AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks

Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, Wolfgang De Meuter
Programming Technology Lab
Vrije Universiteit Brussel
Brussels, Belgium
tvcutsem|smostinc|egonzale|jededeck|wdmeuter@vub.ac.be

***Abstract*—In this paper, we describe AmbientTalk: a domain-specific language for orchestrating service discovery and composition in mobile ad hoc networks. AmbientTalk is a distributed object-oriented programming language whose actor-based, event-driven concurrency model makes it highly suitable for composing service objects across a mobile network. The language is a so-called ambient-oriented programming language which treats network partitions as a normal mode of operation. We describe AmbientTalk's object model, concurrency model and distributed communication model in detail. We also highlight the major influences from other languages and middleware that have shaped AmbientTalk's design.**

***Index Terms*—distributed languages, actors, events, publish/subscribe, service discovery, service composition, mobile networks, pervasive computing**

I. Introduction

With the introduction of ever smaller, handheld computing devices over the past few decades, there has been a tremendous increase in research into *mobile ad hoc networks*. Such networks are composed of mobile devices equipped with wireless communication technology and are often not administered. This hardware constellation is often claimed to serve as a fruitful basis for many pervasive and ubiquitous computing [1] 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 appeared.

Although there has been a lot of active research with respect to mobile computing middleware [3], there has been little innovation in the field of programming language research to tackle the issues raised by mobile networks. Although distributed programming languages are rare, they form a suitable development tool for encapsulating many of the complex issues engendered by distribution [4], [5]. The distributed programming languages developed to date have either been designed for high-performance computing (e.g. X10 [6]), for reliable distributed computing (e.g. Argus [7]) or for general-purpose distributed computing in fixed, stationary networks (e.g. Emerald [8], Obliq [9], E [10]). None of these languages have been explicitly designed for mobile networks. They lack the language support necessary to deal with the radically different network topology.

In this paper, we introduce *AmbientTalk*, a distributed object-oriented programming language which has been designed for mobile ad hoc networks from the ground up. AmbientTalk is a small, dynamically typed object-oriented language. What makes the language suitable for composing services across a mobile network is its actor-based, event-driven concurrency model in combination with its built-in peer-to-peer, publish/subscribe service discovery abstractions. AmbientTalk has previously been described as an exemplar of the *ambient-oriented programming* (AmOP) paradigm [11]. We will revisit the key characteristics of the AmOP paradigm in section III. The language described in this paper is actually AmbientTalk/2, an updated version of the language as it is presented in [11]. However, we will simply refer to the updated language as AmbientTalk because it supplants its predecessor while staying true to its fundamental characteristics.

The paper is structured according to three large parts. In sections II and III, we describe the salient features of mobile ad hoc networks in more detail, revisit the ambient-oriented programming paradigm and motivate the need for novel programming language support. Subsequently, in sections IV, V and VI we introduce the AmbientTalk language itself. We first describe its standard object-oriented features, then its concurrent and finally its distributed language features. The final part of the paper describes the advancements with respect to previous work and discusses related work in the field of programming languages and middleware which have influenced the design of the AmbientTalk language.

II. Mobile Ad Hoc 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:

Draft version. Revised version accepted at the XXVI International Conference of the Chilean Computer Science Society, SCCC 2007.

Tom Van Cutsem is a Research Assistant of the Fund for Scientific Research, Flanders (F.W.O.). Stijn Mostinckx is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

- **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* network partitions should not affect an application, allowing both parties to continue their collaboration where they left off. Dealing with partial failures is not a new ingredient of distributed systems, but 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. In mobile networks, disconnections become so omnipresent that they should be considered the rule, rather than an exceptional case.
- **Zero Infrastructure.** In a mobile network, devices that offer services spontaneously join with and disjoin from the network. Moreover, a mobile ad hoc network is often not manually administered. 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. This lack of infrastructure requires a *peer-to-peer* communication model, where services can be directly advertised to and discovered on proximate devices.

Any application designed for mobile ad hoc networks has to deal with the above phenomena. Because the phenomena are universal, an appropriate computational model can and should be developed which eases distributed programming in a mobile network by taking these phenomena into account from the ground up. Moreover, because the effects engendered by partial failures or the absence of remote services often pervade the entire application, the above phenomena are not easily hidden behind traditional library abstractions. Therefore, distribution is often dealt with in dedicated middleware or programming languages.

III. Ambient-oriented Programming Revisited

In previous work, we have described an ambient-oriented programming language as a programming language that adheres to a set of well-defined characteristics [11]. The two characteristics which deal directly with the hardware characteristics of mobile networks described in the previous section are detailed below.

