MORENA: A Middleware for Programming NFC-Enabled Android Applications as Distributed Object-Oriented Programs

Andoni Lombide Carreton, Kevin Pinte, Wolfgang Meuter

MORENA: A Middleware for Programming NFC-enabled Android Applications as Distributed Object-Oriented Programs

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. NFC is a wireless technology that allows software to interact with RFID tags and that is increasingly integrated into smartphones and other mobile devices. In this paper, we present MORENA: a middleware that treats NFC-enabled programs as distributed object-oriented programs in which RFID tags are represented as intermittently connected remote objects. We draw inspiration from the ambient-oriented programming paradigm to represent these objects as first-class remote references which only offer asynchronous communication with the tag to which they refer. This allows the programmer to implement mobile applications that read from or write to RFID tags without having to handle every single fault manually and without blocking the entire application during read or write operations. We built MORENA on top of the Android platform and evaluated our abstractions by implementing a representative application running on NFC-enabled Android phones using MORENA.

Keywords: RFID, mobile applications, Android, pervasive computing

1 Introduction

The Internet of Things [1][2] research vision can now be implemented using mainstream hardware. Smartphones and other mobile devices are increasingly equipped with NFC (Near Field Communication) chips that allow to read and write a wide range of RFID tags. The most prominent ones are high-end phones running Google's Android platform [3], such as Google's Nexus S. One of the reasons is that companies such as Google are interested in mobile payment applications, such as Google Wallet [4]. However, such applications are only a fraction of what is possibly with an NFC-enabled smartphone. Unfortunately, current APIs that allow the programmer to implement NFC-enabled applications are designed for very specific scenarios (such as mobile payment) and hence exhibit a number of drawbacks that make developing more complicated applications hard and error-prone.

1.1 Drawbacks of the Android NFC API

MORENA (MObile RFID-ENabled Android middleware) is designed around the Google Android NFC API, currently to our knowledge the most advanced NFC API for mobile devices that is available in the mainstream. This API is designed to cover the bare essentials to allow the programmer to implement NFC-enabled applications while not having to deal with every single hardware detail. Still, it suffers from a number of drawbacks, which we describe below.

Synchronous communication. Read and write operations on RFID tags are blocking operations in the Android NFC API. This means that a program performing such operations is suspended until these operations succeed or fail. Since these operations are slow in comparison with the rest of the program, the application becomes unresponsive when not carefully used. Therefore, the documentation of the API strongly recommends to run RFID operations in a separate thread. This burdens the programmer with manual concurrency management, which is hard and error-prone.

Coupling in time. Reading or writing RFID tags frequently fails because the tag in question is out of range. Especially with tiny NFC chips as the ones found in mobile devices, failure is the rule instead of the exception. Manually dealing with faults requires every single RFID operation to be protected with exception handling code, further complicating the application code. In many cases, operations will succeed shortly after their first failed attempt because of a small change in the physical environment, such as an RFID tag that is positioned differently with respect to the smartphone. This causes the programmer to write looping code merely for retrying failed operations. We say that communication is coupled in time.

Manual data conversion. The Android NFC API abstracts away the low-level memory layout of RFID tags. Still, the programmer must manually convert application-specific data that has been read from or that will be written to an RFID tag. This means that when RFID operations are separately developed from the application logic, the application programmer must have internal knowledge of these operations to understand how he or she should convert application-specific data to a suitable representation for storage on the RFID tags' memory. This is error-prone because the API does not enforce specific data conversions to be associated with specific applications and RFID tags.

Tight coupling with activity-based architecture. Android applications are always *activities*: special Java objects representing an Android GUI with a thread of execution. It is via these activities that the application is notified of I/O and user interface events (by means of *intents*) such as RFID events. Although this event-driven API makes it straightforward to override a number of

callbacks that capture these events and directly undertake the necessary actions in the activity code, it also introduces a tight coupling of the RFID operations with activities (i.e. the user interface). This makes it harder to perform RFID operations outside of the context of such an activity.

1.2 Ambient-oriented Programming

In this paper, we consider interaction with an RFID tag a distributed computing problem as opposed to traditional I/O. More specifically, we draw inspiration from the *ambient-oriented programming* paradigm [5], which is a programming paradigm targeting distributed systems consisting of mobile devices interconnected via unreliable, ad hoc wireless networks. Indeed, NFC can be regarded as an unreliable, ad hoc wireless communication technology while RFID tags can be considered as simple remote devices.

We have previously integrated RFID into ambient-oriented programming [6] by relying on non-mainstream RFID hardware and by building dedicated abstractions into an ambient-oriented research language called AmbientTalk [7]. In this work, we crystallize the ideas behind this research into an implementation on top of mainstream hardware (namely Android smartphones) and using mainstream programming technology (namely the Android platform in the Java language). The concepts from ambient-oriented programming discussed below are carried over as follows.

Tracking of connectivity. Ambient-oriented programs must keep track of which services become available and unavailable as devices roam. Similarly, an RFID-enabled application must be able to keep track of which RFID tags are currently in and out of range and be notified of changes in the connectivity with the tags it is interacting with.

