Object-oriented Coordination in Mobile Ad hoc Networks

Tom Van Cutsem*, Jessie Dedecker, and Wolfgang De Meuter

tvcutsem|jededeck|wdmeuter@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Brussels – Belgium

Abstract. We introduce an object-oriented referencing abstraction to express coordination between objects hosted on mobile devices interconnected by a wireless ad hoc network. On the one hand, we notice that the most popular communication paradigms for mobile ad hoc networks, such as publish/subscribe and tuple space architectures, promote loose coupling of collaborating participants. On the other hand, the paradigm in which many applications are developed is object-oriented, and traditional object referencing abstractions typically lack the beneficial loose coupling properties of aforementioned paradigms. This paper proposes to close the paradigmatic gap between an object-oriented language and its distributed communication infrastructure by introducing *ambient references*: loosely-coupled remote object references designed for mobile ad hoc networks.

1 Introduction

The flourishing of research fields such as pervasive and ubiquitous computing [1] has lead to a tremendous increase in research on *mobile ad hoc networks* – networks composed of portable, mobile devices interconnected by wireless communication media. Such networks are often regarded as the ideal hardware infrastructure to support pervasive and ubiquitous computing scenarios [2]. The network's wireless capabilities, combined with the mobility of the devices, results in applications where software entities spontaneously detect one another, engage in various collaborations, and may disappear as swiftly as they have appeared. Example applications range from modest, already commonplace applications like collaborative text-editors, to more futuristic scenarios such as warehouses equipped with digital infrastructure allowing customers to interact with products, their shopping carts, etc.

This paper focuses on distributed programming language support for mobile networks. In distributed programming, communication paradigms based on loose coupling between the participants have been especially promoted in the context of mobile ad hoc networks [3–6]. Interestingly, none of these approaches is object-oriented in nature, while most mainstream programming languages in which applications are developed are. One of the reasons for this paradigm mismatch is that remote object references

* Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

have not been as successful in achieving loose coupling between collaborating parties as other paradigms, such as publish/subscribe [7] and tuple space [8] architectures.

The contribution of this paper is the proposition of a loosely coupled object-oriented coordination abstraction for mobile networks, named an *ambient reference*. This abstraction eliminates the paradigm mismatch between object-oriented applications and loosely coupled distributed coordination infrastructure, because it allows object-oriented programs to interact without leaving the paradigm, while keeping the benefits of loose coupling promoted by established collaboration paradigms. Ambient references have been implemented in a distributed object-oriented language called AmbientTalk, which we will briefly describe as well.

2 Motivation

Based on the fundamental characteristics of mobile hardware, we discern a number of phenomena that set mobile networks apart from their traditional, fixed counterparts. We show how these phenomena have motivated the choice of loosely coupled interaction paradigms for use in mobile networks. Next, we will highlight why traditional object-oriented distributed computing does not promote loose coupling and hence requires the use of other communication paradigms, leading to a paradigm mismatch.

2.1 Characteristics of Mobile Networks

There are two discriminating properties of mobile networks, which clearly set them apart from traditional, fixed computer networks: applications are deployed on *mobile* devices which are connected by *wireless* communication links with a limited communication range. Such networks exhibit two phenomena which are rare in their fixed counterparts, and which will be shown to be the main instigators for loosely-coupled interaction:

Volatile Connections. Mobile devices equipped with wireless media possess only a limited communication range, such that two communicating devices may move out of earshot unannounced. The resulting disconnections are not always permanent: the two devices may meet again, requiring their connection to be re-established. Quite often, such *transient* disconnections should not affect an application, allowing both parties to continue their collaboration where they left off. Although dealing with disconnection is not a new ingredient of distributed systems, these more frequent transient disconnections do expose applications to a much higher rate of partial failure than that which most distributed languages or middleware have been designed for.

Zero Infrastructure. In a mobile network, devices that offer services spontaneously join with and disjoin from the network. As a result, in contrast to stationary networks where applications usually know where to find collaborating services via URLs or similar designators, applications in mobile networks have to find their required services dynamically in the environment. Services have to be discovered on proximate devices, possibly without the help of shared infrastructure (e.g. a wireless base station), requiring a *peer-to-peer* network topology.

Any application designed for mobile ad hoc networks will have to deal with the above phenomena. Moreover, these phenomena are not easily hidden within a standard programming language or middleware because their effects pervade the entire application. In the following section, we show how dedicated communication paradigms can drastically ease the burden of dealing with these phenomena.

2.2 Loosely-coupled Collaboration

In this section, we describe requirements for communication paradigms that, when adhered to, significantly reduce the impact of the above phenomena on software. The first three requirements pertain to *decoupling* the communicating parties along three dimensions as explained in detail in [7]. For each requirement, we state why it is critical in the context of mobile ad hoc networks.

Requirement 1 (Decoupling in Time) *The communicating parties do not need to be online at the same time.*

Requirement 1 states that a sender may send a message to a recipient that is offline, and a recipient may receive and process a message from a sender that is offline. This makes it possible for communicating parties to interact across volatile connections. Decoupling in time is directly inspired by the need to deal with the intermittent disconnections inherent to mobile ad hoc networks.