A. Non-blocking Communication

In an AmOP language, all distributed communication is *non-blocking*, i.e. asynchronous. The main reason behind this strict asynchrony is that communicating parties remain *loosely-coupled*. It is this loose coupling which significantly reduces the impact of volatile connections on a distributed application. With respect to communication, two degrees of coupling between communicating parties can be distinguished, as explained in detail in [12]:

- **Decoupling in Time**: *The communicating parties do not need to be online at the same time.*
  Decoupling in time implies 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.
- **Synchronisation Decoupling**: *The control flow of communicating parties is not blocked upon sending or receiving.*
  Synchronisation decoupling implies 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.

B. Ambient Acquaintance Management

An AmOP language should have built-in support for *ambient acquaintance management*: the discovery and management of proximate devices and their hosted services. However, the way in which communicating parties can discover one another reveals yet another degree of coupling with important repercussions in mobile ad hoc networks:

- **Decoupling in Space**: *The communicating parties do not need to know each other beforehand* [12].
  Decoupling in space implies that communicating parties do not necessarily need to know one another's exact address or location. However, this means that communicating parties must 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.

Ambient acquaintance management implies more that simply the discovery of new parties. It also implies that communicating parties must be able to keep an up-to-date view of which participants are (dis)connected. At first glance, this requirement seems to somewhat contradict the purpose of non-blocking communication as described previously, 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.

IV. The AmbientTalk Language

In this section, we introduce AmbientTalk as an object-oriented programming language. Even though AmbientTalk is a domain-specific language for distributed programming, it remains a full-fledged object-oriented language in its own right. AmbientTalk inherits most of its standard language features from Self, Scheme and Smalltalk. From Scheme, it inherits the notion of true lexically scoped closures. From Self and Smalltalk, it inherits an expressive block closure syntax, the representation of closures as objects and the use of block closures for the definition of control structures. AmbientTalk's object model is derived from Self: classless, slot-based objects using delegation [13] as a reuse mechanism.

A. Running Example: the Ubiquitous Flea Market

We will describe AmbientTalk by means of a concrete example ad hoc networking application, called the ubiquitous flea market [14], later named *AgoraM* [15]. This example application is meant to run on the cellular phone of the user. Using the application, users can either advertise items they wish to sell or place a demand for items they wish to buy. What makes this example an ad hoc application is that buyers and sellers are only matched when they are proximate, e.g. if they have joined the same ad hoc network. For example, the flea market applications could use the user's personal area network delimited by the cellular phone's bluetooth communication range. When a potential buyer/seller has been found for an item, the user is notified and contact details are exchanged. The fact that the ubiquitous flea market only matches proximate buyers and sellers is useful if the item to be sold is immediately required by the buyer. For example, it can be used to buy or sell concert or sports tickets directly at the venue itself [15].

Although the ubiquitous flea market example is small, it is a representative application because it embodies all characteristics of a typical mobile ad hoc networking application. The applications have to discover one another's supplied/demanded items without any predefined infrastructure. Furthermore, communication is easily disrupted because of the unpredictable movement of the users.

B. AmbientTalk Objects

AmbientTalk is a dynamically typed, object-based language. Computation is expressed in terms of objects sending messages to one another. Objects are not instantiated from classes. Rather, they are either created *ex-nihilo* or by cloning and adapting existing objects. A central abstraction in the ubiquitous flea market example is the item to be traded between peers. The following code excerpt shows the prototype definition of such item objects.

```
def Item := object: {
  def category; // a type classifying the item
  def description; // a string describing the item
  def contactDetails; // string describing contact details
  def init(cat, desc, contact) {
    category := cat;
    description := desc;
    contactDetails := contact;
  };
  def getContactInfo(buyerInfo) {
    contactDetails; // return the contact details
  };
  def placeSupply() { ... };
  def placeDemand() { ... };
};
```

The above code defines a new anonymous object and binds it to a variable named `Item`. This object serves as a prototypical item object, defining a number of fields to store the item's state and a number of methods to define useful behaviour, which is described in more detail later. This prototypical object can be instantiated to create new items:

```
def ticket := Item.new(Ticket,description,phoneNo);
```

Every object understands the message `new`, which creates a clone (a shallow copy) of the receiver object and initializes the clone by invoking its `init` method with the arguments that were passed to `new`. Hence, the `init` method plays the role of "constructor" for AmbientTalk objects. AmbientTalk's object instantiation protocol closely corresponds to class instantiation in class-based languages, except that the new object is a clone of an existing object, rather than an empty object allocated from a class.

AmbientTalk provides support for block closures reminiscent of those in Self and Smalltalk. A block closure is an anonymous function object that encapsulates a piece of code and the bindings of lexically free variables and `self`. Block closures are constructed by means of the syntax `{ |args| body }`, where the arguments can be omitted if the block takes no arguments. The following code excerpt shows a typical use of blocks to iterate over all of the elements of an array storing all items provided by the ubiquitous flea market application.

```
suppliedItems.each: { |item|
  system.println(item)
}
```

Block closures are frequently used in AmbientTalk to represent *delayed* computations, e.g. for implementing control structures but also for implementing nested event handlers, as will be described later. Note that AmbientTalk supports both traditional canonical syntax (e.g. `o.m(a,b,c)`) as well as keyworded syntax (e.g. `o.at: key put: value`) for method definitions and message sends.