Asynchronous communication. All distributed programming systems have primitives for sending and receiving data across the network. Ambient-oriented programming requires these primitives to be non-blocking: a process or thread of control should not be suspended if the operation cannot be completed immediately. This requirement is based on the fact that in an unreliable network, communicating parties can often be unavailable, and making a communication operation block until the communicating party is available may lead to unacceptable delays. Non-blocking communication is also known under the term *asynchronous communication*, the style of communication now also popular in rich web applications using AJAX that should not block the web interface. Similarly, for RFID-enabled applications, communication with an intermittently connected RIFD tag should not block the application when the tag is temporarily out of communication range.

Decoupling in time. Unreliable wireless connections require communication models that can abstract from the network connectivity between communicating

processes. It should be possible for two processes to express communication independently of their connectivity. This significantly reduces the case-analysis for the programmer, which can reason in terms of a fully connected network by default, and can deal with border cases in an orthogonal way. Similarly, exchanging data with an intermittently connected RFID tag is prone to many failures. In many cases, multiple attempts at reading from or writing to an RFID tags memory are needed before an operation succeeds. This should happen without immediately signaling an error for every single fault to the programmer. Instead, the implementation should retry these operations without blocking the application or signaling an error.

First-class references to remote objects. Decoupling in time is achieved by storing sent messages in an intermediary data-structure. This makes it possible for communicating parties to interact across unreliable connections, because the logical act of information sending is decoupled from the physical act of information transmission, allowing for the information to be saved and transmitted at a later point in time when the connection between both parties is restored.

In AmbientTalk, remote services and RFID tags are represented as remote objects which are always referred to by a remote reference called a *far reference*. These far references (first proposed in the E language [8]) are first class, encapsulate the identity of a remote object and store messages directed towards the remote objects that could not be sent due to physical phenomena. Additionally, far references encapsulate a thread of control that, in response to connectivity changes with the object which it refers to, attempts to forward its stored messages (in the correct order). Far referencers offer an asynchronous interface such that the programmer can register observers on it to be notified of connectivity changes and messages being successfully sent or timed out.

1.3 Approach

It was our goal to integrate the concepts described above into mainstream technology such as the Android platform. This is achieved by providing a middleware that readily integrates with the Android platform (version 4.0 and up). In short, the Android NFC API models RFID operations as **file I/O**, while MORENA treats RFID operations as **network communication**. Additionally, MORENA tackles the remaining drawbacks in the Android NFC API, namely manual data conversion and the tight coupling of the API with the activity-based architecture.

In the next section, we describe the abstractions offered by MORENA for interacting with RFID-tagged objects as if they where remotely connected software objects which are automatically converted to the correct data format for reading from and writing to RFID tags. Thereafter in Section 3, we descend one level deeper into the MORENA middleware which allows the programmer to deal with references to RFID tags directly and allows to encode custom encoding strategies for Java objects. Subsequently, in Section 4 we discuss the application of MORENA in a representative application. Section 5 discusses related work and finally, Section 6, details future work on MORENA and concludes this paper.

2 RFID-enabled Android Applications as Distributed Object-Oriented Programs

As mentioned in the introduction, the main idea is to no longer treat RFID communication as a form of I/O, but to come up with a suitable representation for RFID tags such that they can be treated as first-class remote objects. A second objective is to loosen the coupling with activity-based architecture of the Android API.

MORENA offers two layers of abstraction. On the highest level, the programmer uses special Java objects called *things* which are causally connected to a specific RFID tag and which can be automatically converted to the correct data format to be read from or written to RFID tags. The lower level requires the programmer to interact through a reference with the bare RFID tag, but allows to come up with custom data conversion strategies (a good example is storing specific fields of an object directly on the RFID tag while other fields are stored in some external database). This section is about the highest level where the programmer uses things.

2.1 Things

Consider an application where facilities offer guests access via their smartphones or tablets to their WiFi access points by swiping over an RFID tag that contains the credentials for connecting to the WiFi network. Using the things abstraction, an object that is read from and written to RFID tags must be a thing. Consider the `WifiConfig` class that allows us to create such things defined below.

```
public class WifiConfig extends Thing {
  public String ssid_;
  public String key_;

  public WifiConfig(
      ThingActivity<WifiConfig> activity,
      String ssid,
      String key) {
    super(activity);
    ssid_ = ssid;
    key_ = key;
  }

  public boolean connect(WifiManager wm) {
    // Connect to ssid_ with password key_
  };
}
```

`WifiConfig` things are simple objects containing two fields, the SSID and password of a WiFi network. All fields that are not declared **`transient`** are serialized when the thing is stored on an RFID tag. In this case, both fields are stored. Serialization of things happens by converting Java objects to the JSON format

using Google's own serialization library (GSON) built into the Android platform. GSON performs deep serialization of all JSON-serializable fields, but does not support cycles in the object graph to serialize.