Requirement 2 (Decoupling in Space) *The communicating parties do not need to know each other beforehand.*

Requirement 2 states that communicating parties do not necessarily need to know one another's exact address or location. It implies that communicating parties can rely on some mechanism other than precise addresses or URLs to get to know one another. Decoupling in space is an important property in mobile ad hoc networks because they have a minimum of shared infrastructure, making reliance on servers to mediate collaborations impractical.

Requirement 3 (Synchronization Decoupling) *The control flow of communicating parties is not blocked upon sending or receiving.*

Requirement 3 states that a sending party can employ a form of *asynchronous* message passing, such that the act of message *sending* becomes decoupled from the act of message *transmission*. Likewise, allowing recipient parties to process messages asynchronously decouples the act of message *reception* from the act of message *processing*. Message transmission and reception require a connection between sender and receiver, but message sending and processing can be decoupled, allowing communicating parties to abstract over the fact whether the other party is online or not. This requirement is again directly derived from the volatile connections phenomenon in mobile networks. It allows parties to perform useful work while being disconnected.

Requirement 4 (Connection Awareness) *Communicating parties must be able to keep an up-to-date view of which participants are (dis)connected.*

At first glance, requirement 4 seems to somewhat contradict the above three requirements, because it seems to state that a process is no longer able to abstract over the state of the connection with communicating parties. However, this is not necessarily the case if the aspect of communication can be separated from the aspect of failure handling by means of orthogonal mechanisms. Being aware of the state of the connection of a participant is important because due to the limited infrastructure in mobile ad hoc networks, delivery guarantees for exchanged messages are often very weak. Hence, communicating parties must sometimes take explicit action when a participant disconnects.

2.3 A Paradigm Mismatch

In object-oriented distributed computing, objects distributed across several machines may refer to and communicate with one another by means of *remote object references*. A remote object reference represents a communication channel to a particular remote object. In its most simple form, distributed message passing is a straightforward adaptation of local message passing, known as remote method invocation (RMI). Using RMI, distributed request/response interaction is very easily expressed. Unfortunately, RMI does not decouple objects in time, space or synchronization [7]. However, asynchronous adaptations of RMI (e.g. Rover's Queued RPC [9]) have achieved decoupling in time and synchronisation.

Other communication paradigms have been more successful at achieving loose coupling between participants. For example, in *publish/subscribe* communication publishers asynchronously publish *events* on channels which leads to the asynchronous notification of registered subscribers [7]. Quite often, an *event service* acts as a middle man between publishers and subscribers, allowing them to be decoupled in space. Publishers may publish events even if no subscribers are registered on a channel and vice versa, making them decoupled in time. Tuple spaces, discussed in more detail in section 6.2, achieve a similar decoupling between participants.

In practice, object-oriented programs that require loosely coupled distributed communication abandon the remote reference and message passing abstractions in favour of paradigms such as publish/subscribe and tuple spaces. This requires object-oriented code to be adapted to the communication paradigm. For example, rather than sending messages to remote objects, publishers publish *event objects* on an event channel [10] or processes insert *tuple objects* into a shared space. Method invocation is replaced by subscribing event handlers on channels or by querying a tuple space using a template object, as in JavaSpaces [11].

These adaptations achieve better decoupling of objects, but at the price of giving up on the advantage of remote references to easily express request/response interactions. For example, messages sent via a remote reference have an explicit receiver, so multiple messages sent via the same reference are processed by the same receiver. Without additional programming, this property no longer holds when broadcasting events or publishing tuples. Also, messages invoke methods which have a return value. In contrast, matching an event or tuple that represents a request with its corresponding response event or tuple must be done explicitly in the code.

The contribution of this paper lies in an integration of the above requirements in an object-oriented language, such that distributed communication can still be expressed in

terms of objects sending messages to one another. Before introducing ambient references, we first introduce the object-oriented programming language in which they have been developed.

3 The AmbientTalk Language

Ambient references have been implemented as part of the AmbientTalk programming language. AmbientTalk is an object-oriented distributed programming language specifically designed for distributed programming in mobile ad hoc networks [12]. The language has been implemented as an interpreter written on top of the Java Virtual Machine. A J2ME version exists which can be deployed on PDAs.

We will use a typical collaborative ad hoc networking application to illustrate the language and the ambient reference abstraction. After a short description of this running example, we describe standard, sequential programming in AmbientTalk to familiarise the reader with the language's syntax and semantics. Subsequently, we cover concurrent and distributed programming.

3.1 Running Example

Consider a music player running on mobile devices such as PDAs or cellular phones. The music player contains a library of songs. When two people running the music player enter one another's personal area network (delineated by e.g. the bluetooth communication range of their cellular phones), the music players exchange their music library's index (not necessarily the songs themselves). After the exchange, the music player can calculate the percentage of songs both users have in common. If this percentage is high enough, the music player can e.g. warn the user that someone with a similar taste in music is nearby and display those songs in the other user's library.