V. Concurrent Programming in AmbientTalk

In AmbientTalk, concurrency is spawned by actors: one AmbientTalk virtual machine may host multiple actors which

execute concurrently. AmbientTalk's concurrency model is based on the communicating event loops model of the E programming language [10], which is itself an adaptation of the well-known actor model [16]. The E language combines actors and objects into a unified concurrency model. Unlike previous actor languages such as Act1 [17], ABCL [18] and Actalk [19], actors are not represented simply as "active objects", but rather as *vats* (containers) of regular objects, shielding them from harmful concurrent modifications.

Before describing how actors have been integrated in the AmbientTalk language, we first highlight the fundamental concurrency properties of event loop concurrency, on which AmbientTalk is based.

A. Event Loop Concurrency

The E language's communicating event loops and – as will be described later – AmbientTalk's actors employ an event-driven concurrency model, as opposed to traditional multi-threaded concurrency. In an event-driven model, an *event loop* is a thread of execution that perpetually processes *events* from its *event queue* by invoking a corresponding *event handler*. In addition, an event loop model can enforce three essential concurrency control properties:

- **Serial Execution**: *An event loop processes incoming events from its event queue one by one, i.e. in a strictly serial order.*
  As a consequence of serial execution, the handling of a single event is atomic with respect to other events. Hence, race conditions on an event handler's state caused by concurrent processing of events cannot occur.
- **Non-blocking Communication**: *An event loop* never *suspends its execution to wait for another event loop to finish a computation. Rather, all communication between event loops occurs strictly by means of asynchronous event notifications.*
  As a consequence of non-blocking communication, event loops can never deadlock one another. However, in order to guarantee progress, an event handler should not execute e.g. infinite `while` loops. Rather, long-running actions should be performed piecemeal by scheduling events recursively, such that an event loop always gets the chance to respond to other incoming events. The only situation where an event loop can be suspended is when its event queue is empty.
- **Exclusive State Access**: *Event handlers and their associated state belong to a single event loop. In other words, an event loop has* exclusive *access to its mutable state.*
  Because event handlers are not shared between event loops, they never have to lock mutable state. Mutating another event loop's state has to be performed indirectly, by asking the event loop to mutate its own state via an event notification.

Event loop concurrency avoids deadlocks and certain race conditions *by design*. The non-determinism of the system is confined to the order in which events are processed. In standard pre-emptive thread-based systems, the non-determinism is more substantial because threads may interleave upon each single instruction. In the following section, we describe how the abstract event loop model is incorporated into the AmbientTalk language.

B. AmbientTalk actors

In AmbientTalk, actors are not represented as active objects, but rather as event loops: the event queue is represented by an actor's message queue, events are represented as messages, event notifications as asynchronous message sends, and event handlers are represented as (the methods of) regular objects. The actor's event loop thread perpetually takes a message from the message queue and invokes the corresponding method of the object denoted as the receiver of the message. Messages are processed serially to avoid race conditions on the state of regular objects.

In AmbientTalk, each object is said to be *owned* by exactly one actor. Only an object's owning actor may directly execute one of its methods. Objects owned by the same actor may communicate using standard, sequential message passing or using asynchronous message passing. 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()`). It is possible for objects owned by an actor to refer to objects owned by other actors. Such references that span different actors are named *far references* (the terminology stems from E [10]) and only allow asynchronous access to the referenced object. Any messages sent via a far reference to an object are enqueued in the message queue of the owner of the object and processed by the owner itself.

Figure 1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the event loop threads of the actors which perpetually take messages from their message queue and synchronously execute the corresponding methods on the actor's owned objects. An event loop thread never "escapes" its actor boundary. When communication with an object in another actor is required, a message is sent asynchronously via a far reference to the object. For example, when `A` sends a message to `B`, the message is enqueued in the message queue of `B`'s actor which eventually processes it.