Creating a thing requires passing the Android activity in its constructor, as shown in the example. MORENA offers a dedicated activity called `ThingActivity` which is parametrized with the type of things the activity is interacting with. In this case, this is the `WifiConfig` thing type. Internally, such a `ThingActivity` captures all low level Android events (such as the ones typically signaled by means of intents) and triggers the correct actions on the associated thing objects. This frees the programmer from dealing with Android activities directly for every single operation or event.

2.2 Initializing Things

In this section we discuss the initialization of empty RFID tags using things. On the level of abstraction discussed in this section, the programmer can make use of several callbacks that can be overridden on the `ThingActivity`. The one to use for initializing things is the one overridden below.

```
@Override
public void whenDiscovered(EmptyRecord empty) {
  empty.initialize(
    myWifiThing,
    new ThingSavedListener<WifiConfig>() {
      @Override
      public void signal(WifiConfig thing) {
        toast("WiFi joiner created!");
      }
    },
    new ThingSaveFailedListener() {
      @Override
      public void signal() {
        toast("Creating WiFi joiner failed, try again.");
      }
    });
}
```

It is triggered each time an empty RFID tag is detected. It is triggered with an `EmptyRecord`, which is a special thing object denoting an empty tag. Its `initialize` method is used to initialize it with a thing object that at that moment in time is not bound yet to a particular RFID tag. Note that initializing a thing involves writing data to the RFID tag to store the serialized thing in its memory. Since this is an operation that may be long lasting (compared to other computations) and since it may frequently fail, MORENA enforces that it happens asynchronously. For this, `initialize` takes in this case three arguments: the thing to store on the empty tag, a listener object that will be invoked when the thing is successfully initialized, and an object listener that will be invoked when the operation fails given a default timeout. Various overloaded versions of

`initialize` exist, such that for example the failure listener can be omitted or the timeout value can be manually specified. We chose to expect two different listener objects as opposed to a single listener object implementing two different callbacks: one for success and one for failure. The reason is flexibility: these tiny listener objects are usually created by directly implementing an interface, while at the same time, in many cases different success listeners are needed while only a single or handful failure listeners are required (or the other way around). Separating them into separate first-class objects introduces more syntax, but prevents code duplication in such situations.

2.3 Discovering and Reading Things

Just like the `whenDiscovered` callback for detecting empty RFID tags shown above, there is an overloaded variant that can be used to detect things that are already initialized. In our example application, it is overridden as follows:

```
@Override
public void whenDiscovered(WifiConfig wc) {
  toast("Joining Wifi network " + wc.ssid_);
  wc.connect();
}
```

This callback will be triggered every time an RFID is scanned which contains a thing of type `WifiConfig`. Upon scanning, the data is deserialized and passed as a `WifiConfig` argument to this callback.

For ease of programming, such a thing object like `wc` encapsulates a cached version of this deserialized object which allows synchronous access to its fields and methods. This is used in the example above to call the `connect` method which connects the Android device to the WiFi network specified in the `wc`.

However, synchronous access is not without danger since other devices might have concurrently updated the thing stored in the RFID tags memory. In this case, no problem can occur because immediately after detecting the thing, the `connect` method is called. For critical cases, the programmer must rely on the asynchronous operations discussed in Section 3.

2.4 Saving Modified Things

The programmer is free to modify thing objects. However, this will render them inconsistent with their serialized counterpart stored on the corresponding RFID tag. To write through any changes performed on a thing to the tag memory, the programmer must explicitly *save* the object. Since such a save operation involves writing the serialized thing onto the tag, which is a long-lasting operation that may frequently fail, MORENA enforces save operations to happen asynchronously. The code snippet below shows how saving a modified thing happens.

```
myWifiConfig.ssid_ = "MyNewWifiName";
myWifiConfig.key_  = "MyNewWifiPassword";

myWifiConfig.saveAsync(
  new ThingSavedListener<WifiConfig>() {
    @Override
    public void signal(WifiConfig wc) {
      toast("WiFi joiner saved!");
    }},
  new ThingSaveFailedListener() {
    @Override
    public void signal() {
      toast("Saving WiFi joiner failed, try again.");
    }});
```

Analogous to thing initialization discussed earlier in Section 2.2, a success listener and a failure listener can be supplied to be notified of a successful or failed save. Again, various overloaded versions of the `saveAsync` method exist, depending on which callbacks must be specified and whether the timeout value should be different from the default one. Since we are dealing with NFC technology, which only has a range of a few centimeters, we assume that race conditions are nigh impossible if no exuberantly large timeout values are chosen by the programmer. One of the future features of MORENA that we are investigating is providing alternative protection mechanisms against such race conditions.