3.2 Sequential Computation

AmbientTalk is a dynamically typed object-oriented language. Computation is expressed in terms of objects sending messages to one another. The following code excerpt shows the definition and use of a simple `Song` object in AmbientTalk:

```
def Song := object: {
  def artist := nil;
  def title := nil;
  def timesPlayed := 0;
  def init(artist, title) {
    self.artist := artist;
    self.title := title;
    self.timesPlayed := 0;
  };
  def play() { timesPlayed := timesPlayed + 1; /* play the song */ };
};
def s := Song.new("U2", "One");
s.play();
```

In this example, a prototypical song object is assigned to the variable `Song`. A song object has three fields, a constructor (always called `init` in AmbientTalk), and a method `play` to play the song. Sending `new` to an object creates a copy of that object, initialised using its `init` method.

3.3 Distributed Computation

AmbientTalk's concurrency model is based on that of *communicating event loops* as featured by the E programming language [13]. This concurrency model has its roots in the actor model of computation [14] and its incarnation in stateful active objects in languages such as ABCL/1[15]. In the model, regular objects are associated with at most one actor (a *vat* in E terminology) and each actor has an associated message queue. Every actor is associated with exactly one thread, the *event loop* which perpetually takes messages from its message queue and invokes the corresponding methods on its associated objects. Within the confines of an actor, computation happens sequentially and objects communicate using sequential message sending, as in Java or Smalltalk. AmbientTalk actors process incoming messages in a serial manner, to ensure that no race conditions can occur on the internal state of their associated objects.

Asynchronous Message Passing An object `a` owned by one actor can acquire a reference to an object `b` owned by another actor. In that case, `a` can only send messages to `b` *asynchronously*. When `a` sends a message to `b`, the message is placed in the incoming message queue of `b`'s actor. Only when the actor processes the message at a later point in time is `b`'s method invoked.

In the example scenario, each music player is modelled as an actor. Each such music player actor contains a music library, represented as a set of `Song` objects. When two such actors discover one another in the local ad hoc network, they exchange their music library index. Before a music player downloads the library index, it first asks for the size of the remote library. Given that `remotePlayer` denotes a reference to a remote music player (see section 4), this can be expressed as follows:

```
def sizeFuture := remotePlayer<-getSizeOfLibrary();
```

AmbientTalk borrows from the E language the syntactic distinction between sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`). An asynchronous message send always immediately returns a *future*, which is a placeholder for the actual return value. Once the return value is computed, it "replaces" the future object; the future is then said to be *resolved* with the value. Futures (also known as promises) are a frequently recurring abstraction in concurrent languages (e.g. in ABCL [15], Argus [16], E [13] and recently also in Java).

Futures are objects which can in turn be sent asynchronous messages. Those messages are accumulated within the future as long as it is unresolved. When the future is resolved, these messages are then forwarded to the resolved value. In the E language, it is possible to register a block of code with a future, which is executed asynchronously

when the future becomes resolved. AmbientTalk also allows the expression of such "inline event handlers", which are very useful when access to the actual return value of a message send is required. For example, if the user must be informed of how many songs another user is sharing, the size of the other user's music library must be printed on the screen. This can only happen when the `sizeFuture` from the previous example is resolved to an integer value:

```
when: sizeFuture becomes: { |size|
  // execution of this code is postponed until the future is resolved
  system.println("User is sharing ", size, " songs.");
} catch: { |exception| ... };
// code following when: is processed normally
```

If the asynchronously invoked method raises an exception, rather than returning a result, the corresponding future is resolved with the exception and the `catch` clause rather than the `when` clause of the above code is executed. This enables applications to catch asynchronously invoked exceptions in a way similar to the well-known `try-catch` abstraction of sequential languages.

Exporting Objects In order to make some objects available to remote actors and their objects, an actor can explicitly *export* objects that represent certain services. Because remote objects do not necessarily know the name or address of the exported service object, a service object is always exported together with a *service type*. A service type is a subtype of one or more other service types. Service types are not associated with a set of methods and are merely used to categorise which objects export what kinds of services[1].

In the music player example, each music player actor exports an `interface` object that can be used by other music players to start a communication session to exchange libraries. This object is exported with the service type `MusicPlayer`, as follows:

```
deftype MusicPlayer;

def interface := object: {
  def openSession(remotePlayer) {
    // return a session object (explained later)
  };
  def getSizeOfLibrary() { ... };
};

export: interface as: MusicPlayer;
```

From the moment an object is exported by its actor, it is discoverable by other actors by means of ambient references via its service type. This is explained in detail in the following section.

[1] Service types are best compared with empty Java interface types, like the typical "marker" interfaces used to merely tag objects. Example interfaces are `java.io.Serializable` and `java.lang.Cloneable`.