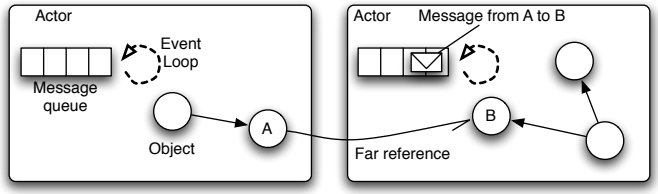

Fig. 1. AmbientTalk actors as communicating event loops

C. Asynchronous Message Passing

In AmbientTalk, asynchronous messages can be sent between objects owned by the same or by different actors. In the

case where both sender and receiver are owned by the same actor, the message is simply added to the owner's message queue and parameters are passed *by reference*, exactly as is the case with synchronous message sending. For inter-actor message sends, where an object sends an asynchronous message via a far reference to an object owned by another actor, objects are parameter-passed *by far reference*: the parameter of the invoked method will be bound to a far reference to the object. Objects that have declared themselves to be serializable form an exception. Serializable objects are instead passed by (deep) copy. This allows the recipient actor to operate on the copy synchronously, without additional inter-actor communication and without violating the exclusive state access property.

To illustrate asynchronous message passing more concretely, consider the ubiquitous flea market again. Each cellular phone runs an AmbientTalk application consisting of a single actor representing the ubiquitous flea market. When two such actors discover one another in the ad hoc network, they exchange their supplied and demanded items. This is described in more detail in section VI. When a matching item is found, the buyer explicitly has to ask for the seller's contact details, passing along its own contact details. Given that `adItem` denotes a far reference to the advertised item of another actor, the contact details of the user of the supplied item can be requested as follows:

```
def contactFut := adItem<-getContactInfo(myContact);
```

An asynchronous message send 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. In AmbientTalk, 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, accumulated messages are 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 "in-line event handlers", which are very useful when access to the actual return value of a message send is required. For example, the contact details of the user that supplied the found item can only be printed to the screen when the `contactFut` future is resolved to a string value:

```
when: contactFut becomes: { |contactInfo|
  // execution is postponed until future is resolved
  system.println("Found item, contact: " + contactInfo);
} catch: { |exception| ... };
// code following when: is processed immediately
```

The `when:becomes:catch:` function takes a future and two closures as arguments, and registers the closures as *observers* on the future. If the future is resolved to a proper value, the closure passed as the `becomes:` is applied with the resolved value as parameter. If the asynchronously invoked method raises an exception, rather than returning a value, the corresponding future is resolved with the exception and the closure passed as the `catch:` argument is applied to the exception. This enables applications to catch asynchronously raised exceptions in a way similar to the well-known `try-catch` abstraction of sequential languages. The execution of either of the above closures is always scheduled in the owning actor's message queue, such that their execution is serialised w.r.t. other messages processed by the actor.

VI. Distributed programming in AmbientTalk

In AmbientTalk, two objects are said to be *local* when they are owned by the same actor. Objects are considered *remote* when they are owned by different actors, even if those actors are hosted by the same virtual machine. Of course, within one virtual machine there is no notion of partial failure: either all actors within one VM are alive, or they have all crashed. Nevertheless, AmbientTalk abstracts from the physical location of actors and considers actors as the unit of distribution. Because objects residing on different devices are necessarily owned by different actors, the only kinds of object references that can span across different devices are far references. This ensures by design that all distributed communication is asynchronous.

A. Far References and Partial Failures

By admitting far references to cross virtual machine boundaries, we must specify their semantics in the face of partial failures. AmbientTalk's far references are by default resilient to network disconnections. When a network failure occurs, a far reference to a disconnected object starts buffering all messages sent to it. When the network partition is restored at a later point in time, the far reference flushes all accumulated messages to the remote object in the same order as they were originally sent. Hence, messages sent to far references are never lost, regardless of the internal connection state of the reference. Making far references resilient to network failures by default is one of the key design decisions that make AmbientTalk's distribution model suitable for mobile ad hoc networks, because temporary network failures have no immediate impact on the application's control flow.

Far references have been intentionally made resilient to transient partial failures. This behaviour is desirable in mobile networks because it can be expected that many partial failures are the result of temporary network partitions. However, perhaps a remote device has crashed beyond recovery, or it has moved out of the wireless communication range and does not return. Such persistent failures also need to be dealt with.

To cope with persistent failures, AmbientTalk uses the concept of leasing [20]. A *lease* denotes the right to access a resource for a limited amount of time. At the discretion of the owner of the resource a lease can be renewed, prolonging access to the resource. In AmbientTalk, far references play the role of the lease and the objects they refer to play the role of the resource. Hence, a far reference only provides access to a remote object for a limited amount of time. However, as long as the far reference is actively being used (i.e. messages are sent via the reference to the remote object), its lease is transparently renewed. Upon a network partition, the lease

cannot be renewed and will expire if the disconnection outlasts the lease period.

When a far reference's lease eventually expires, it will resolve the future attached to any messages sent to it with an exception, signalling to the sender that its message could not be delivered. Leases are not only important to make the sending party aware of persistent failures, they also have important benefits for memory management. Once all leases for a remote object have expired, the system may garbage collect the object, provided no more local references refer to it. Hence, leases are an important enabler of distributed garbage collection. If a far reference would have no upper bound on its access time, the remote object it refers to could never be reclaimed upon a disconnection because of the failure semantics of far references: when a far reference reconnects the remote object should not have been reclaimed.