2.5 Broadcasting Things

Other than using a phone's built-in NFC chip for reading and writing RFID tags, the Android NFC API allows to use this same wireless communication technology to exchange data in an ad hoc fashion between two phones in NFC communication range. This technology is called *Beam*. The Beam API is largely similar to the API for communication with RFID tags, which it means it suffers from the same drawbacks, namely synchronous communication, coupling in time, manual data conversion and a strong coupling with the activity-based architecture. MORENA allows to easily exchange thing objects between phones over an NFC connection using beam. In our example application, users can connect other users to the WiFi network by bringing their phones close together and broadcasting a `WifiConfig` thing. This happens as follows.

```
myWifiConfig.broadcast(
  new ThingBroadcastSuccessListener<WifiConfig>() {
    @Override
    public void signal(WifiConfig wc) {
      toast("WiFi joiner shared!");
    }},
  new ThingBroadcastFailedListener<WifiConfig>() {
    @Override
```

```
    public void signal(WifiConfig wc) {
      toast("Failed to share WiFi joiner, try again.");
    }});
```

As one can see from the `broadcast` method used above, this is again an asynchronous operation (as it may frequently fail), adhering to the interface used before in this paper.

The reception of such a thing object using `broadcast`, causes the standard `whenDiscovered` callback of the receiving `ThingActivity` to be invoked. Remember from our example application that upon reception it will connect the Android device to the WiFi network stored on the tag. Things received via broadcast will not be bound to a particular RFID tag (although they can later be by initializing empty tags with them).

3 RFID-Tagged Objects by Reference

In this section, we descend a level of abstraction lower in the MORENA middleware. It offers a reference abstraction to RFID-tagged objects instead of thing objects to the programmer, which allows for asynchronous read and write operations with custom data conversion strategies. The thing abstractions are built directly on top of this layer of abstraction. For the sake of brevity, we use a simple example application that allows to read and write strings onto RFID tags supplied by the user.

3.1 Detecting RFID Tags

Detecting RFID tags already happens in an event-driven manner by activities in the Android API. MORENA offers a `TagDiscoverer` class that captures these events generated by a specific activity and uses them to generate *tag reference* objects: the objects that represent remote references to RFID tags in MORENA.

Consider a simple Android application that simply shows plain text stored on the last scanned RFID tag and allows the user to overwrite it with new content. One could create a `TagDiscoverer` subclass as shown below.

```
private class MyTagDiscoverer extends TagDiscoverer {
  @Override
  public void onTagDetected(TagReference ref) {
    readTagAndUpdateUI(reference);
  }
  @Override
  public void onTagRedetected(TagReference ref) {
    readTagAndUpdateUI(reference);
  }
}
```

This subclass overrides two methods that can be used to track the connectivity of an RFID tag: `onTagDetected` for a tag that has never been detected before, and

`onTagRedetected` for a tag that has already previously been detected. These methods are called with a tag reference as sole argument, which can subsequently be used to interact with the RFID tag (as explained below in Section 3.2). In this simple application, the user interface showing the contents of the last scanned tag is updated with the contents of the tag (the implementation of `readTagAndUpdateUI` is shown in Section 3.2).

`TagDiscoverers` are instantiated by passing them the activity (of type `NFCActivity`) that generates the RFID events and a MIME type that identifies the type of data that the tag contains such that the correct intent is triggered by the activity. Tags containing other types of data are disregarded. Typically, this data type is defined per application, as shown below for our example application.

```
new MyTagDiscoverer(
  this,
  TEXT_TYPE,
  new NdefMessageToStringConverter(),
  new StringToNdefMessageConverter());
```

Additionally, `TagDiscoverers` are associated with two converter objects that are responsible for converting objects for storage on RFID tags and data read from an RFID tag back into the correct object. These converter objects are explained later in Section 3.2. The idea is to encapsulate data conversion within `TagDiscoverer`s and the `TagReference`s they generate. This way, an `NFCActivity` can easily use multiple tag references without worrying about data conversion. Once a `TagDiscoverer` is instantiated, the programmer must no longer worry about activities.

3.2 The Tag Reference Abstraction

Once a tag reference is obtained (either through a `TagDiscoverer` or by parameter passing), it offers a non-blocking event-driven API in its own right for asynchronously reading from and writing data from the tag. Additionally, it keeps a queue of buffered read and write operations that are still waiting to be processed (for example because the RFID tag to which it points is temporarily unavailable). Tag references encapsulate a private event loop that uses its own thread of control to sequentially check if the first message in the queue can be processed. If it fails, it just remains in the queue. If it succeeds, the registered event listener on this asynchronous operation is triggered and the operation is removed from the queue, after which the tag reference attempts to execute the next scheduled operation. It is guaranteed that a message is never processed before previously scheduled messages are processed first. If an operation times out, it is removed from the queue as well and the next operation is attempted, but this time the failure listener associated with the operation is triggered (if there is one).

Listeners associated with these non-blocking tag reference operations are always asynchronously scheduled for execution in the activity's main thread, which frees the programmer of manual concurrency management. It also means that

usually all statements after a tag reference operation in the code are executed first before the listeners are executed. Synchronization of operations must happen by nesting these listeners.