4 Ambient References

An ambient reference is a remote object reference that transparently discovers and binds to a remote object by means of a service type. For example, to discover a proximate music player, one creates an ambient reference initialised with the `MusicPlayer` service type, as follows:

```
def musicPlayerFuture := ambient: MusicPlayer;
```

The expression **`ambient:`** `MusicPlayer` initiates a service discovery request for a remote object exported as a `MusicPlayer` and immediately returns a future. When a matching object has been discovered, the future is resolved with an ambient reference bound to the discovered object. As usual, objects can start sending messages to the future before it is resolved, causing the future to accumulate those messages until a remote object has been discovered. One can regard this future as a dangling or unbound remote reference. When the future becomes resolved with an ambient reference, we refer to the remote object to which the ambient reference is bound as the ambient reference's *principal*.

Figure 1 depicts the situation where an ambient reference is asked for, but where no matching principal has yet been found. It shows two actors A and B. The wireless communication links of their host devices are represented as dotted circles which delimit their communication range. Each actor hosts a number of objects (white circles). B has exported an object using a service type symbolized as a diamond. A contains a future (gray circle) for an ambient reference that will bind to objects whose service type "matches" the diamond shape. Although the ambient reference does not yet exist, conceptually the future represents a dangling remote reference. Any messages sent to this future will be accumulated by the future until it is resolved.

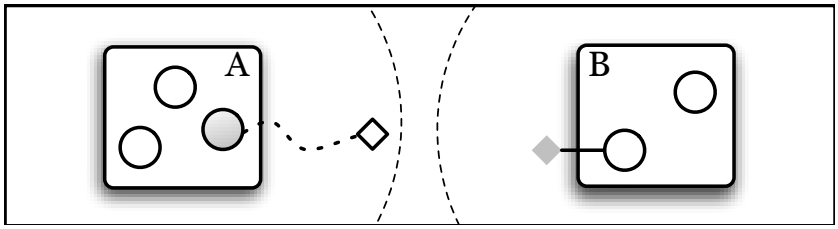

Fig. 1. A future for an ambient reference

Figure 2 depicts the situation where both devices move into one another's communication range. Because a matching service object has been found, `A` creates an ambient reference bound to this remote object and resolves the outstanding future with the bound ambient reference. Any messages that were accumulated in the future are forwarded to the ambient reference.

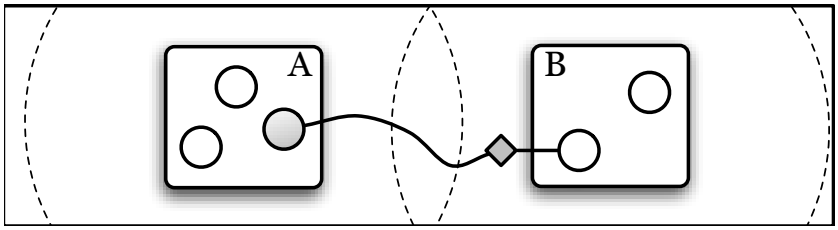

Fig. 2. A connected ambient reference

When the two host devices move back out of one another's communication range, the ambient reference does *not* break. Rather, it maintains the bond with the remote service, as depicted in figure 3. It follows that an ambient reference can be in two states: it can either be *connected* to its principal or *disconnected* from its principal. The influence of these states on message passing is explained in the following section.

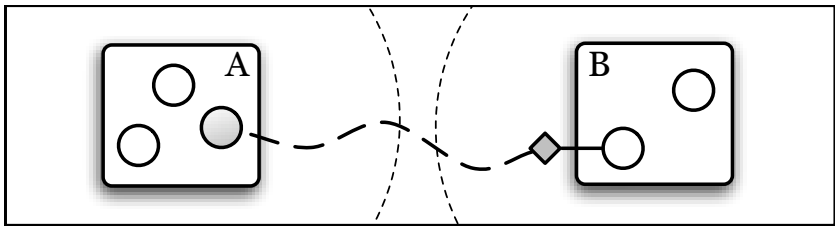

Fig. 3. A disconnected ambient reference

As explained in section 3.3, the resolved value of a future can be awaited using a `when` block. Because the discovery mechanism immediately returns a future for the ambient reference, objects can take explicit action when proximate services appear in their environment by attaching a `when` block to the future for the ambient reference:

```
def musicPlayerFuture := ambient: MusicPlayer;
when: musicPlayerFuture becomes: { |ambientReference|
  system.println("discovered new music player: ", ambientReference);
};
```

It is important to note that the code that exports the `interface` object, and the code above that creates an ambient reference is executed by all music player actors in the network. This enables music players to engage in true peer-to-peer communication: when a music player `A` and a music player `B` enter one another's communication range, `A` will discover `B`'s interface object via its ambient reference and `B` will discover `A`'s interface object via its ambient reference. The discovery is successful because the ser-

vice type of the ambient references, `MusicPlayer`, matches (i.e. is a subtype of) the corresponding service type of the exported `interface` object.

An ambient reference created by an actor will not bind to an object exported by that same actor. Indeed, if the object is local to the actor, it can be passed around by means of regular message passing without the need for a decoupled communication channel such as an ambient reference. Hence, in the example above, the ambient reference created by a music player will never bind to its own interface object. If multiple matching remote objects are available when an ambient reference is created, the reference binds to one single arbitrary matching object. Ambient references that may bind to multiple matching objects are not considered in this paper and are left as future work.