B. Exporting Objects as Services

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

In AmbientTalk, service objects are exported by means of a *type tag*. Type tags are a lightweight classification mechanism, used to categorise objects explicitly by means of a nominal type. One use of type tags in AmbientTalk is to provide an intensional description of what kinds of services an object provides to remote objects. In AmbientTalk, a type tag can be a subtype of one or more other type tags, and one object may be tagged with multiple type tags. Although type tags are not used for static type checking, they are best compared with empty Java interface types, like the typical "marker" interfaces used to merely tag objects (e.g. `java.io.Serializable` and `java.lang.Cloneable`).

In the ubiquitous flea market scenario, when the user supplies an item for other users to buy, the supplied item object is exported. It is assumed that different items are classified according to type tags. Hence, the item advertisement is exported by means of the type tag stored in the advertised item's `category` field:

```
def placeSupply() {
  def pub := export: self as: self.category;
  // this object can be used to unexport the advertisement
  pub;
}
```

From the moment an object is exported, it is discoverable by objects owned by other actors by means of its associated type tag. The `export:as:` function returns an object which can be used to take the exported object offline again, by invoking `pub.cancel()`. How remote objects can acquire a reference to the exported object is explained in detail in the following section.

C. Service Discovery

AmbientTalk employs a publish/subscribe service discovery protocol. A publication corresponds to exporting an object by means of a type tag. The type tag serves as a *topic* known to both publishers and subscribers [12]. A subscription takes the form of the registration of an event handler on a type tag, which is triggered whenever an object exported under that tag has become available in the ad hoc network.

In the ubiquitous flea market application, the user places a demand for a certain item by invoking the item's `placeDemand` method. This method subscribes to the item's `category` type tag, such that it can be notified whenever a matching item has become available:

```
def placeDemand() {
  def sub := whenever: category discovered: { |adItem|
    def contactFut := adItem<-getContactInfo(myContact);
    // notify user of potentially interesting item
  };
  // this object can be used to cancel the subscription
  sub;
};
```

The `whenever:discovered:` function takes as arguments a type tag and a closure that serves as an event handler. Whenever an actor is encountered in the ad hoc network that exports a matching object, the closure is scheduled for execution in the message queue of the owning actor. An object matches if its exported type tag is a subtype of the type tag argument of `whenever:discovered:`. The `asdvertisedItem` parameter of the closure is bound to a far reference to the exported item object of another actor. The closure can then start sending asynchronous messages via this far reference to communicate with the remote object. Similar to the `export:as:` function, the discovery mechanism returns an object whose `cancel()` method cancels the registration of the closure.

VII. Evaluation

In this section, we take a step back from the technicalities of AmbientTalk and emphasise why precisely AmbientTalk's language constructs are suitable for developing mobile ad hoc networking applications. Again, we distinguish between the two most apparent hardware characteristics of mobile ad hoc networks.

A. Volatile Connections

The strictly asynchronous communication between objects owned by different actors is very suitable for mobile ad hoc networks. The built-in message queues of actors and far references decouple communication in time and synchronisation, making the application resilient to transient network failures. A traditional RPC or RMI communication model is not able to provide a similar decoupling. To abstract over temporary disconnections, objects would either remain blocked waiting

for an outstanding RPC to a disconnected object (making the application unresponsive), or the RPC would fail and the programmer potentially has to write cumbersome failure handling code for each remote message send.

The event-driven concurrency model employed by AmbientTalk has the advantage that it maps well onto the inherently event-driven nature of distributed systems. Devices may join or leave the network and messages can be received from remote devices at any point in time. In contrast to multithreaded approaches, event loops are able to restrict non-determinism to the order in which events are processed. One disadvantage that is often attributed to event loop concurrency is that it causes an *inversion of control*, i.e. the programmer must explicitly partition the application into separate event handlers (e.g. callbacks) [21]. However, the use of block closures as nested event handlers (e.g. to await future resolution or for service discovery) mitigate much of this complexity: because the event handlers are nested, the control flow remains clear and because they are full closures, they can maintain their execution context by means of their lexical scope.

B. Zero Infrastructure

In mobile ad hoc networks, services have to be discovered in the proximate environment as devices are roaming. A shared infrastructure is not always available. This network topology implies that objects should not be required to rely on a third party to discover one another. It also implies that remote objects cannot simply be named on the basis of a URL: the device hosting the remote object might not be known or may simply not be available in the local ad hoc network. To deal with these issues, each AmbientTalk actor is a topic-based publish/subscribe engine. The topics are the type tags used to classify objects in a meaningful way, independent of any particular device address, catering for anonymous interactions among objects. Because each actor can both publish services and subscribe to be notified of services that become available in the ad hoc network, no intermediary server is required.