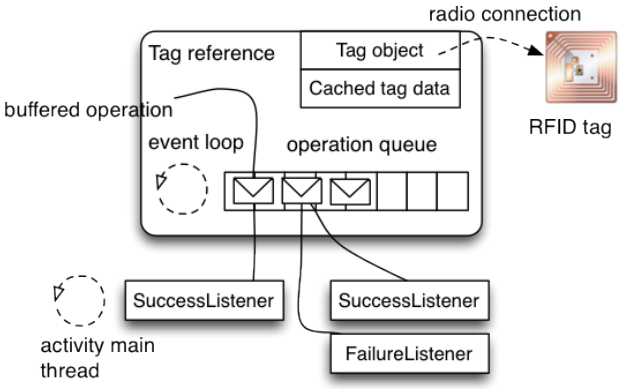

Fig. 1. The tag reference abstraction.

The tag reference abstraction is depicted schematically in figure 1. In addition to the bare Android tag object, it also encapsulates a cached version of the contents of the RFID tag, which is updated after each read and write operation. Although it provides synchronous access to these cached data, the programmer must be aware that if a tag is not seen for some time, its contents might have changed and an asynchronous read is a better option.

Within one Android activity, only a single unique tag reference can exist to the same RFID tag. Behind the scenes, `TagDiscoverer` instances use a private `TagReferenceFactory` that generates tag references for tags that are detected for the very first time, and subsequently reuses these references when tags are redetected and a reference to them is requested. It is however the programmer's responsibility to garbage collect unused tag references, as this is application specific and usually driven by external events (as opposed to internal references). For future versions of MORENA, we are investigating leasing strategies [9] that allow the application to obtain a lease on an RFID tag for a limited amount of time, after which it expires and the reference to the tag can be safely garbage collected.

In the two subsequent sections, we describe the asynchronous interface offered by the tag reference abstraction.

Reading RFID Tags Below is the implementation of the private method that is called by the `TagDiscoverer` class of our simple example application.

```
private void readTagAndUpdateUI(TagReference ref) {
  tagReference_ = ref;
```

```
  ref.read(
    new TagReadListener() {
      @Override
      public void signal(TagReference ref) {
        handleTagRead(ref);
      }},
    new TagReadFailedListener() {
      @Override
      public void signal(TagReference ref) {
        handleTagReadFailed();
      }});
}
```

As we showed in Section 3.1, when a new tag is detected or a previously detected tag is redetected, this method is called with the obtained tag reference. The tag reference is used for asynchronously reading the tag. If this does not succeed within a predefined timeout, an error is shown to the user. If it succeeds within the predefined timeout, the user interface is updated with the cached data of the tag reference.

Writing RFID Tags Writing tags using a tag reference happens in a similar fashion. The listener shown below is triggered by our simple example application when the user clicks the button that causes new text being inputted by the user to be written to the last seen RFID tag.

```
private OnClickListener saveButtonListener =
  new OnClickListener() {
    public void onClick(View button) {
      String toWrite = // Get text from EditText field
      tagReference_.write(
        toWrite,
        new TagWrittenListener() {
          @Override
          public void signal(TagReference ref) {
            handleTagRead(ref);
          }},
        new TagWriteFailedListener() {
          @Override
          public void signal(TagReference ref) {
            handleTagWriteFailed();
          }});
    }
  };
```

It just gets the data from a text field, which is afterwards automatically converted to the appropriate format by the tag reference. This way, data conversion is defined per tag reference and given such a tag reference, the programmer must no longer worry about it.

Just like for reading tags, we allow to register separate listener objects for successful writes and failed writes. In the success listener, the user interface is updated with the new cached data of the tag (which is the data that has been physically written on it, otherwise this listener would not have been triggered). In the failure listener, an error message is shown to the user.

Converting Objects for Storage on RFID Tags Converting objects for storage on RFID tags and converting data read from RFID tags back to objects happens on a per-tag reference and per-`TagDiscoverer` basis. This decouples detection of tags and data conversion from the `NFCActivity`. Implementing these converters requires some knowledge about the Android NFC API, namely its implementation of the NDEF[1] (NFC Data Exchange Format) standard [10].

The class shown below implements a converter for converting data read from an RFID tag into a string for our example application.

```
private class NdefMessageToStringConverter
  implements NdefMessageToObjectConverter {
    @Override
    public Object convert(NdefMessage ndefMessage) {
      return new String(
        (ndefMessage.getRecords()[0]).getPayload());
    }
  };
```

In our simple example application, tags contain just a single record containing a string.

The class shown below implements the corresponding converter for converting a string back to the NDEF format for storage on an RFID tag's memory.

```
private class StringToNdefMessageConverter
  implements ObjectToNdefMessageConverter {
    @Override
    public NdefMessage convert(Object o) {
      String toConvert;
      if (o == null) { toConvert = ""; }
      else { toConvert = (String)o; }
      NdefRecord r = new NdefRecord(
        NdefRecord.TNF_MIME_MEDIA,
        TEXT_TYPE,
        new byte[0], // No id.
        toConvert.getBytes(Charset.forName("UTF-8")));
      return new NdefMessage(new NdefRecord[]{ r });
    }
  };
```

[1] NDEF messages are in essence lists of byte arrays (NDEF records) in which the data must be stored.