4.1 Message Passing

Ambient references follow the rules for inter-actor message passing and operate asynchronously. When a client object sends a message to an ambient reference, it does not wait for the message to be forwarded by the ambient reference to its principal. If the ambient reference is connected to its principal upon message reception, it forwards the message to the principal; if it is disconnected upon message reception, it accumulates the message internally and forwards it whenever it becomes reconnected at a later point in time. Hence, messages sent to ambient references are never lost, regardless of the internal state of the reference. Messages are guaranteed to be forwarded to a principal in the same order as they were received by the ambient reference. Recall that the principal is associated with an actor which ensures that incoming messages (sent by one or more ambient references) are executed serially.

In the music player example, once one music player has a reference to the `interface` object of another music player, it can ask the remote player to open a library exchange session by sending it the `openSession` message. The `interface` object implements this message as follows:

```
def openSession(remotePlayer) {
  def senderLib := Set.new(); // to store sender's music library
  object: {
    def downloadSong(artist, title) {
      senderLib.add(Song.new(artist, title));
      "ok"; // tell sender that song was successfully received
    };
    def endExchange() {
      // myLib and THRESHOLD are instance variables of this actor
      def matchRatio := (myLib.intersect(senderLib)).size() / myLib.size();
      if: (matchRatio >= THRESHOLD) then: {
        system.println("Found user with similar taste in music.");
      };
    };
  };
};
```

Note that the `openSession` method returns a new object which implements two methods which are used by a remote music player to send song information (`downloadSong`) and to signal the end of the library exchange (`endExchange`). A music player sends all of its own songs one by one to this session object after it has discovered a music player:

```
def musicPlayerFuture := ambient: MusicPlayer;
when: musicPlayerFuture becomes: { |ambientReference|
  system.println("discovered new music player: ", ambientReference);
  def session := ambientReference<-openSession(self);
  def iterator := myLib.iterator(); // to iterate over own music library
  def sendSongs() { // auxiliary function to send each song
    if: (iterator.hasNext()) then: {
      def song := iterator.next();
      when: session<-downloadSong(song.artist, song.title) becomes: { |ack|
          sendSongs(); // recursive call to send the rest of the songs
      } catch: { |exception| /* stops exchange (see section 4.2) */ };
    } else: {
      session<-endExchange();
    };
  };
  sendSongs();
};
```

The `session` object is again a future which will be resolved with an ambient reference that is bound to the object returned by the `openSession` method. The auxiliary function `sendSongs` sends the music player's songs one by one to the remote `session` object. This serial behaviour is guaranteed, because each subsequent `downloadSong` message is only sent after the previous one returned an acknowledgement (the simple `"ok"` string returned by the `downloadSong` method defined above).

4.2 Partial Failure Handling

The example application described above illustrates how the use of a loosely coupled communication abstraction (in this case an ambient reference) allows the application developer to abstract over transient disconnections: once the music players have established a library exchange session, they can disconnect from and reconnect to the network without hampering the control flow of exchanged messages. Note that the `catch:` clause in the previous code excerpt is normally not triggered when the ambient reference disconnects, it is only triggered if the invoked method raised an exception. Below, we describe how to trigger this `catch:` clause upon disconnection, such that it can also be used to perform failure handling if necessary.

Although an ambient reference allows a client object to safely abstract from its internal connection state, it is often useful for an application to be informed when a connection with a remote object is lost or reconnected. To this end, it is possible to install observers on an ambient reference which are triggered when the reference becomes disconnected or reconnected. The code below shows how a music player can notify the user whenever a proximate music player moves in and out of communication range:

```
when: musicPlayerFuture becomes: { |ambientReference|
  ...
  when: ambientReference disconnects: {
    system.println("music player disconnected: ", ambientReference);
  };
  when: ambientReference reconnects: {
    system.println("music player reconnected: ", ambientReference);
  };
};
```

The behaviour of ambient references is designed to allow the developer to abstract over transient network failures. However, a developer may want to perform failure handling from the moment an ambient reference has been disconnected for longer than a certain timeout period. The question then becomes how the developer can reasonably deal with all of the messages that have accumulated in the ambient reference while it was disconnected.

To deal with failures, ambient references support one final operation: a developer may *rebind* an ambient reference to point to another principal object. This object may be another remote object, but often it will be a local object that acts as a failure handler for all of the messages that were accumulated by an ambient reference *and* for all of the messages sent to the ambient reference from the moment it has been rebound.

In order to adapt the music player to terminate the library exchange upon disconnection, the ambient reference can be rebound to a failure handler object by means of a `disconnects:` observer (perhaps only after a certain timeout period). This failure handler can then reply to every message by raising an exception. This will resolve each message's future with that exception, which in turn triggers the `catch:` clause of any registered `when` blocks on that future. In the second code excerpt of the previous section, this would trigger the `catch:` clause for the `downloadSong` message, which enables the library exchange protocol to terminate smoothly.