VIII. Previous Work

As mentioned in the introduction, the AmbientTalk language described in this paper is actually an updated version of the language with the same name presented in previous work [11]. In this section, we detail how the design of the updated language – which we shall refer to as AmbientTalk/2 – differs from its predecessor, AmbientTalk/1. AmbientTalk/2's new design decisions are scrutinized below:

- **Double-Layered Object Model.** AmbientTalk/1's object model distinguishes between active and passive objects. In AmbientTalk/1, actors are modelled as ABCL/1-like active objects. However, AmbientTalk/2's concurrency model replaces the notion of actors as active objects with the notion of actors as vats, based on the vat model of the E language [10]. In this model, actors become containers of passive objects. Despite these differences, in both versions each passive object is contained within exactly one actor. However, AmbientTalk/2 allows both sequential and asynchronous message sends between passive objects whereas AmbientTalk/1 only supports sequential message passing between passive objects.
  Both versions also consider actors as the unit of distribution. However, in AmbientTalk/1, passive objects are not remotely accessible. Only the *behaviour* object of an actor can be referenced from other actors. In AmbientTalk/2, passive objects can be remotely referenced by other actors by means of far references. Hence, the revised object model of AmbientTalk/2 allows more fine-grained remote interactions between passive objects.
- **Inter-Actor Message Passing Semantics.** The parameter-passing semantics of inter-actor message sends is also slightly different in the two versions. Parameter-passing is a critical operation in AmbientTalk because it potentially allows different actors to access the same object concurrently. The parameter-passing semantics must ensure that the exclusive state access property, introduced in section V, is upheld. AmbientTalk/1 upholds this principle by *always* parameter-passing passive objects *by deep-copy*, such that both sending and receiving actor obtain an independent object. In AmbientTalk/2, objects can also be passed *by far reference*. Because far references only allow asynchronous access to an object via its actor's message queue, the exclusive state access property can be upheld. The advantage of passing objects by far reference is that remote communication is more lightweight because the object-graph does not need to be completely deep-copied.
- **Distributed memory management.** AmbientTalk/1 did not address memory management of remote objects. Remote objects were always expected to remain valid during a disconnection and to reconnect once the network connection is reestablished. Hence, this prevented the reclamation of remote objects in the face of persistent failures. In contrast, AmbientTalk/2 integrates leasing into far references so that remote objects can eventually be reclaimed if the network failure persists.

IX. Related Work

In this section, we highlight a number of programming languages, models and middleware which have influenced the design of AmbientTalk in significant ways.

Actors AmbientTalk's integration of concurrent and distributed computing with object-oriented computing is founded on the actor model of computation [16]. In the model 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. AmbientTalk does not assume the eventual delivery guarantees of messages, as in the actor model. As discussed in section VI-A, far references are allowed to expire, to properly deal with partial failures. Asynchronous communication by means of mail addresses decouples actors in time and synchronisation. This property makes the actor model in itself almost suitable for mobile networks.

The main feature lacking in the actor model to fit mobile networks is a means to perform service discovery, i.e. to acquire the mail address of a remote actor via anonymous communication. Mail addresses do not decouple sender and receiver in space. Extensions of the actor model have already tackled this issue. For example, in the ActorSpace model [22], messages can be sent to a *pattern* rather than to a mail address, and they will be delivered by the actor system to an actor with a matching pattern. The ActorSpace model, however, was conceived for traditional networks, as it relies on infrastructure to manage the matching of the patterns.

Our view of actors as communicating event loops is directly based on the communicating event loops model of the E programming language [10], [23]. AmbientTalk also inherits from E the distinction between different types of references (i.e. local references versus far references) and message passing semantics. E is designed for writing secure peer-to-peer distributed programs for open networks, but not specifically for mobile ad hoc networks. That is why AmbientTalk diverts from E's distribution model with respect to the failure semantics of far references. A network disconnection in E immediately *breaks* the far reference: any message sent after the disconnection is not stored, and the message's future is resolved with an exception. Hence, E's far references do not decouple participants in time and are not designed to express communication over volatile connections. E does not provide any built-in service discovery mechanism to engage in anonymous, space-decoupled communication. Rather, objects can acquire a far reference to a remote object by means of an explicit URI.

Futures Futures (also known as promises) are a frequently recurring abstraction in concurrent languages [5]. They serve as an essential synchronisation tool when asynchronous message passing semantics are introduced. The use of futures as return values from asynchronous message sends can be traced back to actor-based languages such as ABCL [18]. In Argus, futures were further extended to support pipelined message sends [24]. Most future abstractions support synchronisation by suspending a thread that accesses an unresolved future. This is sometimes called *wait-by-necessity* [25]. It is the E language which pioneered the `when` construct to express synchronisation on the resolution of a future (promise) in a non-blocking, event-driven manner [10].