The details are not of great importance to this paper. It simply creates a byte representation of the string in the correct charset and stores it in a single `NdefMessage` object contained into a new `NdefRecord`. This record specifies the type of tags on which `TagDiscoverer`s filter.

3.3 Interaction with Other Phones Using Beam

Similar to interaction with RFID tags, we built an asynchronous, event-driven API for exchanging beamed messages. Being notified of an asynchronously received beam message happens by registering a `BeamReceivedListener`, such as shown below.

```
new MyBeamListener(
  this,
  TEXT_TYPE,
  new NdefMessageToStringConverter());
```

Just like a `TagDiscoverer`, its constructor takes an `NFCActivity` as first argument, the tag MIME type and a read converter. This allows that the `BeamReceivedListener` autonomously converts received NDEF messages to objects without the programmer needing to worry about the activity which signals the low-level events.

Below is the implementation of the subclass instantiated above.

```
private class MyBeamListener extends BeamReceivedListener {
    @Override
    public void onBeamReceived(Object o) {
      // Set text of EditText field.
    }
}
```

The programmer must override the `onBeamReceived` callback to react on a received beam message. The data transported in the beam message is automatically converted into an object using the read converter of the `BeamReceivedListener`.

In contrast to the interaction with RFID tags, beaming does not happen by means of a reference abstraction. The reason is that beaming is an undirected operation that broadcasts a message to any device willing to accept the beamed data. Instead, beaming messages to other phones happens using `Beamer` objects that again encapsulate data conversion to decouple this from the activity. The instantiation of the `Beamer` object used by our example application is shown below. The first argument is the `NFCActivity`.

```
private Beamer beamer_ = new Beamer(
  this,
  new StringToNdefMessageConverter());
```

Just like for RFID operations, beaming messages must happen asynchronously, using the `beam` method that is used below in the listener that is triggered when the user clicks the beam button.

```
private OnClickListener beamButtonListener =
  new OnClickListener() {
    public void onClick(View button) {
      String toBeam = // Get text from EditText field
      // Beaming is undirected.
      beamer_.beam(
        toBeam,
        new BeamSuccessListener() {
          @Override
          public void signal() {
            handleBeamSucceeded();
          }},
        new BeamFailedListener() {
          @Override
          public void signal() {
            handleBeamFailed();
          }});
    }
  };
```

When this button is clicked, the data to be beamed is retrieved from a text field in the user interface and passed to the asynchronous beam operation. To detect a successful beam operation, it takes a listener as second argument. To detect if the beamed message times out, as a third argument it takes another listener. These listeners are optional and are the only way to be notified of the state of the asynchronous operation. It exhibits the same behavior as performing an asynchronous write operation on an RFID tag.

Of particular importance is the fact that data conversions are now encapsulated in `TagDataConverter` objects, which are associated with `TagReference`, `TagDiscoverer`, `Beamer` and `BeamReceivedListener` objects. This means that a single activity can use multiple `TagDiscoverer`s generating different `TagReference`s and different `Beamer`s and `BeamReceivedListener`s all with their separate data conversion strategies that are automatically applied when exchanging data with RFID tags or using Beam.

3.4 Filtering Events

As discussed earlier in this section, the only way to distinguish between interesting scanned tags or interesting received beam messages and non-interesting ones, is to filter on the tag type (as is done by the `TagDiscoverer`s and `BeamReceivedListener`s). Since this is a rather coarse-grained way of filtering, the programmer finds himself implementing filtering behavior manually and scattered over the application code. This is why `TagDiscoverer`s and `BeamReceivedListener`s offer an additional method that can be optionally overridden by the programmer.

For `TagDiscoverer`s, this `checkCondition` method is a predicate that will be applied on the tag reference generated or retrieved by the `TagDiscoverer`, as shown below.

```
private class MyTagDiscoverer extends TagDiscoverer {
  // ... Same as before ...
  @Override
  public boolean checkCondition(TagReference ref) {
    // ... condition ...
  }
}
```

A typical pattern is that the cached data of the tag reference is used to filter on.

For `BeamReceivedListener`s, a similar predicate can be applied on the object received in the beam message, as shown below.

```
private class MyBeamListener extends BeamReceivedListener {
  // ... Same as before ...
  @Override
  public boolean checkCondition(Object o) {
    // ... condition ...
  }
}
```

Only when these predicates are satisfied, the listeners are triggered.

4 Evaluation

In this section, we compare two versions of the WiFi sharing application used as an example throughout this paper. The first version is based on the standard NFC API of the Android platform. The second is almost exactly the same application[2], but built on top of the MORENA middleware[3]. The focus of our work is reducing the effort that is needed to develop an RFID-enabled Android application. As a metric we chose to count the lines of code needed for implementing particular RFID subproblems in the application. These subproblems are **(1) event handling** (e.g. to be notified of detected tags), **(2) data conversion**, **(3) failure handling**, **(4) read/write functionality**, and finally **(5) concurrency management** (to prevent blocking the application on tag I/O).