5 Evaluation

Now that ambient references have been properly introduced, we can evaluate them with respect to the requirements postulated in section 2.2.

Requirement 1 Ambient references decouple the communicating objects in time. When a client object first requests an ambient reference, it will immediately get a future for the reference, allowing it to continue its computation until a suitable service object has been found. Moreover, clients are not obliged to send messages via an ambient reference only when it is connected, because an ambient reference properly accumulates messages while it is disconnected.

Requirement 2 Ambient references decouple the communicating objects in space by means of service types. Objects address one another by means of the services they describe and do not know or need to know the address of the hosting device. An exported service object also does not necessarily know which or how many client objects refer to it via an ambient reference. Thanks to the use of futures, a service can easily reply to its clients without referring to them explicitly simply by returning values from its invoked methods. These return values implicitly resolve the futures held by clients, allowing them to process replies asynchronously.

Requirement 3 Ambient references decouple the control flow of client and service objects. Client objects send messages asynchronously and can await results asynchronously by registering blocks of code with the futures. Exported service objects are hosted by an actor, whose incoming message queue ensures that messages can be received even while the service object is busy processing another message.

Requirement 4 Via the registration of dedicated observers which trigger upon the disconnection or reconnection of a principal, an application can have an up-to-date

view of the internal state of an ambient reference without affecting other application code that sends messages and receives replies via that ambient reference. Failure handling can be performed by rebinding the ambient reference to a dedicated failure handler object. Any undelivered messages accumulated by the ambient reference are then forwarded to that object.

Because they adhere to the first three requirements, ambient references form a loosely coupled communication channel between objects, without sacrificing the remote object referencing abstraction. The contribution of the ambient reference abstraction lies in the combination of:

1. An abstract type-based discovery mechanism that immediately returns a future when a discovery request is made. The future represents a "not yet discovered" object. This enables a client object that needs to interact with a remote object to immediately interact with the future as if that future already *is* the remote object.
2. Asynchronous message passing semantics which allows one to abstract over the state of the connection with the remote object. This is achieved by implicitly accumulating messages within the remote reference itself while it is disconnected.
3. Using observers to keep track of changes in the state of the connection of the reference, such that failure handling can be performed separately from the main use of an ambient reference as a time- and space-decoupled communication channel.

While none of these mechanisms are by themselves novel, the contribution of ambient references lies in the combination of service discovery and asynchronous communication into one coherent language construct and its application to mobile ad hoc networks.

6 Related Work

We categorise related work into **1)** object-oriented referencing abstractions, **2)** tuple space architectures and **3)** publish/subscribe architectures. For each approach, we summarise their applicability to mobile networks and how they resemble or differ from ambient references.

6.1 Object-oriented Referencing Abstractions

M2MI The design of ambient references has been inspired by the notion of a *handle* in the many-to-many invocations (M2MI) paradigm [17]. M2MI is a paradigm for building collaborative systems deployed on wireless proximal ad hoc networks. M2MI handles use Java interfaces to decouple remote objects in space. M2MI handles also employ asynchronous message passing.

Although M2MI has influenced the design of ambient references, there are some important differences. First, M2MI handles do not decouple participants in time: if a message is sent to an object which is not in communication range at that time, the message is lost. Second, M2MI invocations require that asynchronous messages do

not return a value or throw an exception. This makes it more cumbersome to express request/response interactions due to the lack of futures.

Actors In the actor model of computation [14], actors refer to one another via *mail addresses*. When an actor sends a message to a recipient actor, the message is placed in a mail queue and is guaranteed to be eventually delivered by the actor system. One can regard a mail address as a "remote actor reference". Although such a reference decouples actors in time and in synchronisation (actors communicate strictly asynchronously), it does not decouple them in space. A mail address represents a unique actor and does not allow actors to discover one another by means of an abstract description.

E The E language [13, 18] is designed for writing secure peer-to-peer distributed programs in open networks. Our notion of distinguishing intra-actor communication (synchronous message passing) from inter-actor communication (asynchronous message passing) is directly derived from E's similar message passing semantics. E pioneered the `when` construct to deal with the resolution of futures (or promises) in an entirely non-blocking, event-driven manner.

E was designed for open distributed systems, but not specifically for mobile ad hoc networks. This shows in a number of important differences with respect to the semantics of remote references in AmbientTalk. First, a network disconnection in E immediately *breaks* the remote reference: any message sent after the disconnection is not stored, and the message's promise is resolved with an exception. Hence, E's remote references do not decouple participants in time and are not designed to express communication over volatile connections. E does feature a hook similar to the one introduced in AmbientTalk to enable the programmer to react upon the disconnection of a remote reference. There is no corresponding hook for reconnection in E, because once broken, a remote reference in E remains broken.

To regain connectivity after a network failure, E features special references, known as *sturdy references*, which do survive network failures. Sturdy references, however, are created by means of an explicit address (in the form of a URL) and are meant to denote specific objects, so they do not decouple objects in space.