Leasing The use of leasing to manage the lifetime of AmbientTalk's far references is derived from the use of leasing in Jini [26]. Jini is a platform for service-oriented computing built on top of Java. In Jini, resources owned by a service should be accessed by clients by means of leases. This ensures that services can gracefully deal with unexpected disconnections. For example, services may advertise themselves by registering with a lookup service, but must explicitly renew their registration by means of a lease. Otherwise, the lookup service removes the advertisement, ensuring that it does not advertise stale information of services that have become unavailable [26].

Mobile Computing Middleware Over the past few years, many middleware platforms to support mobile computing have been proposed [3]. As language designers, our goal has been to select the most appropriate techniques employed by various middleware solutions and then to embed those techniques in language features. For example, the Rover toolkit decouples communication in time by queueing RPCs [27], a feature we have embedded into the concept of a far reference.

Communication by means of shared *tuple spaces*, as originally proposed in Linda [28], has proven to be a particularly good communication model for mobile networks. This is witnessed by middleware such as LIME [29] and TOTA [30], which are based on distributed variants of the original, shared memory tuple space model. In the tuple space model, processes communicate by inserting and removing tuples from a shared tuple space, which acts like a globally shared memory. This communication is decoupled in time because processes can insert and retract tuples independently. It is decoupled in space because the publisher of a tuple does not necessarily specify, or even know, which process will consume the tuple. In the original tuple space model, synchronisation decoupling is violated because there exist synchronous (blocking) operations to extract tuples from the tuple space. However, mobile computing middleware such as LIME extends the basic model with *reactions* which are callbacks that trigger asynchronously when a matching tuple becomes available in the tuple space. In this regard, they closely correspond to AmbientTalk closures that observe futures.

The publish/subscribe communication paradigm [12] has also proven to be a fruitful basis for mobile computing middleware because it supports decoupling in time, space and synchronisation. The main difference between traditional, centralised publish/subscribe architectures and those for mobile networks is the incorporation of geographical constraints on the event disseminations and subscriptions. For example, in the location-based Publish/Subscribe (LPS) [31] architecture, 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. The Scalable Timed Events and Mobility (STEAM) middleware [2] even introduces geographical locations as first-class entities named *proximities*.

AmbientTalk's service discovery mechanism is based on the publish/subscribe paradigm. However, the matching between exported objects and subscribed event handlers is currently based entirely on connectivity as defined by the underlying network layer. For example, the matching range is entirely dependent on the fact whether e.g. Bluetooth or WiFi is used. This implies that constraining the match based on geographical parameters has to be encoded on top of AmbientTalk's discovery mechanism. In future research, we intend to integrate such geographical constraints at the language level.

X. Implementation Status and Future Work

An interpreter for the AmbientTalk language has been implemented in Java[1]. The implementation can run on the Java 2 micro edition (J2ME) platform, under the connected device configuration (CDC). This means that AmbientTalk code can be executed on PDAs and high-end cellular phones. Our current experimental setup consists of a number of QTek 9090 smartphones which communicate by means of a wireless ad hoc WiFi network.

At the implementation level, AmbientTalk interpreters communicate with one another by means of sockets. AmbientTalk's topic-based publish/subscribe service discovery mechanism is peer-to-peer and does not require a centralised repository. AmbientTalk interpreters discover one another by means of the network's support for multicast messaging. After a successful discovery, the two interpreters exchange discovery information (e.g. registered subscriptions and exported objects) in order to find a match.

One feature of AmbientTalk which has not been discussed in this paper is that it is a *reflective* programming language. It provides a powerful reflection API which can be used to extend the language from within itself. Part of our research lies in using this reflective API to develop novel language constructs [11].

XI. Conclusion

We have introduced AmbientTalk, a distributed object-oriented programming language specifically designed for composing service objects in mobile ad hoc networks. AmbientTalk's language features have been selected to mitigate the effects of the hardware characteristics inherent to mobile ad hoc networks. The language's asynchronous concurrency model allows objects to abstract over temporary network failures *without* affecting the control flow (i.e. the application remains responsive). The language's built-in publish/subscribe engine allows objects to discover one another in a peer-to-peer manner, without depending on any centralised infrastructure.

Although none of AmbientTalk's language features are novel in their own right, the contribution of AmbientTalk lies in its integration of the many interesting language features drawn from a variety of other languages and middleware into a consistent object-oriented language framework, and this in the specific domain of mobile ad hoc networks.

References

[1] M. Weiser, "The computer for the twenty-first century," *Scientific American*, pp. 94–100, september 1991.

[2] R. Meier, V. Cahill, A. Nedos, and S. Clarke, "Proximity-based service discovery in mobile ad hoc networks," in *Distributed Applications and Interoperable Systems*. Springer, 2005, pp. 115–129.