Figure 2 shows two graphs comparing both implementations. The graph on the left-hand side shows a comparison of the number of lines of code dedicated to each subproblem. The total number of RFID-related lines of code for the handcrafted implementation is 197 and for the implementation based on MORENA 36 (a reduction by a factor 5).

The right-hand side shows the percentages that the RFID subproblems constitute to the total count. We observe that MORENA shifts the focus to event

[2] We will discuss the differences at the end of this section.

[3] The source of these experiments and the MORENA middleware can be downloaded at: `http://soft.vub.ac.be/amop/research/rfid/morena/files`

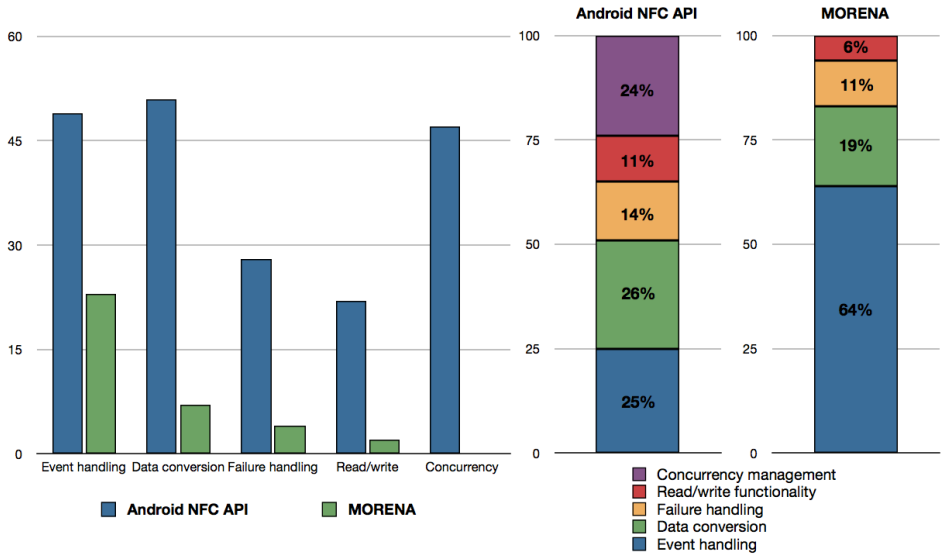

Fig. 2. Comparison of handcrafted RFID code and MORENA code.

handling and frees the programmer of any concurrency management. This is to be expected because MORENA's asynchronous communication abstractions take care of concurrency automatically at the expense of more event-driven code.

This leads us to a final note on this comparison. MORENA not only simplifies dealing with RFID technology, it also holds another bonus over the handcrafted implementation. Thanks to its asynchronous communication abstractions, operations that fail due to tag disconnections are automatically retried, which is not incorporated in the handcrafted version, in which the user must manually reattempt the operation. Furthermore, in the MORENA version, multiple write operations can be batched until a tag comes in range, while in the handcrafted solution the user can only attempt to write as soon as a tag is in range. Implementing the same behavior in the handcrafted version will further complicate the implementation. In short, MORENA not only significantly reduces the complexity of implementing RFID-enabled Android applications, but in comparison to naively using the Android NFC API offers a better user experience as well.

5 Related Work

Typical application domains for RFID technology are asset management, product tracking and supply chain management. In these domains RFID technology is usually deployed using traditional RFID middleware, such as Aspire RFID [11] and Oracle's Java System RFID Software [12]. 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, although some of them allow to store information directly on the tags, such as for example WinRFID.

WinRFID [13] 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 is also specified in XML and is 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.

Fosstrak [14] (formerly named Accada) is an open source RFID middleware platform that is based on the Electronic Product Code standards [15]. Fosstrak offers a virtual tag memory service (VTMS) that similarly to our approach facilitates writing application-specific data to RFID tags asynchronously. However, Fosstrak only supports writing key-value pairs.

In contrast to MORENA, the systems discussed above do not target mobile applications running on for example smartphones. Still, in the literature one can find interesting mobile applications making use of RFID, such as home care [16] or the tracking of personal belongings [17]. This reinforces our idea that there is a need for better programming abstractions in this domain. Conversely, MORENA does not target industrial applications which have to deal with a massive amount of RFID tags, and thus require greater scalability. We are currently investigating how to carry over some of MORENA's concepts into such a middleware.

An alternative distributed computing paradigm to the ambient-oriented programming, on which MORENA is based, is distributed tuple spaces. In [18], RFID tags are used to store application-specific data and form a distributed tuple space that is dynamically constructed by all tuples stored on the tags that are in reading range. Mobile applications interact by means of traditional tuple space operations. However, there is no way to control on which specific tag tuples will be stored.