Jini The Jini architecture for network-centric computing [19, 20] is a platform for service-oriented computing built on top of Java. Jini introduces the notion of lookup services. Services may advertise themselves by uploading a proxy to the lookup service. Clients search the network for lookup services and may launch queries for services they are interested in. Clients can download the advertised proxy of a remote service and may interact with the remote service through the proxy. Java interface types are used to describe and discover services. Our use of service types to describe to which kinds of objects an ambient reference may bind has been inspired by this mechanism.

Jini is primarily a framework for bringing clients and services together in a network with minimal administrative infrastructure. Once a client has downloaded a service proxy, the proxy is the communication channel to the service. This proxy may be implemented however the service sees fit. For example, it is possible to construct proxy references which e.g. accumulate messages when the remote service is disconnected to achieve decoupling in time. Hence, Jini's *architecture* is flexible enough to accomodate ambient references. However, to the best of our knowledge, Jini does not by default

offer any advanced remote "service" references. By default, the proxies advertised by services communicate synchronously with their service.

6.2 Tuple Spaces

Linda and LIME Tuple spaces as originally introduced in the coordination language Linda [8] have received renewed interest by researchers in the field of mobile computing. Adaptations of tuple spaces for mobile computing, such as LIME [4], feature tuple spaces local to each device which are merged into a *federated transiently shared tuple space* when joining the network. In the tuple space model, processes communicate by inserting and removing tuples from the shared tuple space, which acts like a globally shared memory. Decoupling in time is achieved because processes can insert and retract tuples independently. Decoupling in space is achieved because the publisher of a tuple does not necessarily specify, or even know, which process will consume the tuple. Synchronisation decoupling is not adhered to in the original Tuple space model: although tuple insertion is asynchronous, there exist synchronous (blocking) operations to extract tuples from the tuple space.

As the need for total synchronization decoupling became apparent for mobile networks, extensions of the model such as LIME provide *reactions* which are callbacks that trigger asynchronously when a matching tuple becomes available in the tuple space. LIME adheres to requirement 4, connection awareness, by introducing a read-only, system-maintained tuple space whose tuples represent metadata, such as the hosts that are currently connected. Registering reactions on such tuples achieves a connection awareness strategy similar to one using the observers introduced in section 4.2.

The main difference between LIME and ambient references lies in their employed communication paradigm. Ambient references foster a more object-oriented programming style because communication is one-to-one rather than one-to-many and happens by means of asynchronous message sends (which capture the communication of both request *and* reply in one single abstraction).

ActorSpace The inability of mail addresses to represent unknown, undiscovered actors have been addressed in the ActorSpace model [21]. This model is a unification of concepts from both the actor model and the tuple space model of Linda. Callsen and Agha note that, on the one hand, the actor model provides a secure model of communication as an actor may only communicate with actors whose mail address it has been explicitly given via message passing. On the other hand, this disallows actors to get acquainted with other actors in a time- and space-decoupled manner.

The ActorSpace model augments the actor model with *patterns*, denoting an abstract specification of a group of actors. The actor model's `send` primitive, which normally takes a receiver mail address and a message and sends the message to the corresponding mail address, is changed such that `send` now also accepts a pattern rather than a mail address. For example, `send("MusicPlayer", "getSizeOfLibrary")` can be received by any actor whose own name matches the pattern within the context of a so-called *actorspace*. The `send` primitive delivers the message to a non-deterministically chosen matching actor. Although this behaviour is good when it does not matter to the sender which specific actor receives the message (e.g. when the re-

ceiver is a replicated file server), it is not similar to an ambient reference in the sense that multiple messages sent to the same pattern may be received by different actors.

6.3 Publish/Subscribe Architectures

LPS Location-based Publish/Subscribe (LPS) [6] is a publish/subscribe architecture designed specifically for the collaboration of mobile ad hoc applications. The main difference between LPS and traditional publish/subscribe architectures is that event dissemination and reception is bounded in physical space: a publisher defines a *publication range* and a subscriber defines a *subscription range*. Only when the publication range of the publisher and the subscription range of the subscriber overlap is an event disseminated to the subscriber.

STEAM Scalable Timed Events and Mobility (STEAM) is an event-based middleware designed for supporting collaborative applications in mobile ad hoc networks [22, 2]. STEAM builds upon the observation that the physically closer an event consumer is located to an event producer, the more interested it may be in those events. It allows events disseminated by producers to be filtered based on geographical location using *proximities*. Proximities are first-class representations of a physical range, which may be absolute or relative (i.e. a relative proximity denotes a surrounding area relative to a mobile node).

Both LPS and STEAM are publish/subscribe middleware and have no notion of remote object references. Applications are structured as a suite of event handlers and do not use the message passing abstraction to engage in distributed communication. As publish/subscribe architectures, they naturally decouple participants in time, space and synchronization. It is not immediately clear how the models allow applications to perform failure handling when publishers or subscribers disconnect.

7 Research Status and Future Work

Ambient references have been implemented as part of the AmbientTalk programming language[2]. The mobile music player used as a running example in this paper is also available for download as an example AmbientTalk program.