[3] C. Mascolo, L. Capra, and W. Emmerich, "Mobile Computing Middleware," in *Advanced lectures on networking*. Springer-Verlag New York, Inc., 2002, pp. 20–58.

[4] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Comput. Surv.*, vol. 21, no. 3, pp. 261–322, 1989.

[1]This implementation can be downloaded at http://prog.vub.ac.be/amop/at/download.

[5] J.-P. Briot, R. Guerraoui, and K.-P. Lohr, "Concurrency and distribution in object-oriented programming," *ACM Computing Surveys*, vol. 30, no. 3, pp. 291–329, 1998. [Online]. Available: citeseer.ist.psu.edu/article/briot98concurrency.html

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM Press, 2005, pp. 519–538.

[7] B. Liskov, "Distributed programming in Argus," *Communications Of The ACM*, vol. 31, no. 3, pp. 300–312, 1988.

[8] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109–133, February 1988. [Online]. Available: citeseer.ist.psu.edu/jul88finegrained.html

[9] L. Cardelli, "A Language with Distributed Scope," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 1995, pp. 286–297.

[10] M. Miller, E. D. Tribble, and J. Shapiro, "Concurrency among strangers: Programming in E as plan coordination," in *Symposium on Trustworthy Global Computing*, ser. LNCS, R. D. Nicola and D. Sangiorgi, Eds., vol. 3705. Springer, April 2005, pp. 195–229.

[11] 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)*, ser. Lecture Notes in Computer Science, D. Thomas, Ed., vol. 4067. Springer, 2006, pp. 230–254.

[12] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[13] H. Lieberman, "Using prototypical objects to implement shared behavior in object-oriented systems," in *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*. ACM Press, 1986, pp. 214–223. [Online]. Available: http://doi.acm.org/10.1145/28697.28718

[14] B. Garbinato and P. Rupp, "From ad hoc networks to ad hoc applications," in *Proceedings of the 7th International Conference on Telecommunications*, 2003, pp. 145–149.

[15] P. Eugster, B. Garbinato, and A. Holzer, "Pervaho: A development & test platform for mobile ad hoc applications," in *Third annual International Conference on Mobile and Ubiquitous Systems: Networking & Services*, July 2006, pp. 1–5.

[16] G. Agha, *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[17] H. Lieberman, "Concurrent object-oriented programming in ACT 1," in *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds. MIT Press, 1987, pp. 9–36.

[18] A. Yonezawa, J.-P. Briot, and E. Shibayama, "Object-oriented concurrent programming in ABCL/1," in *Conference proceedings on Object-oriented programming systems, languages and applications*. ACM Press, 1986, pp. 258–268. [Online]. Available: http://doi.acm.org/10.1145/28697.28722

[19] J.-P. Briot, "From objects to actors: study of a limited symbiosis in smalltalk-80," in *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*. New York, NY, USA: ACM Press, 1988, pp. 69–72.

[20] 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: ACM Press, 1989, pp. 202–210.

[21] P. Haller and M. Odersky, "Event-based programming without inversion of control," in *Proc. Joint Modular Languages Conference*, ser. Springer LNCS, 2006.

[22] C. J. Callsen and G. Agha, "Open heterogeneous computing in ActorSpace," *Journal of Parallel and Distributed Computing*, vol. 21, no. 3, pp. 289–300, 1994. [Online]. Available: citeseer.ist.psu.edu/callsen94open.html

[23] M. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, John Hopkins University, Baltimore, Maryland, USA, May 2006.

[24] B. Liskov and L. Shrira, "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, pp. 260–267.

[25] D. Caromel, "Towards a method of object-oriented concurrent programming," *Communications of the ACM*, vol. 36, no. 9, pp. 90–102, 1993. [Online]. Available: citeseer.ist.psu.edu/300829.html

[26] J. Waldo, "Constructing ad hoc networks," in *IEEE International Symposium on Network Computing and Applications (NCA'01)*, 2001, p. 9.

[27] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek, "Rover: a toolkit for mobile information access," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Colorado, December 1995, pp. 156–171.

[28] D. Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, Jan 1985.

[29] A. Murphy, G. Picco, and G.-C. Roman, "LIME: A middleware for physical and logical mobility," in *Proceedings of the The 21st International Conference on Distributed Computing Systems*. IEEE Computer Society, 2001, pp. 524–536. [Online]. Available: citeseer.ist.psu.edu/murphy01lime.html

[30] M. Mamei and F. Zambonelli, "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, p. 263.

[31] P. Eugster, B. Garbinato, and A. Holzer, "Location-based publish/subscribe," *Fourth IEEE International Symposium on Network Computing and Applications*, pp. 279–282, 2005.