6 Conclusion and Future Work

In this paper, we have presented MORENA, a middleware that aims at raising the level of abstraction on which developers can build RFID-enabled Android applications. We have evaluated the abstractions offered by MORENA by implementing a mobile RFID-enabled application using the bare essentials provided by the Android platform and comparing the implementation to an implementation based on MORENA. We observe that using MORENA significantly eases the development of mobile RFID-enabled Android applications.

The main feature that remains to be added in a future version of MORENA is a leasing mechanism which has two goals. The first goal is to protect cached thing objects from data races when other RFID-enabled devices are able to write new data on their corresponding RFID tags. The second goal is to allow cached objects to be garbage collected automatically. The mechanism that we envision is to write a locking timestamp and a device ID on the RFID tag's memory by the device willing to interact with the tag. Only if this succeeds, the device is granted exclusive access. The timestamp dictates for how long the device has exclusive access to the memory of the tag. Beyond this timestamp, the lease expires and the device looses its exclusive access, unlocking the tag for interaction with other devices. The assumption made here is that the clock drift among Android devices is small enough to exclude practically all race conditions.

To summarize, the abstractions offered by the MORENA middleware make developing mobile RFID-enabled Android applications easier as follows:

Automatic conversion of thing objects. MORENA's thing objects can be used as regular Java objects, but can in addition be seamlessly read from or written to RFID tags.

Tracking of connectivity. MORENA offers an event-driven interface such that an application can be notified if a *particular* RFID tag is in or out of communication range.

First-class references to RFID tags. In MORENA, RFID tags are uniquely linked to tag references or thing objects.

Asynchronous communication and decoupling in time. MORENA offers asynchronous and fault-tolerant operations for reading or writing the RFID tags' memories, reducing the case analysis for the programmer and freeing the programmer from manual concurrency management to keep the application responsive.

Looser coupling from the activity-based architecture. MORENA encapsulates the low-level NFC API which is tightly coupled to Android activities into thing objects, tag references or other higher level abstractions such that applications become less coupled to the user interface.

Support for Beam. Just like reading and writing things to and from RFID tags, using the same abstractions, things can be broadcasted to other phones using NFC.

References

1. L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.
2. G. Kortuem, F. Kawsar, V. Sundramoorthy, and D. Fitton, "Smart objects as building blocks for the internet of things," *IEEE Internet Computing*, vol. 14, pp. 44–51, 2010.
3. S. Komatineni, D. MacLean, S. Y. Hashimi, S. Komatineni, D. MacLean, and S. Y. Hashimi, "Introducing the android computing platform," in *Pro Android 3*, pp. 1–20, Apress, 2011.

4. R. Handa, K. Maheshwari, and M. Saraf, *Google Wallet - A Glimpse Into the Future of Mobile Payments*. GRIN Verlag GmbH, 2011.
5. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-oriented Programming in Ambienttalk," in *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)* (D. Thomas, ed.), vol. 4067 of *Lecture Notes in Computer Science*, pp. 230–254, Springer, 2006.
6. A. Lombide Carreton, K. Pinte, and W. De Meuter, "Software abstractions for mobile rfid-enabled applications," *Software: Practice and Experience*, 2011.
7. T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter, "Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks," in *XXVI International Conference of the Chilean Computer Science Society*, pp. 3–12, IEEE Computer Society, 2007.
8. 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.
9. C. Gray and 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*, (New York, NY, USA), pp. 202–210, ACM Press, 1989.
10. G. Madlmayr, J. Ecker, J. Langer, and J. Scharinger, "Near field communication: State of standardization," in *Proceedings of the International Conference on the Internet of Things 2008* (F. Michahelles, ed.), vol. 1 of *1*, p. 6, ETH Zürich, ETH Zürich, 03 2008.
11. N. Kefalakis, N. Leontiadis, J. Soldatos, K. Gama, and D. Donsez, "Supply chain management and NFC picking demonstrations using the AspireRfid middleware platform," in *ACM/IFIP/USENIX Middleware '08*, (New York, NY, USA), pp. 66–69, ACM, 2008.
12. Oracle (Sun Developer Network), "Developing auto-id solutions using sun java system rfid software."
13. B. S. Prabhu, X. Su, H. Ramamurthy, C.-C. Chu, and R. Gadh, "Winrfid – a middleware for the enablement of radio frequency identification (rfid) based applications," white paper, UCLA – Wireless Internet for the Mobile Internet Consortium, January 2008.
14. 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.
15. EPCGlobal Standards Overview, "http://www.epcglobalinc.org/standards," September 2010.
16. J. Sidén, V. Skerved, J. Gao, S. Forsström, H.-E. Nilsson, T. Kanter, and M. Gulliksson, "Home care with nfc sensors and a smart phone," in *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, ISABEL '11, (New York, NY, USA), pp. 150:1–150:5, ACM, 2011.
17. M. K. Watfa, M. Kaur, and R. F. Daruwala, "Ipurse: An innovative rfid application," in *Advances in Education and Management* (M. Zhou, ed.), vol. 211 of *Communications in Computer and Information Science*, pp. 531–538, Springer Berlin Heidelberg, 2011.
18. 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.