Roduner, M. Lampe. RFID Application Development With the Accada Middleware Platform. *IEEE Systems Journal* 1:13, November 2007.

[GC89] C. Gray, D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*. Pp. 202–210. ACM, New York, NY, USA, 1989. doi:http://doi.acm.org/10.1145/74850.74870

[Gri04] R. Grimm. One.world: Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing* 3(3):22–30, 2004. doi:http://dx.doi.org/10.1109/MPRV.2004.1321024

[HO06] P. Haller, M. Odersky. Event-Based Programming without Inversion of Control. In *Proc. Joint Modular Languages Conference*. Lecture Notes in Computer Science 4228, pp. 4–22. Springer, 2006.

[KB02] A. Kaminsky, H.-P. Bischof. Many-to-Many Invocation: a new object oriented paradigm for ad hoc collaborative systems. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Pp. 72–73. ACM, New York, NY, USA, 2002. doi:http://doi.acm.org/10.1145/985072.985109

[KLS+08] N. Kefalakis, N. Leontiadis, J. Soldatos, K. Gama, 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*. Pp. 66–69. ACM, New York, NY, USA, 2008. doi:http://doi.acm.org/10.1145/1462735.1462751

[KR05] O. Kasten, K. Römer. Beyond event handlers: programming wireless sensors with attributed state machines. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*. P. 7. IEEE Press, Piscataway, NJ, USA, 2005.

[LC02] P. Levis, D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*. Oct. 2002. http://citeseer.ist.psu.edu/levis02mate.html

[MC02] R. Meier, V. Cahill. STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. Pp. 639–644. IEEE Computer Society, Washington, DC, USA, 2002.

[MPR01] A. Murphy, G. Picco, G.-C. Roman. Lime: A Middleware for Physical and Logical Mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*. Pp. 524–536. IEEE Computer Society, 2001. citeseer.ist.psu.edu/murphy01lime.html

[MQZ06] M. Mamei, R. Quaglieri, F. Zambonelli. Making tuple spaces physical with RFID tags. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*.

Pp. 434–439. ACM, New York, NY, USA, 2006.
doi:http://doi.acm.org/10.1145/1141277.1141378

[MTS05] M. Miller, E. D. Tribble, J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In Nicola and Sangiorgi (eds.), *Symposium on Trustworthy Global Computing*. LNCS 3705, pp. 195–229. Springer, April 2005.

[Van08] T. Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, May 2008.

[VMD08] T. Van Cutsem, S. Mostinckx, W. De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages Systems & Structures* 35(1), 2008.
doi:10.1016/j.cl.2008.06.005

[VMG+07] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter. AmbientTalk: object-oriented event-driven programming in Mobile Ad hoc Networks. In *Proceedings of the XXVI International Conference of the Chilean Computer Science Society (SCCC 2007)*. Pp. 3–12. IEEE Computer Society, 2007.
doi:http://doi.acm.org/10.1109/SCCC.2007.12