We are currently investigating a family of ambient reference abstractions with slight variations on the semantics presented here. For example, we are experimenting with ambient references that bind to all available matching services (rather than a single one). Such ambient references form a group communication channel which broadcast messages to all matching objects. Other kinds of ambient references vary in their binding semantics with their principal. As explained in section 4, when the remote principal becomes disconnected, the ambient reference remains bound to it. Sometimes it is more appropriate to clear the binding when a disconnection occurs, such that the ambient reference can *rebind* to other available matching objects (e.g. in the case of a replicated service where the identities of the replicated exported objects themselves are not important).

[2] The language is available at `http://prog.vub.ac.be/amop/at/download`

Another aspect of ambient references which has currently not yet been thoroughly addressed is the garbage collection of exported objects. For the purposes of this paper, it is assumed that exported service objects are long-lived objects which have to be unexported explicitly in order to be reclaimed. Once an exported object is advertised on the network, it can no longer be reclaimed automatically because at any point in time an ambient reference may bind to it. Moreover, in mobile ad hoc networks where relationships between devices are short-lived, traditional cooperative distributed garbage collection approaches become impractical. As illustrated by networking technology such as Jini, the notion of a *leased* reference provides more robust garbage collection in the face of both transient and permanent disconnections [19]. In future work, we would like to integrate leasing with ambient references. Using leases, exported objects can be unexported when their lease expires, while ambient references that still refer to the exported object are responsible for renewing the lease in time.

8 Conclusions

This paper put forward ambient references as a loosely coupled object-oriented coordination abstraction for mobile ad hoc networks. The conception of this abstraction is motivated by the observation that: **a)** mobile networks require loosely coupled communication abstractions and **b)** traditional distributed object-oriented computing abstractions do not fit these requirements which **c)** requires object-oriented programs to leave the object-oriented paradigm when performing distributed communication.

The contributions of this paper are: **a)** an analysis of the requirements for coordination abstractions for mobile ad hoc networks and **b)** the introduction of a coordination abstraction for mobile ad hoc networks in the guise of a language construct which is both object-oriented (it is an object reference carrying messages) *and* loosely coupled. We have exemplified ambient references by means of a typical collaborative application, developed in the AmbientTalk programming language.

Acknowledgments. The authors would like to thank Stijn Mostinckx, Elisa Gonzalez and Jorge Vallejos for the many discussions that have contributed to the ideas presented here. The authors also thank the anonymous referees for their valuable comments and suggestions for improvement.

References

1. Weiser, M.: The computer for the twenty-first century. Scientific American (1991) 94–100
2. Meier, R., Cahill, V., Nedos, A., Clarke, S.: Proximity-based service discovery in mobile ad hoc networks. In: Distributed Applications and Interoperable Systems. Springer (2005) 115–129
3. Mascolo, C., Capra, L., Emmerich, W.: Mobile Computing Middleware. In: Advanced lectures on networking. Springer-Verlag New York, Inc. (2002) 20–58
4. Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proceedings of the The 21st International Conference on Distributed Computing Systems, IEEE Computer Society (2001) 524–536

5. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the TOTA middleware. In: PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications, Washington, DC, USA, IEEE Computer Society (2004) 263
6. Eugster, P., Garbinato, B., Holzer, A.: Location-based publish/subscribe. Fourth IEEE International Symposium on Network Computing and Applications (2005) 279–282
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2) (2003) 114–131
8. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7**(1) (1985) 80–112
9. Joseph, A.D., deLespinasse, A.F., Tauber, J.A., Gifford, D.K., Kaashoek, M.F.: Rover: a toolkit for mobile information access. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95), Colorado (1995) 156–171
10. Eugster, P.T., Guerraoui, R., Damm, C.H.: On objects and events. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2001) 254–269
11. Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley Longman Ltd., Essex, UK (1999)
12. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented Programming in Ambienttalk. In Thomas, D., ed.: Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP). Volume 4067 of Lecture Notes in Computer Science., Springer (2006) 230–254
13. Miller, M., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In Nicola, R.D., Sangiorgi, D., eds.: Symposium on Trustworthy Global Computing. Volume 3705 of LNCS., Springer (2005) 195–229
14. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
15. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press (1986) 258–268
16. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, ACM Press (1988) 260–267
17. Kaminsky, A., Bischof, H.P.: 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, New York, NY, USA, ACM Press (2002) 72–73
18. Miller, M.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, John Hopkins University, Baltimore, Maryland, USA (2006)
19. Waldo, J.: The Jini Architecture for Network-centric Computing. Commun. ACM **42**(7) (1999) 76–82
20. Arnold, K.: The jini architecture: Dynamic services in a flexible network. In: 36th Annual Conference on Design Automation (DAC'99). (1999) 157–162
21. Callsen, C.J., Agha, G.: Open heterogeneous computing in ActorSpace. Journal of Parallel and Distributed Computing **21**(3) (1994) 289–300
22. Meier, R., Cahill, V.: Exploiting proximity in event-based middleware for collaborative mobile applications. In: Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03). (2003)